Introduction

MCSD Training Guide: Visual Basic 6 Exams is designed for developers with the goal of certification as a Microsoft Certified Solutions Developer (MCSD). It covers both the Designing and Implementing Distributed Applications with Microsoft Visual Basic 6.0 exam (70-175) and the Designing and Implementing Desktop Applications with Microsoft Visual Basic 6.0 exam (70-176). These exams measure your ability to design and implement distributed and desktop application solutions by using Microsoft Visual Basic version 6.0.

This book is your one-stop shop. Everything you need to know to pass the exams is in here, and Microsoft has approved it as study material. You do not have to take a class in addition to buying this book to pass the exam. However, depending on your personal study habits or learning style, you may benefit from buying this book *and* taking a class.

This book also can help advanced users and administrators who are not studying for the exam but are looking for a single-volume reference on Microsoft's TCP/IP implementation.

How This Book Helps You

This book conducts you on a self-guided tour of all the areas covered by the VB6 Distributed Applications exam and the VB6 Desktop Applications exam and teaches you the specific skills you need to achieve your MCSD certification. You'll also find helpful hints, tips, real-world examples, exercises, and references to additional study materials. Specifically, this book is set up to help you in the following ways:

- **Organization.** This book is organized by individual exam objectives. Every objective you need to know for the VB6 Distributed Applications exam and the VB6 Desktop Applications exam is covered in this book. The objectives are not covered in exactly the same order as they are listed by Microsoft, but we have attempted to organize the topics in the most logical and accessible fashion to make it as easy as possible for you to learn the information. We have also attempted to make the information accessible in the following ways:
  - The full list of exam topics and objectives is included in this introduction.
  - Each chapter begins with a list of the objectives to be covered. Each objective is also identified as one that applies to the Distributed Applications exam, the Desktop Applications exam, or both.
  - Each chapter also begins with an outline that provides you with an overview of the material and the page numbers where particular topics can be found.
  - We also repeat objectives before where the material most directly relevant to it is covered (unless the whole chapter addresses a single objective).
  - Information on where the objectives are covered is also conveniently condensed on the tear card at the front of this book.
- **Instructional Features.** This book has been designed to provide you with multiple ways to learn and reinforce the exam material. Following are some of the helpful methods:

- *Objective Explanations*. As mentioned previously, each chapter begins with a list of the objectives covered in the chapter. In addition, immediately following each objective is an explanation in a context that defines it more meaningfully.
- *Study Strategies*. The beginning of the chapter also includes strategies for approaching the studying and retaining of the material in the chapter, particularly as it is addressed on the exam.
- *Exam Tips*. Exam tips appear in the margin to provide specific exam-related advice. Such tips may address what material is covered (or not covered) on the exam, how it is covered, mnemonic devices, or particular quirks of that exam.
- *Review Breaks and Summaries*. Crucial information is summarized at various points in the book in lists or tables. Each chapter ends with a summary as well.
- *Key Terms*. A list of key terms appears at the end of each chapter.
- *Notes*. These appear in the margin and contain various kinds of useful information such as tips on technology or administrative practices, historical background on terms and technologies, or side commentary on industry issues.
- *Warnings*. When using sophisticated information technology, there is always the potential for mistakes or even catastrophes that occur because of improper application of the technology. Warnings appear in the margin to alert you to such potential problems.
- *In-depths*. These more extensive discussions cover material that may not be directly relevant to the exam but which is useful as reference material or in everyday practice. In-depths may also provide useful background or contextual information necessary for understanding the larger topic under consideration.
- *Step by Steps*. These are hands-on, tutorial instructions that lead you through a particular task or function relevant to the exam objectives.
- *Exercises*. Found at the end of the chapters in the "Apply Your Knowledge" section, exercises may include additional tutorial material as well as other types of problems and questions.
- *Case Studies*. Presented throughout the book, case studies provide you with a more conceptual opportunity to apply and reinforce the knowledge you are developing. They include a description of a scenario, the essence of the case, and an extended analysis section. They also reflect the real-world experiences of the authors in ways that prepare you not only for the exam but for actual network administration as well.

◆ **Extensive practice test options.** The book provides numerous opportunities for you to assess your knowledge and practice for the exam. The practice options include the following:

- *Review Questions*. These open-ended questions appear in the "Apply Your Knowledge" section at the end of each chapter. They allow you to quickly assess your comprehension of what you just read in the chapter. Answers to the questions are provided later in the section.

- *Exam Questions*. These questions also appear in the "Apply your Knowledge" section. They reflect the kinds of multiple-choice questions that appear on the Microsoft exams. Use them to practice for the exam and to help you determine what you know and what you need to review or study further. Answers and explanations for them are provided.
- *Practice Exam*. A Practice Exam is included in the "Final Review" section. The Final Review section and the Practice Exam are discussed below.
- *Top Score*. The Top Score software included on the CD-ROM provides further practice questions.

> **NOTE**
> For a complete description of the New Riders Top Score test engine, please see Appendix D, "Using the Top Score Software."

- **Final Review.** This part of the book provides you with three valuable tools for preparing for the exam.
  - *Fast Facts*. This condensed version of the information contained in the book will prove extremely useful for last-minute review.
  - *Study and Exam Tips*. Read this section early on to help you develop study strategies. It also provides you with valuable exam-day tips and information on new exam/question formats such as adaptive tests and simulation-based questions.
  - *Practice Exam*. A full practice test for each of the exams is included. Questions are written in the styles used on the actual exams. Use it to assess your readiness for the real thing.

The book includes several valuable appendices as well, including a glossary (Appendix A), an overview of the Microsoft certification program (Appendix B), and a description of what is on the CD-ROM (Appendix C).

The Microsoft VB exams assume an elementary knowledge of VB but do not specify this knowledge in the exam objectives. For that reason, this book includes Appendix E, "Visual Basic Basics" that provides you with an overview of the elementary VB knowledge and skills that are not specified as objectives but that you will need to know in order to pass the exam.

Finally, Appendix F provides you with a list of "Suggested Readings and Resources" that provides you with useful information on Visual Basic 6.

These and all the other book features mentioned previously will provide you with thorough preparation for the exam.

For more information about the exam or the certification process, contact Microsoft:

Microsoft Education: 800-636-7544

Internet: `ftp://ftp.microsoft.com/Services/MSEdCert`

World Wide Web: `http://www.microsoft.com/train_cert`

CompuServe Forum: GO MSEDCERT

What the Designing and Implementing Distributed Applications with Microsoft Visual Basic 6.0 Exam (70-175) Covers

The Designing and Implementing Distributed Applications with Microsoft Visual Basic 6.0 Exam (70-175) covers the nine main topic areas represented by the conceptual groupings of the test objectives: Developing the Conceptual and Logical Design, Deriving the Physical Design, Establishing the Development Environment, Creating User Services, Creating and Managing COM Components, Creating Data Services, Testing the Solution, Deploying the Application, and Maintaining and Supporting an Application. Each of these main topic areas is covered in one or more chapters. The exam objectives are listed by topic area in the following sections.

Developing the Conceptual and Logical Design

Given a conceptual design, apply the principles of modular design to derive the components and services of the logical design.

Deriving the Physical Design

Assess the potential impact of the logical design on performance, maintainability, extensibility, scalability, availability, and security.

Design Visual Basic components to access data from a database in a multitier application.

Design the properties, methods, and events of components.

Establishing the Development Environment

Establish the environment for source-code version control.

Install and configure Visual Basic for developing distributed applications.

Configure a server computer to run Microsoft Transaction Server (MTS).

- Install MTS.
- Set up security on a system package.

Configure a client computer to use an MTS component.

- Create packages that install or update MTS components on a client computer.

Creating User Services

Implement navigational design.

- Dynamically modify the appearance of a menu.
- Add a pop-up menu to an application.
- Create an application that adds and deletes menus at runtime.
- Add controls to forms.
- Set properties for controls.
- Assign code to a control to respond to an event.

Create data input forms and dialog boxes.

- Display and manipulate data by using custom controls. Controls include `TreeView`, `ListView`, `ImageList`, `Toolbar`, and `StatusBar`.
- Create an application that adds and deletes controls at runtime.
- Use the `Controls` collection to manipulate controls at runtime.
- Use the `Forms` collection to manipulate forms at runtime.

Write code that validates user input.

- Create an application that verifies data entered at the field level and the form level by a user.
- Create an application that enables or disables controls based on input in fields.

Write code that processes data entered on a form.

- Given a scenario, add code to the appropriate form event. Events include `Initialize`, `Terminate`, `Load`, `Unload`, `QueryUnload`, `Activate`, and `DeActivate`.

Add an ActiveX control to the toolbox.

Create dynamic Web pages by using Active Server Pages (ASP) and Web classes.

Create a Web page by using the DHTML Page Designer to dynamically change attributes of elements, change content, change styles, and position elements.

Use data binding to display and manipulate data from a data source.

Instantiate and invoke a COM component.

- Create a Visual Basic client application that uses a COM component.
- Create a Visual Basic application that handles events from a COM component.

Create callback procedures to enable asynchronous processing between COM components and Visual Basic client applications.

Implement online user assistance in a distributed application.

- Set appropriate properties to enable user assistance. `Help` properties include `HelpFile`, `HelpContextID`, and `WhatsThisHelp`.
- Create HTML Help for an application.
- Implement messages from a server component to a user interface.

Implement error handling for the user interface in distributed applications.

- Identify and trap runtime errors.
- Handle inline errors.
- Determine how to send error information from a COM component to a client computer.

Use an active document to present information within a Web browser.

Creating and Managing COM Components

Create a COM component that implements business rules or logic. Components include DLLs, ActiveX controls, and active documents.

Create ActiveX controls.

- Create an ActiveX control that exposes properties.
- Use control events to save and load persistent properties.

- Test and debug an ActiveX control.
- Create and enable property pages for an ActiveX control.
- Enable the data-binding capabilities of an ActiveX control.
- Create an ActiveX control that is a data source.

Create an active document.

- Use code within an active document to interact with a container application.
- Navigate to other active documents.

This would include the use of client-side servers and offline batch processing, as described in detail in Chapter 9.

- Splitting processing into multiple component tiers. This opens the door to moving these tiers to different servers in the future, thus relieving the load on a particular server.

Availability

Optimum availability of a solution means that it can be reached from the most locations possible and during the greatest possible amount of time.

Availability is therefore affected by the method of deployment:

- Local
- Network
- Intranet
- Internet

The best choice for availability depends on the nature of the solution and on the usage scenarios for the solution.

For a desktop application, local availability might be perfectly acceptable. If the users are mobile laptop users, then Internet availability might be the best bet.

Security

Security is the most readily understandable issue of all those mentioned here.

For best security, a system's services should be centralized in a single physical location (to simplify administration and give less slack to would-be violators of security). Such a requirement might conflict with availability, scalability, and performance needs.

Designing VB Data-Access Components for a Multitier Application

- Design Visual Basic components to access data from a database in a multitier application (70-175 and 70-176).

The role of the data-access component is central to the n-tier application model.

In the standard *client/server* model, the components are divided into the following:

- A client component, usually residing on local workstations and containing a user interface and business rules.
- A server component that implements data access.

In the standard three-tier model, the components are as follows:

- A client component containing the user interface services
- A business logic component containing business rules
- A data-access component

In an *n*-tier model, the components are as follows:

- A client component containing the user interface services
- One or more business logic components that implement various broad areas of business rules
- One or more business data access components that mediate between the business logic components and the data-services component
- A data-services component that provides data to the data-access components

In the context of VB6 development, a data-access component will typically be a COM component exposing one or more data-aware classes, as either of the following:

- An ActiveX EXE (an out-of-process component) running on a server

or

- An ActiveX DLL (an in-process component) running on a server under Microsoft Transaction Server

NOTE

Further References For Creating Data-Access Components This book discusses how to create data-aware components in Chapter 13, "Creating ActiveX Controls," and how to create COM components in Chapter 9, "Creating Data Services: Part 2."

An important consideration for implementing individual data manipulation facilities is the decision about whether to locate them

- With the data itself. In a SQL Server environment, for example, this would take the form of stored procedures or triggers.

or

- In a separate data-access component, as discussed earlier.

Designing Properties, Methods, and Events of Components

▶ Design the properties, methods, and events of components (70-175 and 70-176).

NOTE

Further Discussion of How to Implement Component Members The following sections discuss the design of component members (properties, methods, and events). Other locations in this book discuss the actual programming of component members. See Chapter 12, "Creating a COM Component that Implements Business Rules or Logic," and Chapter 13, "Creating ActiveX Controls."

Because you will essentially implement VB-created components through classes, you will create the three types of members that are available to a VB programmer in all component programming. These three member types are as follows:

- Properties
- Methods
- Events

The following sections discuss the design of these three member types.

Designing Properties of Components

To identify component properties, you need to identify a component's attributes.

You will derive properties in different ways:

- Re-examine the usage scenarios for aspects of the objects that the application needs to track. These attributes will be good candidates for object properties. A usage scenario may speak of looking up a customer's credit balance, for example. This would imply that `Credit Balance` is a property of the `Customer` object. More subtly, it would also mean that you need a unique way to identify each customer: `Customer ID` would therefore be another property.
- As you validate the Logical Model during the component design process, you will find that items you may have initially identified as business objects in their own right are really attributes of other objects. During logical design, the nouns "Customer" and "Contact Person" may be identified as objects, for example. Further consideration will uncover the fact that a `Contact Person` is best stored as an attribute of a customer. You would therefore specify `Contact Person` as a property of the `Customer` object.

Designing Methods of Components

A method is an action or behavior that the object performs.

You can identify an object's methods by

- Reviewing the usage scenarios. Behaviors of objects that the usage scenarios mention can become methods.
- Identifying interactions between objects. When an object needs to cause a second object to do something, you have identified a method of the second object.

Designing Events of Components

An event is a message or notification that an object sends to the system and that can be detected.

One way to identify events is to review the methods that you have defined for an object. If it is clear that a method should run in response to some other behavior in the system (either from the same object or from another object), you have identified an event.

You can design events to encapsulate the behavior of objects to make it unnecessary for objects to manipulate each other directly.

If an object such as `SalesOrder` completes some action, such as completing a detail-line entry, for example, it might require a second object, `Inventory`, to update its `QuantityOnHand` property. The `Inventory` object might have an `UpdateQOH` method. Instead of requiring the `SalesOrder` object to have direct knowledge of the `Inventory` object and its methods, you could define an event for the `SalesOrder` object, `EntryComplete`, that would signal a change to the `SalesOrder` object. A programmer using your component could then decide how to react to that event, perhaps by calling the `UpdateQOH` method.

Implementing Load Balancing

▶ Implement load balancing (70-175).

Load balancing is the process by which server workload is spread among two or more server computers to prevent overload on a single server. With load balancing, a client makes a request for work as if a single server is involved; instead, more than one server can handle the request for the client.

There are two types of load balancing: *static* and *dynamic*, compared as follows:

- In static load balancing, a client always goes to the same server for tasks. The server that handles a client's requests is hard coded at the client site. Load balancing can be controlled by changing settings on the client machine or by routing new clients to new servers.

  With static load balancing, if a particular server is unavailable, the clients using that server cannot continue to work unless they are reconfigured to point to an available server.

- With dynamic load balancing, each time a client requests server-side work, a different server can handle the task. When the client makes a request, that request goes to a *referral server*, which in turn redirects the request to a server that can handle the workload. The referral server monitors the workload of each server and balances work requests based on the workload.

  With dynamic load balancing, if one server becomes available, clients do not have to be reconfigured. The referral server handles redirection of requests.

Load balancing decisions therefore have the following design implications for software solutions:

- *Performance* can be better with static load balancing, because requests do not have to be routed through a referral server.
- *Availability* can be better with dynamic load balancing, because if a server is unavailable, its requests are just shifted to another server that is available. With static load balancing, if a server becomes unavailable, the clients must be reconfigured to point to a different server.
- *Scalability* can be better with dynamic load balancing, because the referral server will automatically allocate requests depending on available resources.

In conclusion, if your system will be extremely stable, with little change anticipated in its configuration or scale and with highly dependable servers, static load balancing might be an option, because it would provide a performance advantage.

If the system needs to scale in the foreseeable future, or if there are factors that affect dependability or servers or server configuration, however, dynamic load balancing should be your choice.

CASE STUDY: SALES-ORDER ENTRY SYSTEM

NEEDS

Your company wants a new order processing system that will make available data from its legacy corporate mainframe system to users in a networked Windows NT environment. Eventually (exact timetable is uncertain) the legacy database will be converted to SQL Server.

Individual users will have state-of-the-art workstations with the fastest available processors and a lot of memory.

Management anticipates high growth for the system, both because business itself will increase, creating the need for new data-entry personnel, and because users from other departments will begin to use the system as the business operations become more integrated. Marketing, finance, and accounting groups (and possibly others as well) will access the same data at some point in the future, and their exact needs are unknown at the moment.

Again, management has not decided at what point in the system's development that the cutover from the legacy database storage to SQL Server will happen.

REQUIREMENTS

The major concerns for this scenario seem to be *scalability*, *availability*, and *extensibility*, with perhaps a secondary need for good *maintainability*.

Scalability is an issue, because rapid growth in the number of users and connections could overwhelm a single server before management has a chance to upgrade hardware to keep pace with demand.

Availability is also an issue, going hand in hand with scalability, because the business objects implemented by this system will need to be available to growing numbers of new users in the future in different locations.

The system must be *extensible* as well, because different groups of users in the future may have different needs that require different user interfaces and perhaps even enhanced sets of business rules.

A secondary requirement would then be *maintainability*, because the dynamic nature of the environment implies that there may be numerous far-reaching changes and enhancements to the system in the future.

DESIGN SPECIFICATION

Because of the high need for extensibility and maintainability, you will definitely want a multitier application divided into at least the following components:

- A client-side user interface tier, which you might consider implementing as a DHTML application over the corporate intranet. This would enable you to offload some business-rule processing from network servers to those high-powered client workstations. Less server-side activity would improve scalability, because increases in user population would create less of an increase in demand on server resources.

CASE STUDY: SALES-ORDER ENTRY SYSTEM

The DHTML solution would at the same time preserve maintainability, because the DHTML application download could be updated in the server deployment files and automatically would download the updated files when users connect with their browsers. Finally, the DHTML client-side interface could be split into several versions for different user groups, and thus provide high extensibility as well.

- One or more business logic tiers, which may split in the future depending on the needs of new groups of users. This would enhance extensibility. The business logic tiers could be implemented as out-of-process COM components for best maintainability and extensibility. When business logic required a change, you could just swap out the old components for their new and improved counterparts.
- A data-access tier separate from the data-services tier. This would help insulate the business logic tiers from changes in the back-end data-services tier when the cut-over from the legacy data services to SQL Server happens. You could write the data-access tier as a COM component exposing a set of data-aware object classes. The actual data-access technology would be private to the object classes, so the business logic clients would need no change if the data-access technology changed when the data-services tier changed.
- A data-services tier, which would initially be the legacy database engine and would at some point be replaced by SQL Server. It would be best to implement as few data integrity or business rules at this level in the legacy database engine, because any such implementation would have to be re-created in SQL Server when the cut-over happened.

CHAPTER SUMMARY

KEY TERMS

- Availability
- Business object
- Conceptual design
- Enterprise Application Model
- Enterprise Development Model
- Extensibility
- Load balancing
- Logical design
- Maintainability
- Performance
- Physical design
- Scalability
- Security
- User scenario

This chapter covered the following topics:

- Microsoft development concepts
- Deriving logical design from conceptual design
- Business objects
- Deriving physical design from logical design
- Performance, maintainability, extensibility, scalability, availability, and security
- Designing VB data-access components for a multitier application
- Designing properties, methods, and events of components
- Implementing load balancing

APPLY YOUR KNOWLEDGE

Review Questions

1. Where does the user interface component of an application normally reside in a system's architecture? Why?
2. Where does the data-access interface component of an application normally reside in a system's architecture? Why?
3. How can striving for a "thin client" improve maintainability of a software solution but possibly hurt scalability?

Exam Questions

1. A software solution that you will implement on your corporate network will need to enforce the following restrictions:
   - The value of a new customer order as well as the sum of the customer's outstanding balance cannot exceed the customer's assigned credit limit.
   - The customer must pay shipping charges for any order weighing more than 16 pounds.

   How should you implement these constraints?

   A. As stored procedures and triggers in the database

   B. As part of an independent data-access component

   C. As part of a business-rules component

   D. As part of the user-interface component

2. You are implementing a system that runs with data from a legacy database. At some point in the future, the system will cut over to a SQL Server database. What is your best choice for implementing the data-access component?

   A. Implement as a COM component exposing data-aware classes for accessing the data from any business logic tiers. Reprogram, recompile, and swap out this component on the network server when the cut-over happens.

   B. Implement as a COM component running on users' workstations. Change the network-based setup package when the cut-over happens and request users or system administrators to rerun the setup.

   C. Implement as executable running on users' workstations. Email the new executable to users or system administrators when the cut-over happens, including instructions about where to copy the executable.

   D. Implement as stored procedures in the original database. Translate these stored procedures to SQL Server stored procedures and triggers in the new system.

   E. Implement as a downloadable component in a browser-based user interface. When the data-access method changes, change the component and place it on the intranet server where the users' browsers will automatically download it and update the users' systems.

3. You are going to implement a new call-tracking system for users in the central office. The design calls for the following components:
   - A user-interface services tier
   - A business-rules tier

APPLY YOUR KNOWLEDGE

- A data-access tier
- A data-services tier

Your best choice for implementing the data-access tier would be as

A. A DHTML application

B. An IIS application

C. A COM component on the network server

D. A standard executable distributed on user workstations

4. You are implementing a solution for a custom job estimating system.

   Essentially, the users need to use the system to enter requirements from customers, and the system can then look up historical information that most closely matches the customer requirements, providing an overall cost estimate based on past jobs that match the current one.

   The users are all located at the company's headquarters and have state-of-the-art workstations connected to the same Windows NT network.

   Customer inquiry volume is very high, and users need to be able to enter information quickly and get results immediately so that they can quickly give quotes to customers and then move on to the next call.

   The organization expects high growth in the short- to mid-term and therefore may double or triple the number of users on the system within the next year.

   You decide to implement the system as a three-tier system, with

   - SQL Server as the data-services tier on the back end
   - A data-access tier residing on the network as a COM component exposing data-aware classes to perform general data access routines
   - A "thick client" tier that resides as a stand-alone executable on users' workstations and combines user interface and business logic

   Which of the following concerns does this solution address? (Pick all that apply.)

   A. Maintainability

   B. Scalability

   C. Security

   D. Extensibility

   E. Performance

   F. Availability

5. A software solution that you will implement on your corporate network will need to enforce the following restrictions:

   - Each sales order must be assigned to exactly one customer and one customer alone.
   - No two sales orders in the system can have the same order number.

   How should you implement these constraints?

   A. As stored procedures and triggers in the database

   B. As part of an independent data-access component

   C. As part of a business-rules component

   D. As part of the user-interface component

APPLY YOUR KNOWLEDGE

6. You are implementing a solution that will allow your organization's employees to enter their weekly hours into the corporate time-and-billing database.

   Some of the employees will be at remote locations and can only connect to your network remotely from their PCs. There is also a wide variation in the power of the hardware that is available on employee workstations.

   You decide to implement this solution as an IIS application over the corporate Web server.

   Which of the following concerns does this solution satisfy? (Pick all that apply.)

   A. Maintainability

   B. Scalability

   C. Availability

   D. Extensibility

   E. Performance

   F. Security

7. A software solution that you will implement on your corporate network will need to enforce the following restrictions:

   - All alphanumeric data should be stored in uppercase.
   - Product dimensions will be displayed and entered in centimeters, with up to three decimal places of precision.

   How should you implement these constraints?

   A. As stored procedures and triggers in the database

   B. As part of an independent data-access component

   C. As part of a business-rules component

   D. As part of the user-interface component

8. Which considerations might affect load balancing decisions? (Pick all that apply.)

   A. Maintainability

   B. Scalability

   C. Security

   D. Extensibility

   E. Performance

Answers to Review Questions

1. The user interface component of an application normally resides on client workstations, because it is the part that actually provides the user connection to the rest of the system. See "Designing VB Data-Access Components for a Multitier Application."

2. The data-access interface of an application normally resides on the server so that it can provide consistent service and server resource management. It is also more maintainable if it resides in a single central location. See "Designing VB Data-Access Components for a Multitier Application," "Availability," and "Maintainability."

3. A "thin client" (that is, a workstation client that implements as little functionality as possible) can improve a software solution's maintainability, because more processing will be implemented on servers. Such centralization of functionality means that there are less locations where software changes have to be distributed. By putting more processing burden on servers, however, performance can degrade dramatically as more demand is placed on the server through the addition of new users. See "Maintainability" and "Performance."

APPLY YOUR KNOWLEDGE

Answers to Exam Questions

1. **C.** A business-rules component is the best place to enforce constraints such as customer credit enforcement and shipping charge rules. Both constraints are clearly part of the way that the organization does business. These rules could change over time, or even change in different directions for different parts of the same organization. For more information, see the sections titled "Maintainability," "Designing VB Data-Access Components for a Multitier Application," and this chapter's case study.

2. **A.** A COM component with data-aware object classes is the best solution for implementing a data-access component whose data-access platform will change in the future. This provides the best maintainability, because it isolates the other tiers from needing to be aware of the type of data access that's needed. B (COM component on workstations) would be an inferior solution, because it would give you the proverbial "maintenance nightmare" by requiring many individuals (at assuredly varying levels of system competence) to perfectly perform the same action at the same time. C is even more laughably inadequate, for the same reasons. D is inappropriate because it mixes the functions of the data-services tier itself (data-integrity rules) with the data-access methods specific to the application itself. It also might be trickier than you think to transfer all the rules in stored procedure from one DBMS platform to another. E doesn't require any conscious user interaction, but the location of the data-access component itself will probably be a performance drag, because all data access will have to take place across network boundaries. For more information, see the sections titled "Maintainability," "Performance," "Designing VB Data-Access Components for a Multitier Application," and this chapter's case study.

3. **C.** A COM component with data-aware classes would be the best choice for implementing a data-access tier for centralized users. Although an IIS application could be a *part* of the solution described in the question, it is not really appropriate for a data-access tier, but rather perhaps a user-interface tier. The same could be said of a DHTML application and of a standard client-side executable. For more information, see the sections titled "Maintainability," "Performance," "Designing VB Data-Access Components for a Multitier Application," and this chapter's case study.

4. **B, E.** The somewhat surprising "thick client" solution that implements user interface and business rules on the client workstation best addresses concerns of scalability and performance. The key to both these considerations is the fact that such a design will offload a lot of processing to client workstations, which in the scenario are described as being quite powerful (and so capable of handling the extra work). The network will be less likely to bottleneck because more users are quickly added to the system (so scalability is served); performance will also degrade less, because individual workstations will be more responsible for the performance for each user. This solution definitely does not address maintainability, because it will be harder to make changes to business-logic components that are scattered over many user workstations and are intertwined with the user interface. For more information, see the sections titled "Maintainability," "Scalability," and "Performance."

5. **B.** An independent data-access component is the best place to enforce rules of referential integrity, such as those mentioned in this scenario. If there were no separate data-access component, A (stored procedures and triggers) would be the best choice.

APPLY YOUR KNOWLEDGE

In this case, however, because an independent data-access component is one of our options, you should normally favor that location. For more information, see the section titled "Designing VB Data-Access Components for a Multitier Application" and this chapter's case study.

6. **A, C, E**. An IIS application solution would favor maintainability (it's centralized and therefore easily changeable), availability (any user with a Web connection and any major Web browser), and performance from the user's point of view, although not for the server, because the server will be doing most of the work. It might not be the best solution for scalability, because it is more server intensive than a DHTML application, and it's probably not very easily extensible, either, because its logic is centered around one type of solution. Security could also be an issue, because users are connecting to your server and your corporate data through the Internet. For more information, see the sections titled "Maintainability," "Availability," "Performance," "Scalability," "Extensibility," and "Security."

7. **D**. The constraints mentioned in the scenario are clearly data-entry and display rules, and therefore are best implemented as part of the user interface component. For more information, see the section titled "Designing VB Data-Access Components for a Multitier Application."

8. **B, E, F**. Availability, extensibility, and performance considerations affect decisions about whether to make load balancing static or dynamic. For more information, see the section titled "Implementing Load Balancing."

OBJECTIVES

This chapter helps you prepare for the exam by covering the following objectives:

Establish the environment for source-code version control (70-175 and 70-176).

- The first objective requires you to have some knowledge of Visual SourceSafe 6.0, which is bundled with VB 6.0, Enterprise Edition. You can use Visual SourceSafe to manage the files for source code through different versions of a single project. You can also use Visual SourceSafe to prevent conflicts and confusion among developers who are working on the same project at the same time.

Install and configure Visual Basic for developing desktop/distributed applications (70-175 and 70-176).

- Although the second objective is different for the two exams (specifying desktop applications for the 176 exam and distributed applications for the 175 exam), it amounts to the same requirement for both exams: knowing the features available in the various editions of VB6 and Visual Studio 6.

CHAPTER 2

Establishing the Development Environment

OUTLINE

Implementing Source-Code Control with Visual SourceSafe **45**

The Nature of a Visual SourceSafe Project 46

The Visual SourceSafe Database 46

Visual SourceSafe Administrator 47

Visual SourceSafe Explorer 51

Installing and Configuring VB for Developing Desktop and Distributed Applications 65

Chapter Summary **67**

STUDY STRATEGIES

- Make sure you have Visual SourceSafe installed on your system (both Visual SourceSafe Administrator and Visual SourceSafe Explorer). You will need access to Visual SourceSafe Administrator (know the Admin password) to do some of the activities suggested here.
- In Visual SourceSafe Administrator, add a new user (Exercise 2.1) and archive and restore a project (see Exercise 2.4).
- Create a new Visual SourceSafe project, either directly from VB6 (Exercise 2.2) or by using Visual SourceSafe Explorer (Exercise 2.3). Add files to the project.
- In Visual SourceSafe Explorer or VB, practice checking out and checking in project files (Exercises 2.2 and 2.3).
- Review the project source-code management scenarios discussed throughout this chapter. Most of the exam questions relating to Visual SourceSafe will be scenario-based.
- In Visual SourceSafe Explorer, share a project's files to a new project and branch the copies (Exercise 2.3).
- Familiarize yourself with the concepts of pinning and merging, as discussed in the sections titled "Pinning an Earlier Version of a File for Use in the Current Version of a Project" and "Merging Two Different Versions of a File."
- Review and compare the features of the Learning, Professional, and Enterprise editions of VB6 as discussed in the section titled "Installing and Configuring VB for Developing Desktop and Distributed Applications."

INTRODUCTION

When the exam objectives refer to "establishing the development environment," they refer to two basic concepts:

- Source-code control
- Knowledge of the general capabilities of the various VB editions

This chapter covers both of these general concepts by discussing Visual SourceSafe and the VB6 editions.

IMPLEMENTING SOURCE-CODE CONTROL WITH VISUAL SOURCESAFE

▶ Establish the environment for source-code version control.

Version control is a term that describes the actions that software developers must take to keep track of the physical files that go into various versions of a software product. Version control is basically concerned with two types of control:

- Keeping track of changes to files over time and matching the various versions of a file with versions of a software product.
- Managing concurrent changes made by multiple developers to a project's files and keeping their changes from conflicting with each other.

Visual SourceSafe has three basic components that you need to know about to understand how it works to implement version control:

- **The Visual SourceSafe database.** This is the repository for the various versions of source code and other files that Visual SourceSafe administers. The Visual SourceSafe database stores the files in a compressed, proprietary format. It must be visible to all developers and administrators who need to use Visual SourceSafe.
- **Visual SourceSafe Administrator.** This application enables one or more administrators to manage the Visual SourceSafe database, to manage users of Visual SourceSafe, and to define the users' rights in general and also for particular projects.

- **Visual SourceSafe Explorer.** This application resides on each developer's workstation. It is the main vehicle that developers use to manage the source code stored in the Visual SourceSafe database.

Visual SourceSafe is typically installed as an option with VB or Visual Studio. You should make sure that the Visual SourceSafe database is installed in a location visible to all who will need to use it (probably on a network drive). The default name for the installed folder is VSS. The exact location of this folder will depend on the choices you make during Visual SourceSafe installation.

The Nature of a Visual SourceSafe Project

A Visual SourceSafe project typically corresponds to the physical files in a development project. Developers usually think of a Visual SourceSafe project as containing source-code files (in the case of a VB project, file types would include VBP, FRM, BAS, CTL, and so forth).

A Visual SourceSafe project can contain any files important to the project, however—such as email documents containing correspondence, design, specification, and project-management files in word processor format or in the formats of other software tools, as well as the latest compiled version of the executable files created from the source code.

The Visual SourceSafe Database

The Visual SourceSafe database contains all the files that Visual SourceSafe maintains. Visual SourceSafe stores the files in a proprietary, compressed format.

You will find the Visual SourceSafe database under the main Visual SourceSafe directory (usually named VSS) in a folder named Database.

You will see several files under this folder and a lot of other folders containing many other files. You should never try to manipulate the contents of the Visual SourceSafe database manually, because Visual SourceSafe has its own internal scheme for managing the database.

It is possible to have multiple Visual SourceSafe databases set up in the same environment. When administrators or developers open the Visual SourceSafe Administrator or the Visual SourceSafe Explorer (as mentioned in the following sections), they have a choice of Visual SourceSafe databases.

Figure 2.10 in the section titled "Using Visual SourceSafe Explorer" illustrates the choice of databases for a user in Visual SourceSafe Explorer.

A Visual SourceSafe administrator can also archive and restore Visual SourceSafe databases, as discussed further in the section titled "Archiving and Restoring Visual SourceSafe Databases."

Visual SourceSafe Administrator

You can use Visual SourceSafe Administrator to

- Set up and remove users of Visual SourceSafe and set their Visual SourceSafe passwords.
- Assign rights to users of Visual SourceSafe for specific projects or in general.
- Set options for Visual SourceSafe projects.
- Archive and restore Visual SourceSafe projects in specific Visual SourceSafe databases.
- Set options for the general behavior of Visual SourceSafe.

To answer SourceSafe-related questions in the exam, you will need to know about some of these administrative functions:

- Archiving and restoring VSS databases
- Setting up VSS users

The following sections discuss these two topics.

Archiving and Restoring Visual SourceSafe Databases

You can use Visual SourceSafe Administrator to archive projects that have fallen out of use. You can also restore projects if you need them back again.

To archive a Visual SourceSafe project, you should follow these steps:

STEP BY STEP

2.1 Archiving a Visual SourceSafe Project

1. Open Visual SourceSafe Administrator.
2. Choose Archive, Archive Projects from the Visual SourceSafe Administrator menu.
3. The Choose Project to Archive dialog box shows a tree of Visual SourceSafe projects and subprojects. Select a project to archive from the tree and click OK. This will take you to Screen 1 of the Archive wizard, shown in Figure 2.1.
4. If you want to archive more projects, click the Add button on the first screen of the Archive wizard to choose additional projects, repeating this step until you have indicated all the projects that you want to archive. After you have finished, click the Next button.
5. On Screen 2 of the Archive wizard, use the Browse button to define a location and a name for the file where you want to save the archived projects' information. (The archive file's extension, .SSA, will be supplied automatically). See Figure 2.2.
6. Choose one of the three archive options:
   - Save data to file
   - Save data to file, then delete from database to save space
   - Delete data permanently

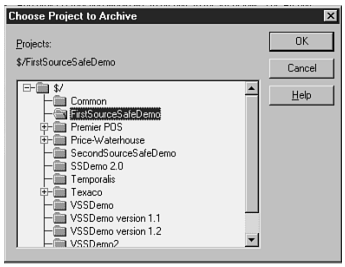


FIGURE 2.1▲
Adding a project to archive from the Archive wizard.

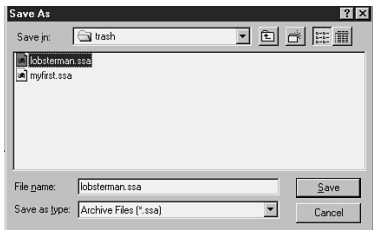
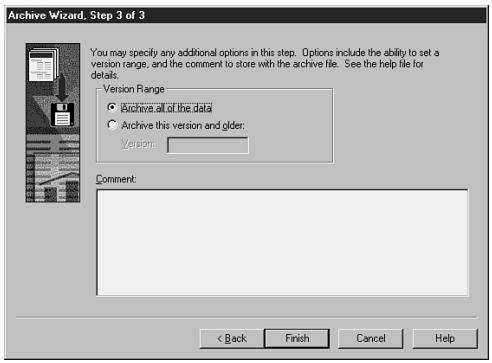


FIGURE 2.2▲
Specifying an archive file from Screen 2 of the Archive wizard.

7. Set any other archiving options on Screen 3 of the Archive wizard, and then click the Finish button (see Figure 2.3).



◀**FIGURE 2.3**
Screen 3 of the Archive wizard.

8. Archive wizard will then archive your data and inform you of its success (see Figure 2.4).

FIGURE 2.4▲
Archive wizard's closing prompt.

To restore a Visual SourceSafe project, follow these steps:

STEP BY STEP

2.2 Restoring a Visual SourceSafe Project from Archive

1. Choose the Archive, Restore Projects menu option.
2. On Screen 1 of the Restore wizard, browse to the Visual SourceSafe archive (SSA) file that you want to restore, and click the Next button (see Figure 2.5).
3. On the screen for step 2, choose the projects from the archive that you want to restore (see Figure 2.6).

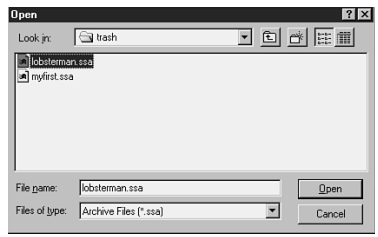
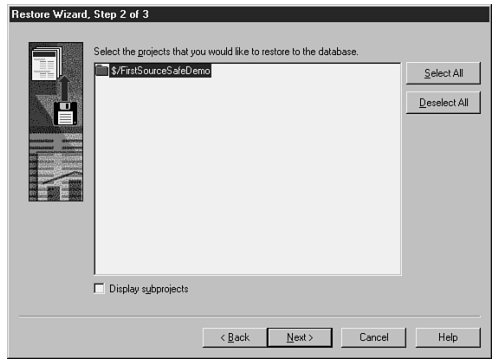


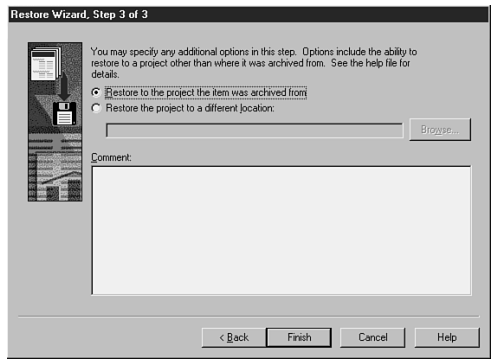
FIGURE 2.5▲
Specifying an existing Archive file to restore from.

FIGURE 2.6▶
Specifying the projects that you want to restore.



4. On the screen for step 3, select where you would like the project restored. Click the Finish button and wait for the Restore wizard's notification of success (see Figure 2.7).

FIGURE 2.7▶
Step 3 of the Restore wizard.



You can use the Archive/Restore feature of Visual SourceSafe to transfer projects between two Visual SourceSafe databases. To transfer a project from one Visual SourceSafe database to another, follow these steps:

1. Archive the project from the database where it originally existed (the source).
2. Restore the archived project to the destination database.

Using Visual SourceSafe Administrator to Set Up and Maintain Users

Although Visual SourceSafe Administrator can leave most general Visual SourceSafe settings at the installed default values, there is one task that you must perform in Visual SourceSafe Administrator for Visual SourceSafe to function properly: setting up users.

Developers cannot have access to Visual SourceSafe Explorer and the projects contained in the Visual SourceSafe database unless they have a logon account for the Visual SourceSafe database.

To set up a user for access to the Visual SourceSafe database, follow these steps:

STEP BY STEP

2.3 Setting Up a User with Visual SourceSafe Administrator

1. Open Visual SourceSafe Administrator and choose Users, Add User from the main menu to bring up the Add User dialog box (see Figure 2.8).
2. Fill in the logon ID for the user whom you are setting up.
3. Assign a password.
4. If you don't want the user to be able to change information stored in the Visual SourceSafe database, check the Read Only box as well.
5. Click OK to close the dialog box and create the user account.

FIGURE 2.8
Adding a SourceSafe user account.

NOTE

Visual SourceSafe Security and Windows Security Visual SourceSafe Explorer will automatically try to use the Windows user ID and password to log on a user who runs Visual SourceSafe Explorer. If the user ID and password match a user ID and password in Visual SourceSafe, the user will receive no logon prompt to Visual SourceSafe Explorer.

Visual SourceSafe Explorer

When you run Visual SourceSafe Explorer, the first thing that you see depends on how the administrator set up your Visual SourceSafe account:

- If the Visual SourceSafe administrator set up a user with the same name and password as your Windows logon and password, Visual SourceSafe Explorer will proceed immediately to the main screen.
- Otherwise, Visual SourceSafe Explorer will present you with the logon screen shown in Figure 2.9. In this case, you will need to supply a valid Visual SourceSafe logon/password combination as set up by the Visual SourceSafe administrator.

FIGURE 2.9▲
Logging on to Visual SourceSafe Explorer.

You can also click the Browse button on the logon screen to choose from all the Visual SourceSafe databases recognized by Visual SourceSafe on your system (see Figure 2.10). You can also wait until you see the main Visual SourceSafe screen, and select the File, Open SourceSafe Database from the menu to choose a different database.

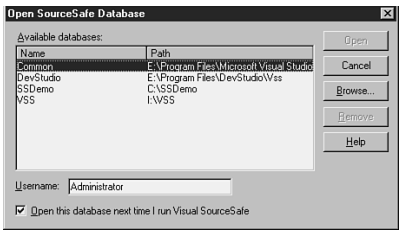


FIGURE 2.10▲
Choosing a Visual SourceSafe database.

Following is a list of some of the things that you can do with a project's files in Visual SourceSafe Explorer:

- **Set working folder.** You can specify the physical location on your system where you want to put a project's files when you view them or work with them.
- **Check out.** You can get a writable copy of a file from the SourceSafe database. Normally, only one developer at a time can check out a given file.
- **Check in.** You can return a modified copy of a file to the SourceSafe database. The modified file is now available for other developers to check out.
- **Get working copy.** You can get a read-only copy of a file from the Visual SourceSafe database. Anyone can get working copies of a file, even if someone else currently has it checked out.
- **Label.** You can designate versions of one or more files in a project with a label of your choosing. You can get copies of all the files designated with a single label and thus reproduce a particular version of a project.
- **Share.** You can share the same copy of one or more files between various projects.
- **Branch.** You can break the link between shared copies of the same file so that you can then develop the copies independently.

- **Pin.** You can freeze a particular version of a file or project so that no more changes can be made to it.
- **Difference.** You can view the differences between two different versions of a file.
- **Merge.** You can merge two different versions of a file together. You can view each difference and decide how to merge.

The following sections discuss many of these activities.

Because certification exam questions on Visual SourceSafe focus on concepts rather than techniques, some of the activities are not discussed, or are only described in general.

Creating a New Project Folder in Visual SourceSafe

The structure for storing Visual SourceSafe projects looks somewhat like the Windows File Folder tree as presented by Windows Explorer (see Figure 2.11).

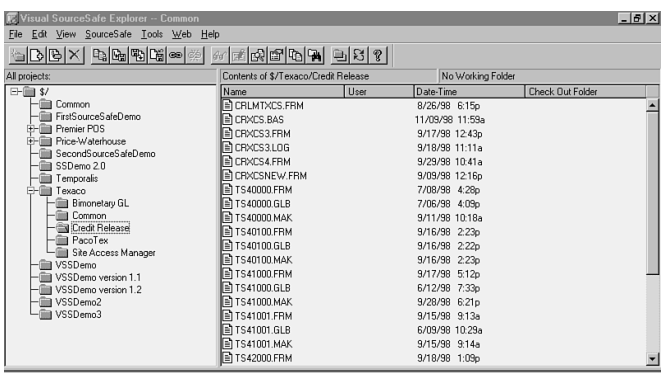


FIGURE 2.11
The project tree in Visual SourceSafe Explorer.

All projects in the Visual SourceSafe database are stored under a root folder (denoted by a folder labeled "$/" at the top of the tree hierarchy).

To add a new project to Visual SourceSafe, follow these steps:

STEP BY STEP

2.4 Adding a New Project to Visual SourceSafe

1. Run and log on to Visual SourceSafe Explorer.
2. Select either the root folder ($/) or an existing project folder. The project you create will belong under the folder that you select.
3. Choose File, Create Project from the menu.
4. Under the Create Project dialog box, assign a name to the project and enter any comments. Click the OK button to create the new project's folder in Visual SourceSafe (see Figure 2.12).
5. Add files to the project (you may do this at a later time). The next section describes this action in detail.

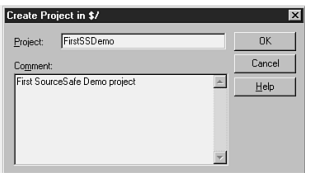


FIGURE 2.12▲
Creating a new project folder (figure shows both the dialog box to create the folder and the newly created folder).

Adding Files to a Visual SourceSafe Project

Before you can use a Visual SourceSafe project, you must indicate which files the Visual SourceSafe database will store in the project. To add files to a Visual SourceSafe project, follow these steps:

STEP BY STEP

2.5 Adding Files to a Visual SourceSafe Project

1. Select the folder of the project that you want to add files to.
2. Choose File, Add Files from the menu.
3. In the resulting Add File dialog box, browse to and select the files that you want to add to the project. You can either add files one at a time with the Add button, or you can select several simultaneously with the mouse and the Ctrl key, and then click Add (see Figure 2.13).
4. Click Close on the Add File dialog box. SourceSafe will give you the chance to add a comment, and then you will see the newly added files in the right-hand pane of Visual SourceSafe Explorer (see Figure 2.14).

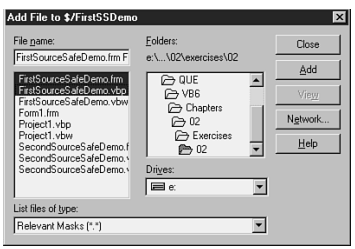


FIGURE 2.13▲
Adding files to a project folder.

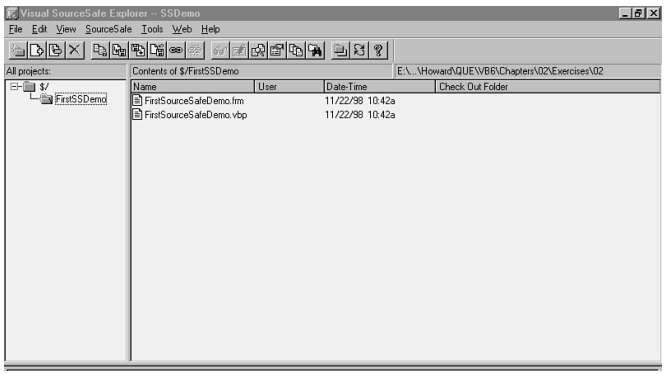


FIGURE 2.14
Newly added files under a SourceSafe project.

Setting the Working Folder for a Project

The Visual SourceSafe folders that you see when running Visual SourceSafe Explorer are *not physical* folders. The Visual SourceSafe folders are *only logical* folders for presenting the information about a project in the Visual SourceSafe environment.

To check out a project's files from Visual SourceSafe, however, you need to associate the project's Visual SourceSafe folder with a physical folder in your system. The physical folder that will hold copies of files is called the *Working Folder* for the Visual SourceSafe folder.

If you have defined no Working Folder for a project, Visual SourceSafe won't let you check out files from that project or get working copies.

To create a Working Folder for a project, choose the project in the SourceSafe tree and select File, Set Working Folder from the main menu. Use the resulting dialog box to browse to a folder where you would like your copies of the project's files to reside. You can define a new folder on-the-fly at this point.

Each developer who uses Visual SourceSafe Explorer must define his or her own Working Folder for a project. Therefore, each developer can have a separate copy of the project, usually on a local drive or on a personal area of a shared network drive.

The existence of multiple copies of the same project files may sound to you like an opportunity for a lot of confusion. That's where the source-code management activities of the next section enter into the picture.

NOTE

Multiple Checkouts Possible on the Same File Under certain circumstances, it is possible for more than one developer to have the same file checked out at the same time. See the note in the section titled "Merging Two Different Versions of a File" for more information.

Checking Out, Checking In, and Getting Working Copy

These three actions are the backbone of day-to-day Visual SourceSafe activity. You employ them in the following manner:

- You check a file out when you intend to make changes to the file. The Visual SourceSafe checkout process places a writable copy of the file in your Working Folder. Usually, no one else can check the file out until you check the file back in.

  You can check a file out in Visual SourceSafe Explorer by selecting the file in the Visual SourceSafe project and choosing SourceSafe, Check Out from the menu. If no one else has the file checked out, SourceSafe gives you the chance to make a comment, checks the files out to you, and changes the file's icon to show that it is checked out (see Figure 2.15).

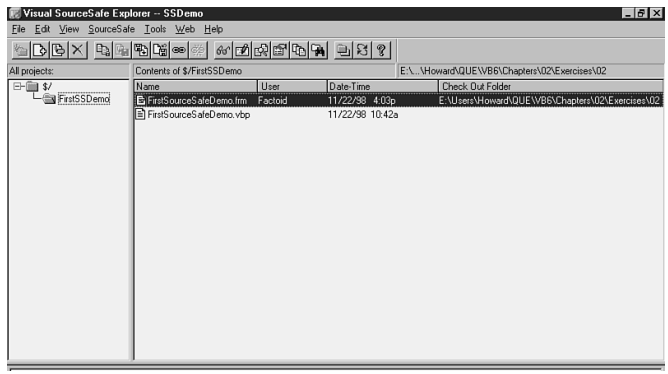


FIGURE 2.15▶
The appearance of a checked-out file in SourceSafe Explorer.

- When you check the file in, Visual SourceSafe stores the changes that you have made to the Visual SourceSafe database.

  You check the file in by selecting it under the Visual SourceSafe project and choosing SourceSafe, Check In from the menu. A dialog box will give you a chance to make comments (see Figure 2.16). The dialog box also enables you to decide whether to keep the file checked out to your account and whether to remove the local copy of the file after you have checked it in.

  Visual SourceSafe also changes the permissions on the copy of the file in your Working Folder to read-only.

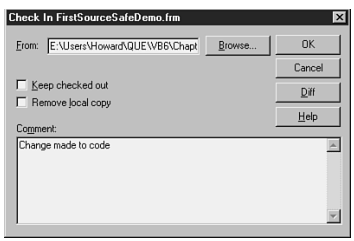


FIGURE 2.16▲
Checking a file in.

When you check the file in, you also have the option of keeping the file checked out, as just mentioned. This has the effect of refreshing Visual SourceSafe's copy of the file, but keeping it checked out to you. In this case, the copy in your Working Folder remains writable.

On the other side of the coin, you also have the option of checking the file in and then removing the local copy that you were working on (also just mentioned). In this way, the most recent version of the file is only stored in the Visual SourceSafe database.

- If you choose Get Working File rather than Check Out, Visual SourceSafe places a read-only copy of the file in your Working Folder.

When developers are working together on a multifile project, the development cycle with Visual SourceSafe goes through something like the following steps:

1. You use Get Working Copy to place the most current copies of all files needed to compile the project on your local system.
2. You check out the file or files that you personally will modify.
3. Because you have copies of all the project's files (some readable from step 1, and some writable from step 2), you can always compile the whole project to test the changes that you are making to the files that you are responsible for.
4. When it is time to compile and test the entire project, you and the other developers can check in all your files (making sure, of course, that the versions you check in run with the rest of the project).
5. If you need to, you can keep your files checked out to continue working.
6. You can also periodically refresh your local read-only copies of the files that other developers are working on. Just run the Get Working Copy action on the files that you don't have checked out.

Source-Code Labeling and Version Numbers

Visual SourceSafe can maintain a history of each check-in of a file as well as the actual contents of all the versions of a file when the versions were checked in.

Visual SourceSafe assigns an incrementing version number (always a whole number) to each checked-in version of a file. Visual SourceSafe also maintains a date-time stamp that helps to identify the version.

Visual SourceSafe's automatically assigned version numbers can be useful for tracking the modification history of a file.

The most useful version-tracking feature of Visual SourceSafe is the user-assigned label. A user of Visual SourceSafe can assign a label to all the files in a project at a given moment. Later Visual SourceSafe operations can manipulate the files as a group.

Assigning labels is one of the primary ways that you can continue to identify a particular version of a product, even after subsequent changes.

Although it is possible to assign labels to individual files in Visual SourceSafe, it is more typical to label an entire project. To label the current version of a project, follow these steps:

STEP BY STEP

2.6 Labeling the Current Version of a VSS Project

1. In Visual SourceSafe Explorer, select the project.
2. Choose File, Label from the Visual SourceSafe menu.
3. In the resulting Label dialog box, type the label that you want to give to this version of the project. Typically, this label will be something along the lines of "Version 1.0" (see Figure 2.17).
4. Click OK to apply the label.

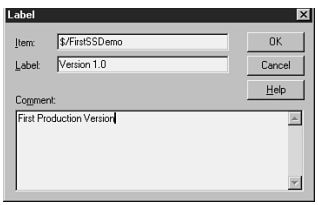


FIGURE 2.17
Labeling a project.

Source-Code Sharing

When you want to share the same source code between two or more projects, you can designate a file or files as shared. The shared files will appear in the Visual SourceSafe folder for all projects where they have been shared.

To share files, follow these steps:

STEP BY STEP

2.7 Sharing Files from a VSS Project

1. In Visual SourceSafe Explorer, select the project to which you want to share the files.
2. Choose SourceSafe, Share from the Visual SourceSafe menu.
3. In the resulting dialog box, use the Projects tree to navigate to the Visual SourceSafe project whose files you want to share. Then choose the desired files so that their names appear in the Files list (see Figure 2.18).
4. Click the Share button.
5. The selected files are now shared between the two projects. Notice that the document icon beside a shared file appears doubled (see Figure 2.19).

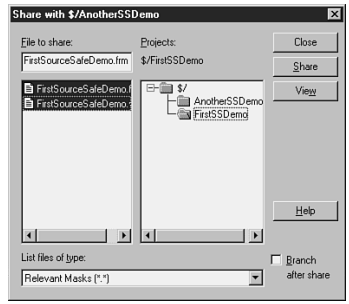
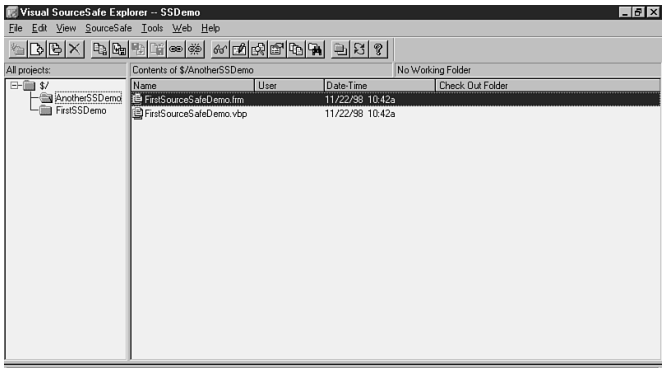


FIGURE 2.18▲
Sharing files.



◀FIGURE 2.19
Files icons showing that the files are shared.

After programmers make changes to a shared file, the changes will appear automatically in all the projects where the file has been shared.

If you want to find out where a file is shared, you can right-click the file in the Visual SourceSafe window, choose Properties from the shortcut menu, and then choose the Links tab from the Properties dialog box.

Source-Code Branching

Sharing is appropriate when you want all subsequent changes to a file to be reflected in each project where the file is shared.

You may want to put a copy of a file into a project, however, and then modify the copy independently of the original.

Making a Visual SourceSafe copy of a shared file independent of the original is called *branching*.

To branch a shared file, follow these steps:

STEP BY STEP

2.8 Branching a Shared File in VSS

1. Select the file in the Visual SourceSafe window.
2. Choose SourceSafe, Branch from the Visual SourceSafe menu (see Figure 2.20).
3. You will notice that the file's icon changes: It no longer appears doubled, but rather now appears as a simple document icon.

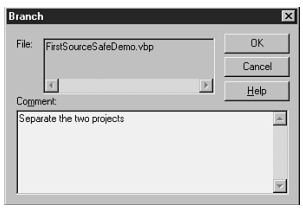


FIGURE 2.20
Branching a file.

The branched file now has an independent life, and any changes you make to the file will *not* be reflected in the original file.

Branching can be appropriate in cases where there is an existing distributed version of a project that needs to be fixed, while at the same time in a separate project you want to begin working on a major new version. Imagine the following scenario, for example:

- Version 1.0 is the current production version of your application.

- Development team A (which may be only one person, of course) will be responsible for ongoing patches to version 1.0 and will work on version 1.1.
- Development team B (once again, this might be a single person—in fact, it might be the person who is development team A, but just wearing a different hat) is responsible for the new major release and will work on version 2.0.

To implement the preceding scenario with Visual SourceSafe, you should take the following steps:

1. Label the project containing the files for version 1.0 if you haven't done so already.
2. Create a new project folder for version 2.0.
3. Share the version 1.0 files to the version 2.0 project, and immediately branch the shared files.
4. You now have two independent projects based on version 1.0, but the projects can now diverge along independent development paths.

You can branch a file at the same instant that you share it by checking the Branch after Share check box on the Share dialog box.

Suppose, for example, that your project contains a module that processes sales orders. You need to create a new module that processes purchase orders. A lot of the general logic will be the same as the logic for processing sales orders, but the details will differ.

In this case, you will want to make a copy of the sales-order processing module as a starting point for the purchase-order processing module. As soon as you make the copy, you want to begin making independent changes. You therefore use the Branch after Share option at the instant that you share the files for this module.

Pinning an Earlier Version of a File for Use in the Current Version of a Project

When you pin a version of a file, you mark that version of the file as the version that you want to use when you check out that file for a particular project.

Pinning therefore enables you to specify an earlier version of a file for use in the current project. This is helpful in a scenario such as the following:

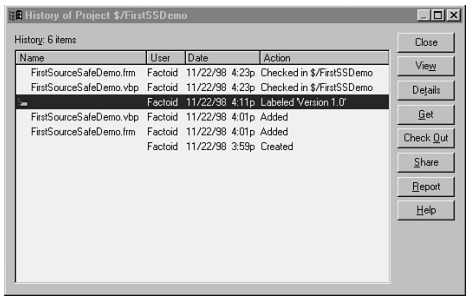
- Version 1.0 is the current production version of the product. Version 1.0 files have been labeled in the project. They are modules A, B, C, and D.
- Version 2.0 is in development. It uses newer versions of modules A, B, C, and D. Beta testing has begun, and these files have been labeled with the version 2.0 stamp.
- In the meantime, version 1.0 is still in production and there is a need to distribute version 1.1 that will fix problems in module C of version 1.0.
- You want to be able to make changes to the bad module C of version 1.0. At the same time, you need to move forward with development of module C of version 2.0.

You can use a combination of sharing, pinning, and branching to fulfill your needs (see Step by Step 2.9).

STEP BY STEP

2.9 Using Share-Pin-Branch to Create a "Service Pack" Version While Concurrently Working on a Newer Version

1. Share the original version 1.0 project files.
    - Select the project's folder in Visual SourceSafe Explorer.
    - Choose Tools, Show History from the Visual SourceSafe menu to bring up the Project History Options dialog box.
    - Make sure that the Include Labels option is checked and click the OK button. This will display the History of Project dialog box.
    - In the History of Project dialog box, select the version labeled 1.0 (see Figure 2.21).



◀FIGURE 2.21
Selecting an older, labeled version of a project in the History of Project dialog box.

- Click the Share button to display the Share From dialog box.
- Make sure that the Visual SourceSafe root is selected in the Share From dialog box's Projects tree (see Figure 2.22), and click the OK button to display the Share dialog box.
- In the Share dialog box, give the new project an appropriate name, such as "Version 1.1" (see Figure 2.23), and click OK on the Share dialog box and Close on the History dialog box.

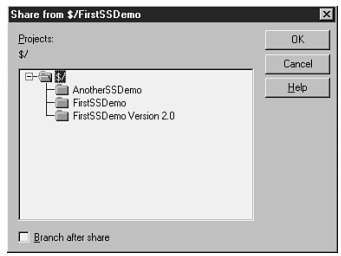
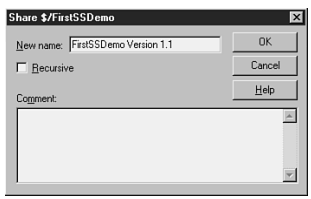


FIGURE 2.22▲
Selecting the appropriate parent project in the Share From dialog box (usually, the root).



◀FIGURE 2.23
Naming the new maintenance project in the Share From dialog box.

2. Open the new project's Visual SourceSafe folder and note that all the project's files are automatically pinned (they have a special pin icon). Visual SourceSafe automatically pinned the files because the files that you shared were not from the latest version (see Figure 2.24).

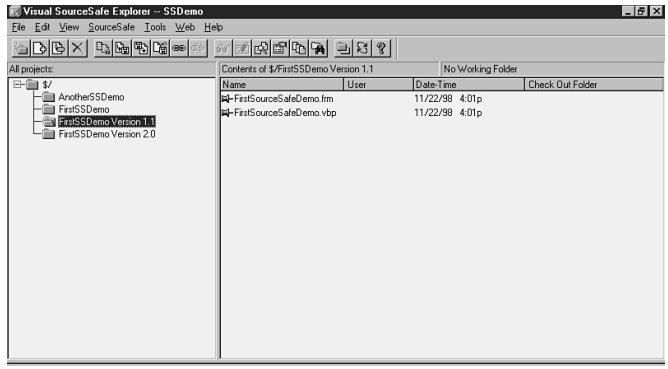


FIGURE 2.24▶
Pinned files in a SourceSafe project.

3. Branch the files for module C (the files that will need to be changed).
   - Select the file or files that you will need to modify for module C.
   - Choose SourceSafe, Branch from the Visual SourceSafe menu.
   - In the Branch dialog box, write any appropriate comments, and then click the OK button to complete the branching operation (see Figure 2.25).
   - The branched file or files are no longer pinned, but the files that you did not select for branching remain pinned (see Figure 2.26).

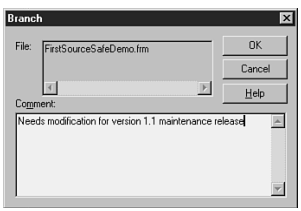


FIGURE 2.25▲
The Branch dialog box.

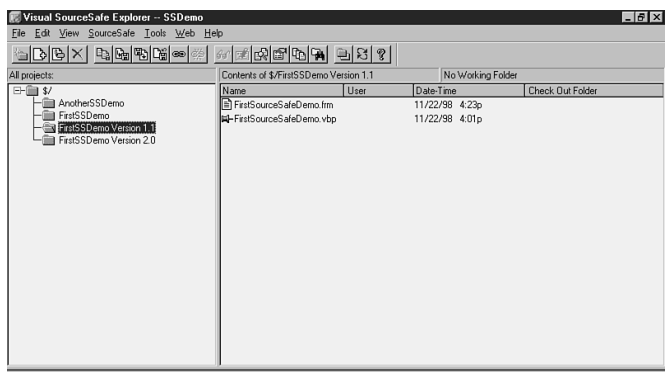


FIGURE 2.26▶
A project that shows a branched file (no longer pinned).

At the end of the preceding operation, you will have a new project that contains pinned, shared files from the original project and branched files from the original project.

Because the pinned files are still shared, they are the same as the files in the original project. Because they're pinned, a programmer working on the new project can't check them out to change them.

The branched files are no longer shared, nor are they pinned. A programmer working on the new project can therefore check them out and change them, without reflecting the changes back to the original project.

Merging Two Different Versions of a File

Occasionally, you will need to reconcile two disparate versions of a file in Visual SourceSafe. There are two scenarios where such a need could occur:

- A project was branched into two projects, as described in the section titled "Source-Code Branching." The two projects follow independent development paths from that moment on. Somewhere down the line, however, developers discover the need for a change (a bug fix or enhancement) that is desirable for both projects. Developers make the change in one of the branches and then merge the change to the other branch.
- Multiple developers have checked out the same file at the same time and now want to check in their separate changes.

NOTE

Multiple Concurrent Checkouts Enabled by Administrator Multiple developers can check out the same file at the same time if a Visual SourceSafe administrator checks the Allow Multiple Checkouts option on the General tab of the SourceSafe Options dialog box. An administrator can invoke this dialog box in Visual SourceSafe Administrator by choosing Tools, Options from the Visual SourceSafe Administrator menu.

Installing and Configuring VB for Developing Desktop and Distributed Applications

- Install and configure Visual Basic for developing desktop/distributed applications.

Practically speaking, this exam objective is the same for both exams. Questions based on these exam objectives require you to know the differences in the features provided with the various editions of VB6 and Visual Studio 6.0.

The following list of features provides the information necessary to answer these questions:

- Visual Basic 6.0 Learning Edition
  - Intrinsic controls
  - Tab control
  - Grid control
  - Data-bound controls
- Visual Basic 6.0 Professional Edition
  - All features included with the Learning Edition
  - Additional ActiveX controls
  - IIS Application Designer
  - VB Database Tools and Data Environment
  - ADO (ActiveX Data Objects)
  - DHTML Page designer
- Visual Basic 6.0 Enterprise Edition
  - All features included with Learning and Professional Editions
  - SQL Server
  - Microsoft Transaction Server
  - Internet Information Server
  - Visual SourceSafe
  - SNA Server
  - Other BackOffice tools

CHAPTER SUMMARY

This chapter covered the following topics:

- Using Visual SourceSafe Administrator to set up SourceSafe user accounts
- Using Visual SourceSafe Administrator to archive and restore projects in SourceSafe databases
- Using Visual SourceSafe Explorer to check projects in and out of a Visual SourceSafe database
- Using Visual SourceSafe Explorer to share, pin, branch, and merge information in development projects
- The different features of the editions of VB that developers can use to create desktop and distributed applications

KEY TERMS

- Branch
- Difference
- Merge
- Pin
- Share
- Version control
- Visual SourceSafe
- Working Folder

APPLY YOUR KNOWLEDGE

Exercises

2.1 Using SourceSafe Administrator

In this exercise, you use Visual SourceSafe Administrator to administer Visual SourceSafe users.

Estimated Time: 20 minutes

> NOTE
>
> **You Need Access to Visual SourceSafe Software Tools for These Exercises** Exercises 2.1 through 2.4 require that you install SourceSafe Administrator and SourceSafe Explorer, which come as part of the install package with VB or Visual Studio. You will need to know the Admin password for SourceSafe Administrator. You will also need to set up a SourceSafe user account that you can use in Exercises 2.2 and 2.3. Exercise 2.1 shows you how to set up a SourceSafe user in SourceSafe Administrator.

1. Open Visual SourceSafe Administrator.
2. If this is the first time that anyone has entered Visual SourceSafe Administrator, there will be no logon screen, because there will be no Admin password. If someone has already set up an Admin password, you will need to know it to be able to log on.
3. To begin adding a new user, choose Users, Add User on the menu. This brings up the Add User dialog box.
4. Fill in the Username field and the Password field. Leave the Read-only box unchecked. If you want to have an automatic logon to Visual SourceSafe Explorer from your workstation, the username and password should be identical to the username and password that you use to log on to your Windows system.
5. Click the OK button to add the user.
6. Add another user, as in steps 3–5. Experiment with changing the properties for this user with the Users, Edit User and Users, and Change Password menu choices. Finally, delete the user by choosing Users, Delete User.
7. Examine some of the other administrative options that are available by choosing Tools, Options from the menu and examining the various tabs on the SourceSafe Options dialog box.
8. Note the two check box options on the General tab: Allow Multiple Checkouts, and Use Network Name for Automatic User Log In. Changing these options will change the behavior of SourceSafe Explorer for developers.
9. Note the choices on the Project Security tab. If you check the Enable Project Security option, the other options will become available. These options will determine which actions new users can perform on projects in SourceSafe Explorer.

2.2 Using SourceSafe from the VB Environment

In this exercise, you use features of SourceSafe that are integrated into the VB IDE. Before doing this exercise, you need to be sure that you have an available SourceSafe user account, as discussed in Exercise 2.1.

Estimated Time: 20 minutes

APPLY YOUR KNOWLEDGE

1. In VB, begin a new standard EXE project. On the General tab of the Project, Options dialog box, change the project name to `FirstSourceSafeDemo`.
2. Save the project to a location on your local hard drive, naming the default form and the project file `FirstSourceSafeDemo.frm` and `FirstSourceSafeDemo.vbp`, respectively.
3. If SourceSafe has been installed on your system, you receive a prompt to add the project to SourceSafe. Answer Yes to the prompt.
4. If you receive a SourceSafe Login screen, enter the username and password that you created for yourself in Exercise 2.1.

   Note that you may need to select a VSS database the first time that you log in. Make sure it is the same database that you used in the preceding exercise.
5. The Add to SourceSafe Project dialog box will appear. The default project name will be the name that you gave to the project in step 1. You can leave the entry as is, type in a different project name, or choose an existing SourceSafe project. For this exercise, leave the entry as the VB project name and click the OK button.
6. SourceSafe will prompt you that the project does not exist in SourceSafe. Click Yes to create the SourceSafe project.
7. The Add Files to SourceSafe dialog box appears. Note that the files in your VB project are listed, and all are checked. Leave them checked, and click the OK button.
8. The `FirstSourceSafeDemo` project has now been added to SourceSafe, and its files have been checked in. Look at the project and the form in Project Explorer and note the small padlock icon to the left of each object. This indicates that the file is read-only (because it has been checked in to SourceSafe).
9. Right-click the form in Project Explorer to bring up the form's shortcut menu. The last four options on the menu are SourceSafe options. Select the Check Out option. Note that the icon next to the form in Project Explorer now has a red check mark in place of the padlock that you saw in the preceding step.
10. Do something to change the form, such as adding a command button.
11. Right-click the form again, and this time choose Check In from the shortcut menu. Note the Check In dialog box that appears. You can type comments for this check-in, and you can also elect to keep the form checked out. This will have the effect of momentarily checking in your file, just long enough to refresh the SourceSafe database with any changes. The file will then remain checked out to you so that you can continue working with it.
12. Choose Tools, SourceSafe from the VB menu and examine the resulting submenu.

2.3 Using SourceSafe Explorer

In this exercise, you use Visual SourceSafe Explorer to manipulate a sample project.

Estimated Time: 45 minutes

1. In VB, begin a new standard EXE project and save its form and VBP files with the names `SecondSourceSafeDemo.frm` and `SecondSourceSafe.vbp`, respectively. Do *not* add the project to SourceSafe from the VB environment. Exit VB.

APPLY YOUR KNOWLEDGE

2. Add a project to the Visual SourceSafe database with the following steps:
   - Select the Root folder (/$).
   - Choose File, Create Project from the main menu.
   - Type the name of the new project, `SecondSourceSafeDemo`, in the Create Project dialog box, and press the Enter key.
   - The folder for the new project should now appear on the SourceSafe Projects tree.
3. Add files to the project with the following steps:
   - Select the project folder for SecondSourceSafeDemo.
   - Choose File, Add Files from the main menu.
   - On the Add File dialog box, browse to the location of the files that you saved in the first step of this exercise.
   - Hold down the Ctrl key (for multiple selection), and select both files from the VB project (`SecondSourceSafeDemo.vbp` and `SecondSourceSafeDemo.frm`). Then click the Add button.
   - In the resulting Add Files dialog box, type the comment `Initial check-in` and click the OK button. If you receive a prompt to create a Working Folder for your project, click No. You will assign a Working Folder later in this exercise.
   - Click Close to end the Add File dialog box.
   - The files you added to the project will now appear in the right-hand pane of the Visual SourceSafe Explorer.
4. Create a new Working Folder for the project by following these steps:
   - Select the project's folder.
   - Choose File, Set Working Folder from the menu.
   - Browse to a different folder and choose that folder as the project's new Working Folder.
5. Check out both files in the project with the following steps:
   - Select both files (`SecondSourceSafeDemo.frm` and `SecondSourceSafeDemo.vbp`) by selecting them in the right-hand pane.
   - Choose SourceSafe, Check Out from the menu.
   - In the resulting Check Out dialog box, type the comment `Checked out to add CommandButton to form` and click the OK button.
6. In VB, open the project in the *new location* where you checked it out and add a command button to the form. Save the project and close VB.
7. In Visual SourceSafe Explorer, check the files back in with the following steps:
   - Select the project's folder.
   - Select both files.
   - Choose SourceSafe, Check In from the menu.
   - In the resulting dialog box, type the comment `Added CommandButton to form` and click the OK button.

APPLY YOUR KNOWLEDGE

8. Create your own label to define a version with the following steps:
   - Right-click the project's folder in the Projects tree.
   - In the resulting shortcut menu, choose Label.
   - In the Label dialog box, type `Version 1.0` as the label, and click OK.
9. Share the files to another project by following these steps:
   - Create another project named SourceSafeDemo 2.0 under the root ($/) and select the new project's folder.
   - Choose SourceSafe, Share from the menu.
   - In the resulting Share With dialog box, navigate to the SecondSourceSafeDemo project and choose its files.
   - Click the Share button, and then click Close.
   - You should now see the files in the second project's folder. Note that the document icon next to each file appears doubled.
10. Branch the files by following these steps:
   - Make sure that the SecondSourceSafeDemo 2.0 project is selected.
   - Select its files in the right-hand pane.
   - On the SourceSafe menu, click Branch.
   - In the resulting dialog box, enter a comment such as `Version 2.0 branched from version 1.0` and click OK.
   - Notice that the icon for each file changes from the Share icon to the standard Document icon.

2.4 Archiving and Restoring a Project in SourceSafe Administrator

In this exercise, you archive and restore a project in SourceSafe Administrator

Estimated Time: 20 minutes

1. Open Visual SourceSafe Administrator.
2. Choose Archive, Archive Projects from the menu.
3. On the Choose Project to Archive screen, choose the `FirstSourceSafeDemo` project that you worked with in Exercises 2.2.
4. Review your choice on Step 1 screen of the Archive wizard. Click the Next button.
5. On the Step 2 screen of the Archive wizard, choose the option Save data to file, then delete from the database to save space.
6. Still on the Step 2 screen, click Browse and use the Browse screen to choose a location and enter a name for the archive file. (The .SAA extension will be supplied by default, so you don't have to specify it.) Click the Next button to proceed to the Step 3 screen of the Archive wizard.
7. Review the information on the Step 3 screen, and click the Finish button. Wait until Archive wizard gives you a message indicating success.
8. Now, you will restore the project that you just archived.
9. Choose Archive, Restore from the menu to bring up the Step 1 screen of the Archive wizard.
10. Click the Browse button to locate and select the archive file that you created in the previous steps. Then click the Next button to proceed to the Step 2 screen.

APPLY YOUR KNOWLEDGE

11. Make sure the archived project is selected on the screen for Step 2 and click the Next button.
12. On the Step 3 screen, make sure that the option labeled `Restore to the project the item was archived from` is selected. Click the Finish button. Wait until Archive wizard gives you a message indicating success.

2.5 Investigating VB Features

In this exercise, you examine the features of VB that enable you to develop distributed and desktop applications.

Estimated Time: 20 minutes

1. Review the final section of this chapter, "Installing and Configuring VB for Developing Desktop and Distributed Applications."
2. Install VB6 on your system (preferably Enterprise Edition), and observe the various options.

Review Questions

1. What is pinning in Visual SourceSafe, and why is it useful?
2. How can a developer get a copy of a file stored in Visual SourceSafe so that the developer can make changes to the file?
3. Which versions of VB come with SQL Server bundled in? Visual SourceSafe?

Exam Questions

1. You have an application whose production version is 1.0. You are currently developing version 2.0, but there is a serious bug in version 1.0 that needs fixing. This will require you to make a "service pack" distribution of 1.0 called 1.1.

   You would like to continue working on version 2.0 while version 1.1 is being prepared. Version 2.0 needs to use some of the same files as 1.1. You would also like to keep a copy of the original source code for version 1.0, including the files that had the serious bug.

   Version 2.0 is currently not a separate SourceSafe project, but is just a set of changes over version 1.0, whose files are labeled in the project.

   How can you use Visual SourceSafe to manage this situation?

   A. Make separate physical copies of all the files in version 2.0 and a new set of copies of the files of version 1.0. Create a new project for version 1.1, and check in the new copies of all the version 1.0 files. Check in all the 2.0 copies to the new 2.0 project.

   B. Create a separate new project for version 1.1. Share all the files from the version labeled 1.0. As programmers make fixes to 1.1, they can check them in. When they are finished, they can label these files as version 1.1.

   C. Check out and remove the 2.0 file and check them into a new, separate project. Create a new 1.1 project for the fix. Programmers working on fixing the problematic module can get a read-only copy of the good files from 1.0 and can check out the files for the bad module as they work on them.

APPLY YOUR KNOWLEDGE

D. Share the files from the project to a new project, 1.1, which will automatically pin the files. In the 1.1 project, branch the "bad" files that need to be fixed.

2. A project has multiple developers working on different parts of the project. At the end of each workday, you want there to be an updated, running version of the project available from SourceSafe to testers. You can achieve such a goal by:

   A. Asking developers to make sure that their code works with the latest code in rest of the project, and then checking in their changes at the end of the day.

   B. Asking developers to keep their source code checked out at all times, but make working copies available in a second Visual SourceSafe project or Visual SourceSafe database.

   C. Asking developers to check their source code in at the end of each day. It is then the project manager or testers' responsibility to determine the latest version of each file that runs with the entire project.

   D. Asking developers to coordinate source code by email or face-to-face meeting at the end of each day, label the latest version of all working source code, and then check in.

3. You want to use a VB project as the basis for another project (not a new version of the first project). If improvements are made to the files during development of either project, you want both projects to share the benefits. The original project is in Visual SourceSafe. The best way to manage this situation with Visual SourceSafe is:

   A. Share the first project, and then define the second project to include the same files.

   B. Pin and branch the first project. One of the branches will become the second project.

   C. Check out the files from the first project and check them in to the second project.

   D. Make separate physical copies of the files in the first project and check the new copies in to the second project.

4. You want to release a new version of your application, but you want to be able to recall all the source code for the old version, even after the new version has been released. What should you do?

   A. Label the new version.

   B. Share the old project under a distinct new name.

   C. Label the original version.

   D. Make separate copies of all the files used in the first version.

5. To copy information from one Visual SourceSafe database to another Visual SourceSafe database, you can:

   A. Copy selected contents of the Data folder of the source database to the Data folder of the target database.

   B. Check out selected contents of the source database and check them in to the target database.

   C. Archive selected contents of the source database and restore the archive into the target database.

   D. Delete selected contents of the source database and add them to the target database.

APPLY YOUR KNOWLEDGE

6. The lowest-end product that provides you with all the tools necessary to develop a solution that uses a SQL Server database is:

   A. VB 6.0 Learning Edition

   B. VB 6.0 Professional Edition

   C. VB 6.0 Enterprise Edition

   D. Visual Studio 6.0

7. The lowest-end product that provides you with all the tools necessary to develop a solution that uses Internet Information Server is:

   A. VB 6.0 Learning Edition

   B. VB 6.0 Professional Edition

   C. VB 6.0 Enterprise Edition

   D. Visual Studio 6.0

Answers to Review Questions

1. Pinning enables you to use an earlier version of a file (not the current version) in a VSS project. It is good for creating maintenance releases of a project while new development on the same project is going on at the same time. See "Pinning an Earlier Version of a File for Use in the Current Version of a Project."

2. A developer can check a file out of Visual SourceSafe to get a modifiable copy. See "Checking Out, Checking In, and Getting Working Copy."

3. Only the Enterprise Edition of VB6 actually includes SQL Server, although it is possible to access SQL Server data from the other editions. See "Installing and Configuring VB for Developing Desktop and Distributed Applications."

Answers to Exam Questions

1. **D.** You can use the share, pin, and branch model to create a maintenance or "service pack" release on the production version of an application that already is under development for a new major version. Answer C contains no provision for using the earlier versions of the files. Answers A and D mention version 2.0, which really doesn't need to be changed for this purpose. For more information, see the section titled "Pinning an Earlier Version of a File for Use in the Current Version of a Project."

2. **A.** Developers can check working code in at the end of each day to make sure that the project always has the latest working copies of source code. Answer B would be unnecessarily cumbersome and would replicate the project in two places. C places unnecessary responsibility on a manual procedure (checking to see whether projects compile and run). D also puts unnecessary reliance on manual procedures and unrealistically relies on perfect group coordination. For more information, see the section titled "Checking Out, Checking In, and Getting Working Copy."

3. **A.** You can share a project's files to make it the basis for another project that needs to share changes with the first project. Answer B ignores the requirement of sharing common changes, because branching cuts off the link between changes in the projects. Answer C is meaningless, and answer D also would destroy the links between the files in the two projects. With the correct answer (answer A), if there is a need in the future for the projects to diverge their copies of these files, you could branch the second project at that time. For more information, see the section titled "Source-Code Sharing."

APPLY YOUR KNOWLEDGE

4. **C.** To keep track of all of a project's older version's source code as a group, you can label the older version. For more information, see the section titled "Source-Code Labeling and Version Numbers."

5. **C.** To move information from one SourceSafe database, you can archive it from the first database and restore it to the second database. Although answer B might work, it would be clumsy. Answer D doesn't achieve the objective of making a copy because it deletes the files from the source database. Answer A is incorrect, because you should never attempt to directly manipulate the contents of the VSS database. For more information, see the section titled "Archiving and Restoring Visual SourceSafe Databases."

6. **C.** The lowest-end product that provides you with all the tools necessary to develop a solution using SQL Server is the Enterprise Edition. Although the other editions enable you to connect to SQL Server data, they don't actually come with a copy of SQL Server. Therefore, unless you already had access to SQL Server from your computing environment, you wouldn't have the necessary tools to do the development. For more information, see the section titled "Installing and Configuring VB for Developing Desktop and Distributed Applications."

7. **B.** The lowest-end product that provides you with all the tools necessary to develop a solution using IIS (known as WebClass or IIS applications in VB) is the Professional Edition. The Learning Edition doesn't have any tools for IIS/WebClass application development. Both the Professional Edition and Enterprise Edition give you the IIS Application Designer. Remember that you must already have access to IIS from your computing environment (an installation of Windows NT 4.0 with Option Pack 3.0); otherwise you won't be able to develop IIS applications, regardless of the VB edition. For more information, see the section titled "Installing and Configuring VB for Developing Desktop and Distributed Applications."

OBJECTIVE

This chapter helps you prepare for the exam by covering the following objective:

Implement navigational design (70-175 and 70-176).

- **Dynamically modify the appearance of a menu**
- **Add a pop-up menu to an application**
- **Create an application that adds and deletes menus at runtime**
- **Add controls to forms**
- **Set properties for `CommandButtons`, `TextBoxes`, and `Labels`**
- **Assign code to a control to respond to an event**

▶ The exam objective addressed in this chapter covers many of the basic elements of user interface programming in Visual Basic. Specifically, this objective addresses two main topics:

▶ How to create and control the behavior of standard Windows menus in a VB application

▶ How to set up and control the behavior and appearance of the three most basic controls (`CommandButton`, `TextBox`, and `Label`) in a VB application

CHAPTER 3

Implementing Navigational Design

OUTLINE

Understanding Menu Basics **81**

Knowing Menu Terminology 81
Using the Menu Editor 82
Attaching Code to a Menu Item's `Click` Event Procedure 83

Dynamically Modifying the Appearance of a Menu **84**

Adding a Pop-Up Menu to an Application **85**

Defining the Pop-Up Menu 85
Determining the Mouse Button 86
Displaying the Pop-Up Menu 88
Controls with Pop-Up Menus 88

Creating an Application That Adds and Deletes Menus at Runtime **89**

Creating Runtime Menu Items 89
Code for Runtime Menu Items 90
Removing Runtime Menu Items 91

Adding Controls to Forms **91**

Setting Properties for `CommandButtons`, `TextBoxes`, and `Labels` **92**

Referring to a Property Within Code 94
Important Common Properties of `CommandButtons`, `TextBoxes`, and `Labels` 95
Important Properties of the `CommandButton` Control 99
Important Properties of the `TextBox` Control 100
Important Properties of the `Label` Control 102

Assigning Code to a Control to Respond to an Event **103**

Changing a Control Name After You Assign Code to the Event Procedure 104
The `Click` Event 105
The `DblClick` Event 106
`MouseUp` and `MouseDown` 106
Mouse Events Compared With `Click` and `DblClick` 108
`MouseMove` 109
The `Change` Event 109
Other Events Commonly Used for Input Validation 110

Chapter Summary **110**

Study Strategies

- Experiment with basic menu setup using the Menu Editor (see Exercise 3.1).
- Experiment with dynamic runtime changes to menu properties (see Exercise 3.2).
- Experiment with pop-up menus (see Exercise 3.3).
- Experiment with menu items in a control array (see Exercise see 3.4).
- Experiment with programming the events and properties of `CommandButtons`, `TextBoxes`, and `Labels` (see Exercises 3.5 and 3.6).
- Memorize the relative timing of `MouseUp`, `MouseDown`, `Click`, and `DblClick` events (see the section of this chapter, “Assigning Code to a Control to Respond to an Event”).

INTRODUCTION

To increase the functionality of a Visual Basic application, menu bars provide the user with a simple way of controlling the program. The menu bar is at the top of a form window in most applications. Visual Basic provides a Menu Editor that simplifies the creation of menus. Once created these menus can be individually programmed to respond when selected.

Another type of menu quite popular with users is the pop-up menu. This menu can be very specific to certain controls or areas of the application. These menus are often called context-sensitive menus.

To provide the user with customized menu options, menus can also be created at runtime. The Most Recently Used File list is a good example of how dynamic menus customize an application based on the user's needs. These menus provide runtime assistance that can vary depending on a user's preferences.

This chapter covers the following topics:

- Menu basics
- Menu terminology
- Using the Menu Editor
- Dynamically modifying the appearance of a menu
- Changing the menu's properties at runtime
- Adding a pop-up menu to an application
- Defining the pop-up menu
- Determining the mouse button
- Displaying the pop-up menu
- Controls with pop-up menus
- Creating an application that adds and deletes menus at runtime
- Creating runtime menu items
- Coding for runtime menu items
- Removing runtime menu items
- Adding controls to forms

- Setting properties for `CommandButtons`, `TextBoxes`, and `Labels`
- Assigning code to a control to respond to an event

UNDERSTANDING MENU BASICS

Applications use *menu bars* to provide an organized collection of commands that can be performed by the user. Menus can inform the user of the application's capabilities as well as its limitations. If a program has properly organized menus, users can easily find common commands as well as less-familiar features. Users can also learn shortcut keys from a well-designed menu structure.

Because menus have so many benefits to the user, a programmer should be well-versed in their creation and the functions they can provide.

Programmers can use one of two different methods to create menus in VB. The first is a built-in Menu Editor dialog box. This editor provides a fast, simple way to generate menus. Once made, all menu objects can have their properties set through program code.

The other method uses the Win32 Application Programmers Interface. The Win32 API is a series of functions that are by VB and provided by the operating system. By using specialized function calls, a programmer can create menus, set properties of menus, and modify menu structure. Most of the functionality found in the API is also available using the Menu Editor. Although more difficult to work with, the API provides enhanced function calls and special capabilities not part of the VB Menu Editor. However, using the API is beyond the scope of this chapter.

Knowing Menu Terminology

Knowing how menus operate is an important part of the design process. It is also important to be familiar with the standard terminology for Windows menu interfaces. Terms such as top-level menu, sub-menu, and pop-up menu all describe how the menu should behave as well as where the user can expect to see the menu, as shown in Figure 3.1 and 3.2.

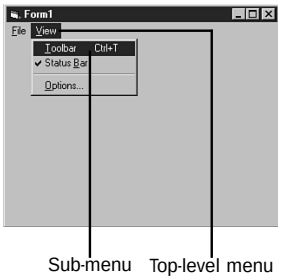


FIGURE 3.1▲
Menu terminology and hierarchy.

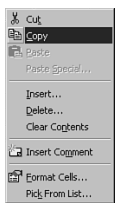


FIGURE 3.2▲
Pop-up menu for Microsoft Excel.

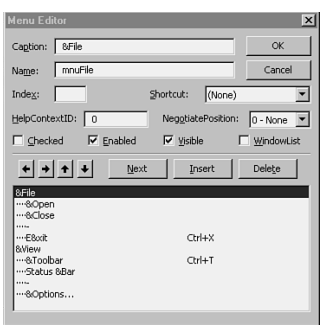


FIGURE 3.3▲
The Visual Basic Menu Editor.

Top-level menus are the items seen on the main part of the menu bar, directly under the title bar of the application window. Standard user interface guidelines state that all top-level menus should have at least one sub-menu.

The sub-menu appears when a top-level menu has been opened. The sub-menu implements commands that logically belong to the top-level menu. One example is the File menu, found in most applications. This is a top-level menu. Once opened, commands such as New, Open, Save, and Close all relate to actions that affect the file.

Using the Menu Editor

Standard Windows menus are always located at the top of a form, just under the title bar. VB programmers can create menus by first selecting the form that will host the menu and then using the VB Menu Editor, as shown in Figure 3.3.

The Menu Editor is available only when a form is being designed. It is located on the Tools menu in VB6.

The first step in creating a menu is to enter the menu item's caption. The caption can incorporate the ampersand (&) for an access key designation, also known as an accelerator key. This enables the user to see an underlined letter in the menu and use the Alt key along with the accelerator key.

After the caption of the menu item has been set, the menu requires an object name for programmatic reference. The menu names can be any valid object name. Naming conventions are preferred to allow for easier reading of source code and quick identification of objects. Menus use the three-letter prefix "mnu" before the selected name.

The remaining task in creating a menu is to decide at which level in the menu structure this item will appear. Will the object be a top-level menu item, sub-menu item, or a sub–sub-menu item? In the list box at the bottom of the Menu Editor, all top-level menu items are flush with the left side of the list box border. Sub-level menu items have four small dots preceding the menu caption; sub–sub-menu items have eight dots. The menu items can also be reorganized according to position in the menu.

To control the level and position of the menu item being entered, just use the four direction arrows in the Menu Editor. The up and down arrows reposition menu items for order. The left and right arrows allow menu items to be top-level, sub-level or sub–sub-level.

You can implement a separator bar in a menu (a horizontal line that visually defines the boundary between two groups of menu items) by putting a single dash (-) as the entire caption for a menu item. This item then becomes the separator bar on the runtime menu. You must remember that even separator items in a menu must each have their own unique `Name` property.

Menu items can also have their appearance properties set at design time through the Menu Editor. Properties such as `Checked`, `Enabled`, `Visible`, `WindowList`, and a shortcut key can all be specified. In Figure 3.3, the Menu Editor check boxes for the `Checked`, `Enabled`, `Visible`, and `WindowList` properties are all visible.

In addition a drop-down list of possible shortcut key combinations allows the programmer to assign a shortcut key to this particular menu item. Unlike the accelerator or access key mentioned above, the shortcut key can be pressed by the user without its corresponding menu item being visible. Windows automatically displays the shortcut key assignment when the menu item is displayed.

All of these properties are also available at runtime except for the shortcut key. To change the desired property of the menu object, just use the object name followed by the property name and the value. Examples of such runtime changes are given in the following section.

Attaching Code to a Menu Item's `Click` Event Procedure

A menu control has a single event procedure: the `Click` event procedure. The `Click` event procedure is the place where you write or call the code that you want to execute when the user chooses the menu item.

You can access a menu control's `Click` event procedure by single-clicking the menu item from the design time copy of the menu.

Dynamically Modifying the Appearance of a Menu

When a menu system is created at design time, the programmer can control property settings for menu items. After these properties are set, they can be altered at runtime to assist the user in interpreting which selections have been made, which items are turned on, and which commands are available.

To dynamically alter the appearance of a menu system, the menu object is referenced along with the desired property to be altered. This syntax is the same for regular controls found on a VB form. The following sections explore the syntax used in setting menu properties.

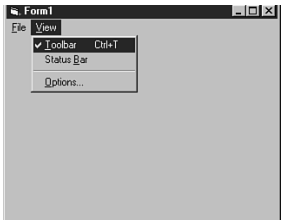


FIGURE 3.4
Sample menu with properties that change at runtime.

The following code assumes that `Form1` has a complete menu system already defined. The View menu has a sub-menu item called Toolbar. This menu item is having the `Checked` property set to true to indicate the toolbar is visible (see the result in Figure 3.4):

```
mnuViewToolbar.Checked = True
```

This code uses the same syntax that other objects use: the object name, a period separator, followed by the property to be set, and then the value after the assignment operator.

The following code demonstrates more menu objects and their property settings:

```
mnuViewStatusBar.Checked = True
mnuFileOpen.Enabled = False
mnuFormatFontBold.Checked = True
mnuPopUp.Visible = False
```

In these examples, notice how the use of the menu naming convention helps decipher which menu is to be affected. The object name starts with the mnu prefix followed by the top-level menu item named `View`, followed by the sub-level menu item, `StatusBar`. This ensures easy readability when going through source code.

Altering application menus at design time is not very different from controlling other objects. The one difficulty that programmers have with menus is remembering to control both the user interface and the menus. The interface can always be seen, but menus must first be opened to view their states. One common technique used to assist with maintaining a consistent interface is to call a procedure.

The procedure will perform the required actions and affect both the interface and required information. This allows various menus to call the same code and keep the program flow easier to follow.

Adding a Pop-Up Menu to an Application

Pop-up menus, also known as "context menus" or "right-mouse menus," provide the user another convenient way to control the application. Pop-up menus are usually unique for various areas of an application. They can be created in Visual Basic or the operating system, and certain objects provide them. Microsoft Excel provides an example of pop-up menus in Figure 3.2. With Excel, the user has the option to use the main menu bar, shortcut keys, or pop-up menus. Different pop-up menus are found, depending on the object selected. If the user selects a set of cells in the spreadsheet, he gets a specific pop-up menu. If the user right-clicks on a worksheet tab, he gets a different pop-up menu.

To provide pop-up menus for different parts of the application, various menus must be used. These menus, created as part of the main menu system, do not have to be visible. When they are needed, they are called through program code.

Defining the Pop-Up Menu

To create a pop-up menu, a top-level menu item is used. This item can be part of the normal menu structure seen by the user, or it can have its `Visible` property set to `False` so that it does not appear with the regular menu(as shown in Figure 3.5).

All items on the pop-up menu are created as sub-level items of the invisible top-level menu. Each sub-menu item can have its properties set as required.

When the pop-up menu is needed, the `PopupMenu` method is called to activate the pop-up menu. The following code demonstrates this:

```
Form1.PopupMenu mnuPopUp1
```

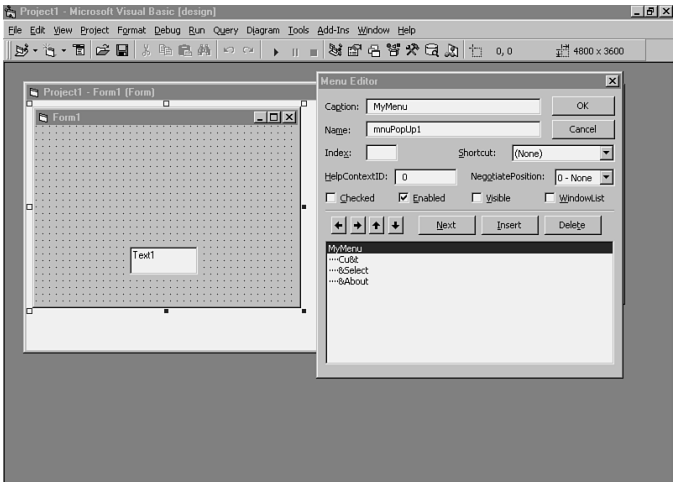


FIGURE 3.5▶
design time view of menu system with a pop-up menu.

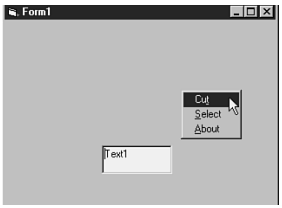


FIGURE 3.6▲
Runtime view of a custom pop-up menu.

This preceding code calls the `PopupMenu` method of `Form1`, as shown in Figure 3.6. The menu item `mnuPopUp1` is passed as an argument and is displayed at the current mouse location. This code assumes that a menu system has been created and a menu item named `mnuPopUp1` exists.

The `PopupMenu` method accepts the name of the menu to activate, arguments that indicate the orientation of the menu (left, center, and right), and the screen coordinates for display and other options.

Determining the Mouse Button

After the desired menus have been created, the next step for the programmer is to decide which objects will call which menus. The form, as well as individual controls on the form, can all have pop-up menus specified.

The standard method used to activate an object's pop-up menu is to use the `MouseUp` event procedure to detect when the user right-clicks the mouse. After the user right-clicks, the menu is displayed.

To determine the state of the mouse, the programmer traps the `MouseUp` event of the object to provide the menu. Using the event's `Button` parameter, the programmer verifies which mouse button the user pressed.

To determine which mouse button was activated, the procedure is passed two arguments: `Button` and `Shift`. The `Button` argument indicates the mouse button that was pressed. The `Shift` indicates whether the Ctrl and/or Alt and/or Shift was active when the mouse click event occurred.

The following code in the form's `MouseUp` event determines which mouse button was pressed and then further distinguishes which Shift key or Shift key combination was pressed along with the mouse button.

```
Sub Form_MouseUp(Button As Integer, Shift As Integer, X
➥As Single, Y As Single)
    Dim blnIsAlt as Boolean
    Dim blnIsCtrl as Boolean
    Dim blnIsShift as Boolean
    BlnIsAlt = Shift And vbAltMask
    BlnIsCtrl = Shift And vbCtrlMask
    BlnIsShift = Shift And vbShiftMask
    If Button = vbLeftButton Then
        If blnIsAlt And blnIsShift Then
            … So something to react to Left Button +
➥Alt + Shift
        End If
    ElseIf Button = vbRightButton Then
        Form1.PopupMenu mnuPopUp1
    End If

End Sub
```

The preceding code uses VB constants to determine the button-Shift key combination. The key combination of Ctrl and/or Alt and/or Shift will also be returned as a number in the Shift parameter. You can test to see whether a particular key was pressed by the user by using the AND operator to compare the Shift parameter with one of the bit mask constants `vbAltKey`, `vbShiftKey`, or `vbCtrlKey`.

In the `MouseUp` event, both `Button` and `Shift` are integer values. When programming for this event, either the integer value can be used or the VB constants that refer to the various possible `Button` and `Shift` key values.

For further discussion of how to program with the `MouseUp` and `MouseDown` event procedures, see the section in this chapter entitled "`MouseUp` and `MouseDown`."

Displaying the Pop-Up Menu

When using pop-up menus with forms or controls, you first define the menu, test for the right mouse button, and then display the desired menu.

The following source code puts together these tasks by using the `PopupMenu` method of the form at the same time that the mouse button has been pressed.

```
Sub Form_MouseUp(Button As Integer, Shift As Integer, X As
➥Single, Y As Single)
    If Button = vbRightButton Then
        Form1.PopupMenu mnuPopup1
    End If
End Sub
```

In this code sample, a simple `If` statement provides the check to see whether the right mouse button has been selected. If the `Button` argument equals the `vbRightButton` constant, the form `PopupMenu` method is called and passed the name of the top-level menu object to be displayed. Remember that the top-level item will not be shown, only sub-level items will be.

Any menu can be passed to the `PopupMenu` method. In the preceding code, a special menu named `mnuPopup1` was created and its visible property was set to `False`. This hides the top-level menu from the application menu bar but allows the menu to be called. The `PopupMenu` method will display sub-level menu items regardless of the top-level menu item's visible property.

The `PopupMenu` method can display only one menu at a time. A second call to the `PopupMenu` will be ignored.

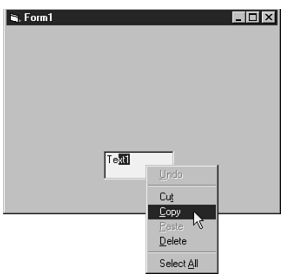


FIGURE 3.7
The built-in pop-up menu of the `TextBox` control.

Controls With Pop-Up Menus

Not all controls require a pop-up menu to be created. Certain controls that ship with Visual Basic, such as the `TextBox` control, already have a pop-up menu built into them, as shown in Figure 3.7. Visual Basic has no method to disable such built-in, pop-up menus.

If the `PopupMenu` method is used with a control that has its own built-in, pop-up menu, the built-in menu will always be displayed first. The custom menu will only be displayed subsequently. Displaying two separate menus is not usually the desired effect for the user interface.

The benefit of built-in menus is that they do not require program code for their functionality. The control provides all mouse handling and functions.

Creating an Application That Adds and Deletes Menus at Runtime

Runtime menus are created dynamically by the application, as they are required. The runtime menu may list user-selected information, recently used files, or a list of Internet Web sites, to name three examples.

To create runtime menus dynamically, you must use a menu control array. At runtime the `Load` statement creates the new menu items, and the `Unload` statement removes them.

The following sections detail how runtime menus are created and removed from the menu system.

Creating Runtime Menu Items

The three steps to creating a runtime menu item are as follows:

STEP BY STEP

3.1 Creating a Runtime Menu Item

1. Create a design time menu that will act as the template, as shown in Figure 3.8. Set the `Index` property of the template item to create a menu control array. This array will allow new elements to be created at runtime. These elements can then be controlled like a regular menu item.
2. Use the `Load` statement at runtime. The `Load` statement accepts the name of the object to be loaded. The following is sample syntax:

```
Load mnuFileItem(1)
```

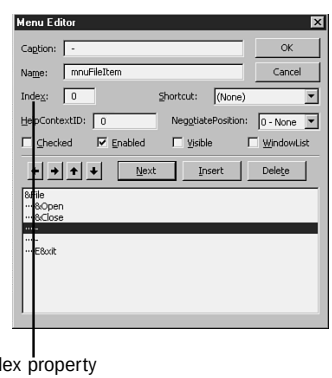


FIGURE 3.8
Creating a template menu item with an index value.

The name of the menu item is based on the design time item that had the `Index` property set. To create a new menu, just use the menu name and a unique number that has not been previously used in the array.

3. After the control has been loaded, use the menu name and the index value to refer to the new menu. The menu item can then have all the normal properties of menus set at runtime.

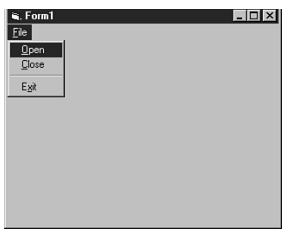


FIGURE 3.9▲
The File menu before runtime items are added.

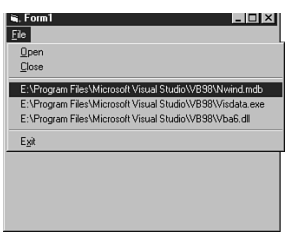


FIGURE 3.10▲
The File menu after runtime items have been added.

Dynamically created menu items appear directly under the preceding index item. This must be taken into consideration when incorporating menu items below the array. Menu items below the array will function as expected; however, as the new elements are added to the collection, regular menu items appear lower on the menu. Compare the positions of the Exit menu item on the File menu before and after dynamic menu items have been added, as shown in Figures 3.9 and 3.10.

Code for Runtime Menu Items

When a menu item has been created at runtime, it is part of a control array. When code is to be associated with the runtime-generated menu, just use the design time menu item that was the first index number in the array.

The template menu item will have an extra argument in the `Click` event. The `Index` argument provides the number used to refer to that control. The following sample code demonstrates one way to code for the dynamic menus:

```
Sub mnuFileItem_Click(Index as Integer)
    Select Case Index
        Case 0
            MsgBox "You clicked the first menu item!"
        Case 1
            MsgBox "You clicked the first dynamically
➥created menu item!"
        Case 2
            MsgBox "You clicked the second dynamically
➥created menu item!"
    End Select
End Sub
```

Of course, code such as the above would be appropriate only when you knew ahead of time exactly which items you would be adding to your menu at runtime and how many maximum items there would be. For a more dynamic example of menu programming, see Exercise 3.4.

Removing Runtime Menu Items

You can use two different methods to remove the runtime menus. The first is to hide the newly created item; the second is to unload it.

When hiding a menu item, the user interface will no longer display the item; however, program code can still use the menu and control the properties.

```
mnuFileItem(1).Visible = False
```

If a runtime menu item is unloaded, that control and the associated properties will be removed from memory. If required again, they will have to be loaded.

```
Unload mnuFileItem(1)
```

Only runtime control names and elements can be passed to the `Unload` statement. If a design time menu item is passed to `Unload`, an application error will occur because you can't unload controls created at design time.

Adding Controls to Forms

You can place controls on the surface of a valid VB container object. Standard container objects in VB include:

- Forms
- PictureBoxes
- Frames

Although `Image` controls and `PictureBox` controls have many features in common, `Image` controls cannot act as containers.

You can use two different methods to add controls to a form or other container object.

The first method uses a mouse double-click and requires the following steps:

STEP BY STEP

3.2 Adding Controls to Forms: Method 1

1. Select the form or other container object (such as a `PictureBox` or `Frame`) where you wish to add the control
2. Double-click the control's icon in the `ToolBox`. A new instance of the control will appear in the selected container.

With the second method, you draw the control on the container surface by following these steps:

STEP BY STEP

3.3 Adding Controls to Forms: Method 2

1. Single-click the control's icon in the `ToolBox`.
2. Release the mouse button and move the mouse pointer to the position on the container where you want to locate the control.
3. Hold down the mouse button and draw a rectangle on the container surface to indicate the size and position for the control instance.

Setting Properties for CommandButtons, TextBoxes, and Labels

You can set most of the properties of a control object at either design time or at runtime.

To set a control's property at design time:

STEP BY STEP

3.4 Manipulating a Control's Property at Design Time

1. Use the F4 key or choose View, Properties from the VB menu to bring up the Properties window, as shown in Figure 3.11.
2. Find the property in question on the list. You may navigate the properties list by pressing Shift + the first letter of the property name.
3. Set the property to the appropriate value.

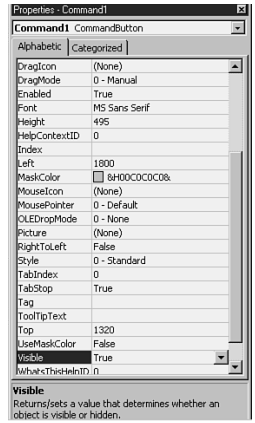


FIGURE 3.11
Setting properties in the Properties window.

Depending on the nature of the information that the property represents, you will find different ways to set the property:

- Properties that can take a wide range of numeric values or a string of text are usually changed by simply typing in the value you want to assign.
- Properties that represent Boolean values (such as the `CommandButton`'s `Cancel` and `Default` properties) can be toggled by double-clicking them.
- Properties that can hold a small range of named, enumerated values (such as the `MousePointer` property) usually exhibit a drop-down list of the possible values.
- Properties (such as `Font` or `BackColor`) that contain subcomponents or more visually complex information may pop-up their own property dialog box or Property Page when you double-click them.

To set a control property at runtime, use the familiar *object.property* syntax to assign a value to the property. For example if you want to assign the string `"Activate"` to the `Caption` property of the `CommandButton` named `cmdExecute` at runtime, you would put the line

```
cmdExecute.Caption = "Activate"
```

Other properties can take the name of the appropriate VB constant. In the following example, the `vbHourGlass` constant for the `MousePointer` can be found in the drop-down list of values at design time. The `vbRed` constant could be found in the ObjectBrowser's list of the VBRun library's Color constants.

```
txtName.MousePointer = vbHourGlass
lblName.BackColor = vbRed
```

Finally, for properties which are complex objects, you may need to use double-dotted syntax to refer to the property of a property of an object, as in the following examples with sub-properties of the `Font` property of a `TextBox`:

```
txtInvitation.Font.Bold = True
txtInvitation.Font.Name = "Arial"
```

Referring to a Property Within Code

If you need to read or write to a property within code, you need to refer to the object's name in front of the property name using the general syntax:

ControlName.PropertyName

For example, if you want to evaluate a CommandButton's `Enabled` property within an If condition, you can do it in one of two ways, as illustrated in the following examples:

```
If cmdAdd.Enabled = True Then
```

or

```
If cmdAdd.Enabled Then
```

Notice that, in this case, the `Enabled` property is a Boolean type and, therefore, you can imply a `True` value, as in the second example.

You can also assign a value to the default property without naming the property as long as you assign the correct data type for the property:

```
cmdAdd.Enabled = True
```

Each control has a default property (usually the most important property for the control in question). The default properties of the three controls under discussion here are:

- **CommandButton** `Value` property
- **Label** `Caption` property
- **TextBox** `Text` property

This property can be set or read in code simply by using the name of the control without the property name. So, for example, you could write the following code to set a `CommandButton`'s `Value` property, a `TextBox` control's `Text` property, and a `Label`'s `Caption` property:

```
cmdOK = True
txtName = "Jones"
lblName = "Name"
```

Some programmers might argue against this type of implicit reference to the default property on the grounds that it's a bit less clear in code. On the other hand, however, implicitly referring to the default property actually makes for faster performance at runtime.

Important Common Properties of `CommandButtons`, `TextBoxes`, and `Labels`

There are several properties that are shared by many of the standard controls.

`Name`

You use a control's `Name` property in code to refer to the control object when you want to manipulate its properties or methods. For example, if you name a `TextBox` `txtFirst`, you could write code to change its `Enabled` property and invoke its `Move` method as follows:

```
txtFirst.Enabled = True
txtFirst.Move 100,200,500,200
```

A control's `Name` also becomes part of all the event procedure names of that control. See the section in this chapter on "Assigning Code to a Control to Respond to an Event" for more discussion and its implications.

You should always rename a control a meaningful name as soon as you place it on its container. Most VB programmers use the "Hungarian notation" convention for naming controls and variables. This means that the name of each control begins with a lowercase prefix that is one to three (or sometimes four or five) letters long. The prefix is the same for all objects of the same type.

For instance, you should rename a `TextBox` control a name beginning with the letters `txt` as shown in the example given just above with `txtFirst`.

Although you can reference the control's `Name` property directly in code (though it's almost never necessary), you cannot change the `Name` property at runtime.

Enabled

The `Enabled` property of a control is a `True`/`False` property that you can set to determine whether or not the control can receive focus or respond to user-generated events such as the `Click` event. Many controls' `Caption` or `Text` properties (including the `CommandButton`, `TextBox` and `Label`) will appear fainter or "grayed out" to the user when you set their `Enabled` properties to `False`, as illustrated in Figure 3.12.

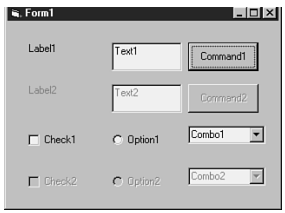


FIGURE 3.12
A form with `Enabled` and `Disabled` controls.

Since the `Label` control never gets focus, its `Enabled` property has no effect on whether the user can set focus to the `Label` (the user never could set focus to a `Label`, anyway). However, an enabled `Label` can still receive events such as the `Click` and `DblClick` events when the mouse pointer is over it. Setting the `Label`'s `Enabled` property to `False` disables these events for the `Label` as it does for other controls.

You can set a control's `Enabled` property at both design time and runtime.

NOTE

Different Use of a `Timer`'s `Enabled` Property The `Enabled` property for the `Timer` control has a different meaning from the other controls. When you set a `Timer`'s `Enabled` property to `True`, you cause the `Timer` event to fire at the interval of milliseconds specified by the `Timer`'s `Interval` property. When you set its `Enabled` property to `False`, the `Timer` event will not fire.

Visible

This property is `True` by default. Setting it to `False` means that the control will not be visible to the user. If you set `Visible` to `False` at design time, you (the programmer) will still be able to see the control on the design surface but the user won't be able to see it at runtime.

You can set the `Visible` property at both design time and runtime.

Font

This property is actually an object that contains many properties of its own. You can manipulate the `Font` object's properties through a design time dialog box that you can call up in one of two ways:

- Double-click the word "Font" in the control's Properties window (see Figure 3.13).
- Click the ellipsis button (…) to the right of the word "Font" in the Properties window.

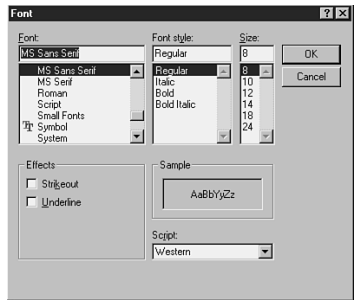


FIGURE 3.13
The `Font` property dialog box.

You may also refer to the `Font` object's sub-properties in your code by using double-dotted syntax or the With construct.

For instance, if you wanted to make the type in the `Label` named `lblName` appear bold (after first saving its original Bold setting), you would write the lines of code:

```
Dim blnOrigBold As Boolean
BlnOrigBold = lblName.Font.Bold
lblName.Font.Bold = True
```

However, if you wanted to refer to or manipulate several `Font` properties at the same time, it is more efficient to write lines such as

```
Dim blnOrigBold As Boolean, blnOrigUnderline As Boolean
Dim sOrigFontName As String
Dim iOrigSize As Integer

With lblName.Font
    BlnOrigBold = .Bold
    .Bold = True
    blnOrigUnderLine = .Underline
    .Underline = False
    sOrigFontName = .Name
    .Font = "Courier"
    iOrigSize = .Size
    .Size = 24
End With
```

Properties that Determine Size and Position

The `Height` and `Width` properties determine an object's size, while the `Left` and `Top` properties determine its position within its container object (`Form`, `Frame`, or `PictureBox`).

The unit of measure for these four properties is the unit of measure given by the `ScaleMode` property of the container. The default unit is the *twip* (twentieth of a point), but it may be different depending on whether or not you change the container's `ScaleMode`.

You can change each of these properties at runtime in your code, and you can also use the control's `Move` method to change a control's size and position.

TabIndex and TabStop

The TabIndex property determines the order in which controls on a form get focus when the user presses the Tab key. Note that Labels have a TabIndex property even though they can't receive focus.

The TabStop property lets you specify whether or not a user can use the Tab key to navigate to the control. Its default value is True. If you set TabStop to False, users will bypass the control as they cycle through the TabIndex order when they press the Tab key.

NOTE

The TabStop Property Only Controls Tab Key Navigation. Remember the TabStop property only controls user navigation with the Tab key. It says nothing about whether the user can set focus to the control in other ways (such as with the mouse or with access keys).

Assigning an Access Key to a TextBox Control Through a Label's Caption

Let's say that you have a TextBox representing the Department name on an entry screen. The Label to the left of the TextBox duly contains the Caption "Department."

If you put an ampersand (&) in front of the "D" in "Department," the "D" will appear underlined. When users see this, they will think, of course, that pressing Alt-d will place the cursor in the TextBox. Unfortunately, the "D" represents an access key for the Label—and moreover, the Label is incapable of receiving focus. Figure 3.14 illustrates a Label that implements an access key for a TextBox.

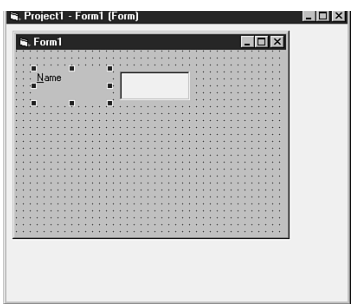


FIGURE 3.14
A Label that implements an access key for a TextBox.

However, if the user does press the Alt-D combination, focus will fall on the control following the Label in TabIndex order. Thus you can implement an access key for a TextBox control by defining an access key on a Label that immediately precedes the TextBox control in the TabIndex order.

So in order to give an access key to a TextBox, follow these steps:

STEP BY STEP

3.4 Giving an Access Key to a TextBox

1. Put a Label to the left of the TextBox.
2. Make sure the Label's UserMnemonic property is True (it should be, as that's the default value).
3. Give the Label a TabIndex that is one less than the TabIndex of the TextBox.

4. Put an ampersand (&) in front of one of the letters in the `Label`'s `Caption`. This letter will become the access key for the `TextBox`.

This technique works because the ampersand in the `Label`'s caption defines an access key, as discussed above in the section on the `Caption` property. However when the user presses the `Label`'s access key, the system is unable to set focus to the `Label` (because `Labels` cannot receive focus). Instead, the focus goes to the next control in the Tab order—which in this case is the `TextBox` that you've strategically placed to receive the focus from the `Label`.

Important Properties of the `CommandButton` Control

Along with the `TextBox` and `Label` controls, the `CommandButton` control is one of the most familiar sights in a Windows application.

The main function of the `CommandButton` control is to run code when the user clicks on it. You should keep the following points in mind when programming with `CommandButton` properties:

- You should always change the `Caption` property of a `CommandButton`. The `Caption` property contains the text displayed on the `CommandButton`. This property can be changed at design time and runtime.
- The `Cancel` property is a Boolean type. It allows VB to associate the Escape key with the `CommandButton`'s `Click` event procedure. Thus if you set `Cancel` to `True`, the `Click` event procedure fires when the user presses the Escape key. `Cancel` buttons are usually associated the with the Escape key.

  Notice that the `Click` event itself does not fire in this case. Instead only the `Click` event procedure runs. There is a difference when only the event procedure runs and the event itself does not fire. The most noticeable difference is that the focus does not change to the `CommandButton` when only the event procedure runs—as it would if the event itself had fired.

- The `Default` property is a Boolean type. It allows VB to associate the Enter key with the `CommandButton`'s `Click` event procedure. Thus, if you set `Default` to `True`, the `Click` event procedure will run if the user presses the Enter key. OK buttons are commonly associated with the Enter key.

  The comment made about the relation between the `Cancel` property's `Click` event and `Click` event procedure also applies to the `Default` property (i.e., the event procedure will fire, but the event itself will not run).

- Only one `CommandButton` on a form can have its `Default` property set to `True` at a time. If you set one `CommandButton` to be the default, all other buttons will have their `Default` property set to `False`. The same rule applies for the `Cancel` property.

- The `Value` property (the `Default` property for the `CommandButton`) is available only at runtime. If you set the `Value` property to `True` within your code, the `CommandButton`'s `Click` event procedure runs. Notice that, as in our comments for the `Cancel` and `Default` property, we say here that the `Click` event procedure will run, but the `Click` event will not fire. Since `Value` is the `CommandButton`'s `Default` property, code such as

  ```
  CmdOK = True
  ```

  would cause the `CommandButton`'s `Click` event procedure to run.

NOTE

`CommandButton` Controls `CommandButton` controls have a `BackColor` property that only takes noticeable effect when the `Style` property has been set to `1 - vbGraphical`.

Important Properties of the `TextBox` Control

The `TextBox` is one of the handiest and most popular controls in a Windows application. It's a common approach to getting free-form input from the user. You can manipulate a `TextBox` control's contents and detect changes the user makes to the `TextBox` through several properties:

- The `HideSelection` property is `True` by default and it signifies that the user-highlighted contents of the `TextBox` will not remain highlighted when focus shifts to another object. If you want the highlighted contents to remain highlighted, just set `HideSelection` to `False`.

- The `MaxLength` property determines the maximum number of characters that the user can enter into the `Text` property of the `TextBox`. If you set `MaxLength` to `0`, there is no limit on the number of characters that the user can enter (well, up to 32K, that is).
- The `Locked` property allows the `TextBox` to reject user changes but still allows the user to set focus to the `TextBox`. Therefore, the user can scroll through the contents of a `TextBox` without accidentally changing anything. Contrast this to the `Enabled` property which doesn't allow the user to set focus to the `TextBox`.
- The `MultiLine` property is only writable at design time, although you can find out its value at runtime. `MultiLine` is `False` by default, meaning that everything in the `TextBox` control will appear on a single line. If `MultiLine` is set to `True`, the `TextBox` will perform word-wrapping and also break a typed line after a hard return.
- The `PasswordChar` property defines a character that will appear on the screen in place of the actual characters in the `Text` property. The underlying value of the `Text` property still contains the actual characters typed by the user, but for each character typed, only the password character will appear. Typically programmers set this property to an asterisk (*) for `TextBox` controls that represent the password on login screens. If you want to eliminate the `PasswordChar` property for a `TextBox`, be careful to erase its old value with your keyboard's Delete key instead of simply overwriting it with the spacebar. A space in the `PasswordChar` property will cause the user's input to appear as a series of blanks!
- The `ScrollBars` property is only writable at design time, although you can check its value at runtime. The default value is `None`, but you can also choose `Horizontal`, `Vertical`, or `Both`. Scrollbars enable the user to scroll vertically through the contents of a multiple line `TextBox` or horizontally through wide contents of `TextBoxes`.
- The `SelText` property assigns or returns the contents of the currently selected text in a `TextBox`. If you assign a string to `SelText` property in your code, you'll replace the currently highlighted text with the contents of the new string and deselect whatever had been selected before.

- The `SelStart` property is an integer value that gives you the position of the first highlighted character in the `TextBox`. It's zero-based. If there's no text currently selected, `SelStart` will represent the current position of the text cursor within the `TextBox` control. If you change the `SelStart` property in your code, you'll deselect whatever text was highlighted and move the text cursor to the position given by `SelStart`.
- The `SelLength` property is an integer that indicates the number of selected characters in the `TextBox`. You can change `SelLength` in your code in order to change the number of selected characters. You can also deselect any highlighted characters by setting `SelLength` to `0`.
- The `Text` property is the `TextBox` control's default property. You can set it at design time or runtime, and you can also read it at runtime. The `Text` property represents the current visible, editable (not necessarily visible or editable, depending on the value of the `Visible` property, `Enabled` and/or `Locked`) contents of the `TextBox`. Since `Text` is the `TextBox` control's default property, code such as

  ```
  txtName = "Elizabeth"
  ```

  would have the effect of setting the `TextBox` control's `Text` property to "Elizabeth."

NOTE

`TextBox` Controls `TextBox` controls have no `Caption` property.

Important Properties of the `Label` Control

Some notable properties of the `Label` are listed here:

- Use the `Alignment` property to align text inside the `Label`—left-, right-, or center-aligned, or not aligned `(None)`.
- The `Appearance`, `BackStyle`, and `BorderStyle` properties together help determine the general appearance of the `Label`. For instance, if you leave `Appearance` at its default setting of `1-3d`, set `BackColor` to `vbWhite`, and set `BorderStyle` to `Fixed Single`, you can give the `Label` the same look as that of a `TextBox`. `BackStyle` is normally `Opaque`, but if you set it to `Transparent`, whatever is on the underlying form will show through and underlie the text in the `Label`'s `Caption`.

- Use the `AutoSize` and `WordWrap` properties to determine how the `Label` displays lengthy text in its `Caption`. If you set the `AutoSize` property of the `Label` to `True` (default is `False`), the `Label` automatically shrinks or stretches to the exact size needed to display the text. The `WordWrap` property determines whether or not an autosized `Label` changes size in a horizontal direction (`WordWrap` = `False`, its default value) or in a vertical direction (`WordWrap` = `True`). Remember, `WordWrap` has an effect only if you first set `AutoSize` to `True`.
- The `Label`'s `Default` property, `Caption`, holds the text that is visible to the user on the `Label`'s surface. You can change `Caption` at runtime. As noted in previous sections, putting an ampersand character (&) in front of a letter in the `Caption` will turn that letter into an access key for the control (usually a `TextBox`) that immediately follows the `Label` in the `TabIndex` order. Since `Caption` is the `Label`'s `Default` property, code such as

  ```
  lblName = "Name"
  ```

  would have the effect of setting the `Label` control's `Caption` to "Name."

NOTE

Visually Displaying an Ampersand in a `Caption` So what if you want an ampersand in a `Label`'s caption to really display as an ampersand—and not to function as an access key? No problem: Just set the `Label`'s `UseMnemonic` property to `False`. This property's default value of `True` indicates that an ampersand will define an access key.

You can also use "&&" to display a single ampersand. Using this second method, you could leave `UseMnemonic` with a value of `True` and place a single ampersand before another letter in the `Caption` to define an access key.

Note that although many controls have a `Caption` property, only the `Label` features the `UseMnemonic` property.

Assigning Code to a Control to Respond to an Event

To cause a control to react in a certain way to user or system activity, you must put code in the appropriate event procedure of the control. The VB IDE automatically provides event procedure stubs for a control as soon as you define an instance of a control by placing its icon on the form designer surface.

When you double-click the control instance, you call up a Code Window for one of the control's event procedures, as seen in Figure 3.15. Which event procedure you see first depends on one of two possibilities:

- If you haven't yet done anything with the control's event procedures, you first see the user interface `Default` event procedure, that is, the procedure for the event that's considered to be the most important event for the control.

NOTE

Events Versus Event Procedures You will often hear VB programmers loosely refer to writing code in "events." Technically, this is not correct. You don't write code in *events*; you write code in *event procedures*.

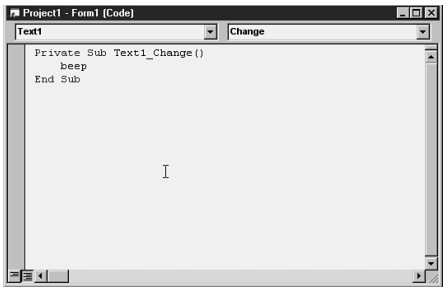


FIGURE 3.15
Editing an event procedure in the Code window.

- If you've already written some event procedure code, then you call up the first event procedure in alphabetical order for which you've already provided some code.

The default event procedures for the three most basic controls are:

- The `Click` event for the `CommandButton`.
- The `Change` event for the `TextBox`.
- The `Click` event for the `Label` control.

It's a very good idea to hold off writing any event procedure code until you've re-named all controls appropriately. We discuss this idea further in the following section.

Changing a Control Name After You Assign Code to the Event Procedure

As mentioned in the discussion of the `Name` property, a control's `Name` becomes part of all the event procedure names of that control. For example, the control named `txtFirst` would have event procedures named `txtFirst_Change`, `txtFirst_GotFocus`, `txtFirst_Click`, and so on.

If you change a control's `Name`, you automatically create new event procedures. If there is already code in the event procedures that uses the old name, those procedures will not be renamed and the code will become "orphaned."

The good news is that the old event procedures are not destroyed outright. However, if you want to get the old event procedures back, you must either copy and paste the code into the event procedures with the new names, rename the old procedures, or rename the control back to its previous name.

For example, if you add a `CommandButton` named `Command1` to a form, write code in its `Click` event procedure, and then change `Command1`'s name to `cmdOK`, the event procedure name does not change and would still be named `Command1_Click`. Therefore, the procedure would no longer be associated with the `CommandButton`. `CommandOK_Click` would be the name of the current `Click` event procedure, and obviously this event procedure begins life with no code.

Conversely, if you happen to write a general procedure and later rename a control in such a way that one of its event procedure names happens to match the name of the existing general procedure, then that general procedure becomes an event procedure for the control. For example, if you write a general procedure whose declaration looks like this:

```
Private Sub Bozo_Change()
```

and then later rename a `TextBox` control to "Bozo," VB associates the `Bozo_Change` procedure with the `TextBox` named "Bozo."

The `Click` Event

A control's `Click` event fires when the control is enabled and the user both presses and releases a mouse button while the mouse pointer is over the control. If the mouse pointer is over a disabled control or if the mouse cursor is over a blank area of the form, then the form receives the `Click` event.

The `Click` event is easy to understand, because it represents a common user action that occurs dozens of times during a single session in any Windows-based application.

Notice that in our definition of the `Click` event in the first paragraph, the user must both press and release the mouse button over the same control. The `Click` event won't occur if the user presses the mouse button over one control and then moves the mouse pointer off the control to release it. The same goes for a form's `Click` event: The user must both press and release the mouse button over an exposed area of the form or over a disabled control in order for the form to receive the `Click` event.

The `Click` event can also fire when the user presses a control's access key.

The following controls support the `Click` event as noted:

- **`CommandButton`** Only the *left* (or, for left-handed mice, the *primary)* mouse button fires a `Click` event for this control. Pressing Enter or SpaceBar when a `CommandButton` has focus will fire the `Click` event. Programmatically setting its `Value` property to `True` will cause the `Click` event procedure to run but will not fire the `Click` event.

NOTE

Choosing the Proper `Mouse` Event Procedure One of the tricks to learning Visual Basic is to determine which events to use in a given situation. When attempting to determine which mouse button was selected, many new programmers do not see much difference between the `Click`, `MouseUp`, and `MouseDown` events. The difference between any two events within VB lies in when they occur with respect to each other. The `Click` event procedure is the preferred place to detect a user's intentions.

The reason for using the `Click` event is to allow the user forgiveness. If the user did not mean to click on your object, or has pressed the button and then realized he did not want to, the mouse could still be removed from the object. If the mouse is dragged outside the area of the object, the `MouseUp` event will still occur. The `Click` event, however, will not occur.

Allowing your program to be forgiving means the user will have an easier time using the software, instead of fearing the next mistake.

NOTE

Pressing and Releasing the Mouse Button To distinguish between pressing and releasing the mouse button, see the discussion of `MouseUp` and `MouseDown` events later in this

WARNING

`Click` Event Procedure Versus Event Setting a `CommandButton`'s `Default` or `Cancel` property to `True` at design time will cause the `CommandButton`'s `Click` *event procedure* to run when the user presses Enter or Esc keys respectively. However, the `Click` *event* itself will not fire.

- **`Label`** Left or right mouse button fires a `Click` event.
- **`TextBox`** Left or right mouse button fires a `Click` event.

The `DblClick` Event

The `DblClick` event occurs on a form or a control when the object is enabled, the mouse pointer is directly over the form or control, and the user clicks the mouse twice in rapid succession. Windows determines whether the user's two clicks represent a double-click or two single clicks. The user can access the Windows Control Panel to set the maximum time interval between two clicks for these two clicks to count as a double-click.

NOTE

`DblClick` Event Only the primary (usually the left) mouse button will fire the `DblClick` event.

The `DblClick` event is not defined for the `CommandButton` in VB, but it is defined for the `Label` and the `TextBox` controls.

When the user double-clicks a form or a control that supports the `DblClick` event, Windows generates a `Click` event followed by a `DblClick` event for that control.

When the user double-clicks a control such as a `CommandButton` that doesn't support the `DblClick` event but does support the `Click` event, Windows will generate two `Click` events for the control.

`MouseUp` and `MouseDown`

The `MouseDown` event fires when the user presses a mouse button over a control or form. Similarly the `MouseUp` event occurs when the user releases the mouse button over a control or form. Note that if the user moves the mouse between the time the button was pressed and released, the same control (i.e., the control that originally received the `MouseDown`) will receive the `MouseUp` event.

During the `MouseDown` and `MouseUp` event procedures, you might want to know whether the left or right mouse button was pressed. You might also want to know whether or not one of the auxiliary keys (Shift, Alt, or Ctrl) also was depressed during the mouse event. Finally, it might be nice to know the relative position of the mouse pointer within the form or control receiving the event.

All of the foregoing information is available in parameters that the `MouseUp` or `MouseDown` event procedure receives from the system. The four parameters are

- **Button As Integer**. This is a value representing which mouse button fired the event. The value of this parameter is either `vbLeftButton`, `vbRightButton`, or `vbMiddleButton`. Again these terms are from the point of view of a right-handed mouse. `vbLeftButton` always refers to the primary button, regardless of whether it's physically the left or right button.
- **Shift As Integer**. This parameter represents an integer that indicates whether an auxiliary key is pressed during the `Mouse` event. It contains a value of 0 (none), 1 (Shift), 2 (Ctrl), 4 (Alt), or the sum of any combination of those keys. For example, if both the Ctrl and Alt key were pressed, the value of the Shift parameter is 6. You can check for the state of any one of the auxiliary keys with one of the VB constants `vbAltMask`, `vbCtrlMask`, or `vbShiftMask`. The following code illustrates how you could store the state of each auxiliary key in a Boolean variable within the `MouseDown` or `MouseUp` event procedure. The bit-wise representation of 1, 2, or 4 in the Shift parameter is 000000001, 000000010, 00000100. By doing a logical AND between the Shift parameter and one of the VB Shift-key constants, you can pick out whether each of the three Shift keys is currently pressed, as illustrated in Listing 3.1.
- **X As Single**. This is the horizontal position of the mouse pointer from the internal left edge of the control or form receiving the event.
- **Y As Single**. This is the vertical position of the mouse pointer from the internal top edge of the control or form receiving the event.

LISTING 3.1

TESTING THE SHIFT MASK IN A MOUSEDOWN OR MOUSEUP EVENT PROCEDURE

```
Dim blnIsAlt As Boolean
Dim blnIsCtrl As Boolean
Dim blnIsShift As Boolean
blnIsAlt = Shift And vbAltMask
blnIsCtrl = Shift And vbCtrlMask
blnIsShift = Shift And vbShiftMask
```

"Of course," you may be thinking, "isn't a `Click` event simply the combination of a `MouseUp` and a `MouseDown`?" How does the system handle this fact when the user clicks or double-clicks the mouse? In the next section we discuss how a VB program handles the combination of these various events.

Mouse Events Compared With `Click` and `DblClick`

Programming mouse-related events is a common task within most applications. This is because the mouse is the most common device for user interaction within a Windows-based user interface.

In particular, the `Click` event is possibly the most commonly programmed event in a VB application. All the mouse events except `MouseMove` are directly related to the action of one of the mouse buttons. For controls that support mouse events, the click-related events take place in the following order:

When the user clicks once over `Label` and `TextBox` controls:

1. `MouseDown`
2. `MouseUp`
3. `Click`

When the user clicks once over a `CommandButton` control:

1. `MouseDown`
2. `Click`
3. `MouseUp`

When the user double-clicks over `Label` and `TextBox` controls:

1. `MouseDown`
2. `MouseUp`
3. `Click`
4. `DblClick`
5. `MouseUp`

When the user double-clicks over a `CommandButton`, no `DoubleClick` event fires (the `CommandButton` does not support a `DoubleClick` event). Instead, the `CommandButton` receives (as you might imagine):

1. `MouseDown`
2. `Click`
3. `MouseUp`
4. `MouseDown`
5. `Click`
6. `MouseUp`

MouseMove

The `MouseMove` event fires every time the mouse moves over a form or control. The `MouseMove` event could therefore fire dozens of times as the user quickly and casually moves the mouse. A user can fire several `MouseMove` events in rapid succession just by being bored enough to move the mouse around while waiting for relatively long processes to complete (for example, data access, or some ActiveX Automation call).

The `MouseMove` event has the same parameters as the `MouseUp` and `MouseDown` events.

You might use the `MouseMove` event to react to the user moving the MousePointer onto a control.

The `Change` Event

We discuss the `Change` event in greater detail in the chapter on Input Validation. We mention it here briefly because the `Change` event is the `TextBox` control's user default event. It fires every time the `Text` property alters. This event can fire due to user input or also because your code has done something to change the `Text` property.

`Labels` also receive a `Change` event whenever the `Caption` changes at runtime. Although user input could never fire a `Label`'s `Change` event (since `Labels` can't receive user input), the `Label`'s `Change` event might fire if the `Label Caption` were changed in code.

Other Events Commonly Used for Input Validation

In the chapter on Input Validation, we discuss several more events at length. We mention them briefly here as well for completeness:

- `Keystroke` events, including the `KeyPress`, `KeyUp`, and `KeyDown` events, fire when the user hits keys at the keyboard.
- `GotFocus` and `LostFocus` events happen when focus comes to or leaves a control. Note, of course, that the `Label` does not support these two events (it can't get focus).

CHAPTER SUMMARY

KEY TERMS

- Access key
- Context-sensitive menu
- Event
- Event procedure
- Pop-up menu
- Right-mouse menu
- Shortcut key
- Sub menu
- Top menu

This chapter covered the following topics:

- Creating and editing a menu with the Menu Editor
- Modifying a menu item's appearance in code
- Programming with Pop-Up menus
- Programming with menu control arrays
- Important properties and events of `CommandButtons`, `TextBoxes`, and `Labels`

APPLY YOUR KNOWLEDGE

Exercises

3.1 Creating a Simple Menu

In this exercise, you create a simple menu for the `Form` object. A standard File menu will be created with common sub-menu items and also a View and Help menu. To create this template, follow these steps:

Estimated time: 10 minutes

1. Start Visual Basic 6.
2. Create a Standard EXE project.
3. Select Tools, Menu Editor. Make sure that `Form1` is currently selected. Otherwise the editor will not be available. At the end of this project, the menu editor should look like Figure 3.16.

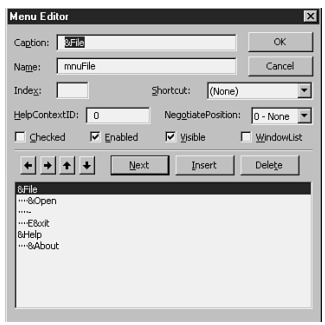


FIGURE 3.16
Menu Editor with completed menus for Exercise 3.1.

4. In the `Caption` text box, type **&File**. Remember that the ampersand is used for shortcut key designation.
5. In the Name text box, type **mnuFile**. The mnu is the three-letter object prefix, and File is the top-level menu caption.
6. Click on the Next button. Notice the &File appears on the left edge of the bottom list box. This menu item will be a top-level menu because it has not been indented.
7. The caption should now be empty. To enter the next menu, type **&Open** in the `Caption` property.
8. In the Name text box, type **mnuFileOpen**. The mnu is the three-letter object prefix, File is the top-level menu, and Open is a sub-menu of File.
9. Click on the right arrow. Notice that the &Open menu item should now be indented one position under the &File menu. This represents a sub-menu item.
10. Continue creating the menus by following Table 3.1. Remember to click on Next to create a new menu and watch the indents. Remember even separator items must have distinct names (see Table 3.1).

TABLE 3.1

MENUS TO CREATE

Caption	*Menu Name*	*Position*
-	mnuFileSep1	Sub-menu of File
E&xit	mnuFileExit	Sub-menu of File
&Help	mnuHelp	Top-level menu
&About	mnuHelpAbout	Sub-menu of Help

11. On `Form1`, click on the File, Exit menu items.
12. In the open code window, enter the following code:

```
Sub mnuFileExit_Click()
    Unload Form1
    End
End Sub
```

APPLY YOUR KNOWLEDGE

13. On `Form1`, click on the Help, About menu item.
14. In the open code window, enter the following code:

```
Sub mnuHelpAbout_Click()
    MsgBox "This is a test application", _
            vbInformation
End Sub
```

15. Run the project.
16. Select the Help, About menu. A message box should appear.
17. Click on OK to close the message box.
18. Select the File. Notice the Separator line. This appears due to the "-" only being used in the caption of the menu item.
19. Choose Exit from the menu. This ends the application.

This exercise demonstrated creating a simple menu system attached to the form object. Code was also assigned to various menu items so that the code would execute when the menu item was selected.

3.2 Dynamically Modify the Appearance of a Menu

In this exercise, you create a menu that is dynamically modified during runtime. To create this template, follow these steps:

Estimated time: 20 minutes

1. Start Visual Basic 6.
2. Create a Standard EXE project.
3. On `Form1` create a menu bar with the items in Table 3.2.

TABLE 3.2

MENU BAR ITEMS

Caption	*Menu Name*	*Position*
&File	mnuFile	Top-level menu
&Open	mnuFileOpen	Sub-menu of File
&Close	mnuFileClose	Sub-menu of File
-	mnuFileSep1	Sub-menu of File
E&xit	mnuFileExit	Sub-menu of File
&View	mnuView	Top-level menu
&Toolbar	mnuViewToolbar	Sub-menu of View
Status &Bar	mnuViewStatusBar	Sub-menu of View
-	mnuViewSep1	Sub-menu of View
&Options	mnuViewOptions	Sub-menu of View

4. Set Shortcut keys as shown in Figure 3.17.
5. With the Menu Editor open, choose the &Close menu option and remove the check mark from `Enabled`. This disables the menu by default. The appearance of the Menu Editor should now resemble Figure 3.17.

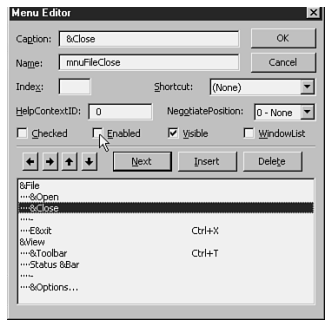


FIGURE 3.17
Menu Editor with completed menus for Exercise 3.2 and &Close menu option disabled.

APPLY YOUR KNOWLEDGE

6. Close the Menu Editor.
7. From `Form1` select File, Open. In the Code window, enter the following code for the menu:

```
Sub mnuFileOpen_Click()
    mnuFileClose.Enabled = True
    mnuFileOpen.Enabled = False
End Sub
```

8. From the Code window, find the `FileClose Click` event procedure and enter the following code for the menu:

```
Sub mnuFileClose_Click()
    mnuFileOpen.Enabled = True
    mnuFileClose.Enabled = False
End Sub
```

9. From `Form1` select View, Toolbar. In the Code window, enter the following code for the menu. This code acts as a toggle switch. If it is on, turn it off. If it is off, turn it on:

```
Sub mnuViewToolbar_Click()
    mnuViewToolbar.Checked =
➥Not(mnuViewToolbar.Checked)
End Sub
```

10. From `Form1` select View, Status Bar. In the Code window, enter the following code for the menu:

```
Sub mnuViewStatusBar_Click()
    mnuViewStatusBar.Checked =
➥Not(mnuViewStatusBar.Checked)
End Sub
```

11. Run the project.
12. From `Form1` select File, Open. Once clicked, return to the File menu. The Open menu should now be disabled, and the Close menu should be enabled.
13. From `Form1` select File, Close. Once clicked, the Open menu should be enabled, and Close should be disabled.
14. From `Form1` select View, Toolbar. Once clicked, return to the View menu. The toolbar should have a check mark beside it.
15. From `Form1` select View, Status Bar. Once clicked, the status bar should have a check mark. Notice that status bar and toolbar do not affect each other. One can be on, the other off, or both the same.
16. End the application.

This exercise demonstrated simple, dynamic changing of the menu appearance. Although no program code was called in this project, each menu item could contain VB commands or calls to other procedures. When changing the appearance of menus, caution should be taken if the code cannot execute or is not successful. The menu appearance should not be altered. By executing code first and changing appearance last, the programmer can better handle the interface needs.

3.3 Add a Pop-Up Menu to an Application

In this exercise, you create a form that uses a pop-up menu. To create this template, follow these steps:

Estimated time: 15 minutes

1. Start Visual Basic 6.
2. Create a Standard EXE project.
3. On `Form1` place a text box in lower portion of the form. Leave half the form empty. This will provide an area for the custom pop-up menu.
4. On `Form1` create a menu bar with the structure in Table 3.3.

APPLY YOUR KNOWLEDGE

TABLE 3.3

MENU BAR STRUCTURE

Caption	*Menu Name*	*Position*
MyMenu	mnuMyMenu	Top level
Cu&t	mnuMyMenuCut	Sub of MyMenu
&Select	mnuMyMenuSelect	Sub of MyMenu
&About	mnuMyMenuAbout	Sub of MyMenu

5. With the Menu Editor open, select MyMenu and set the menu to be invisible.
6. Open the code window for `Form1`.
7. Find the form `MouseUp` event and enter the following code:

```
Sub Form_MouseUp(Button As Integer, Shift
➥As Integer, X As Single, Y As Single)
    If Button = vbRightButton Then
        Form1.PopupMenu mnuMyMenu
    End If
End Sub
```

8. Run the project.
9. Click on the form background. Ensure that you are not on the background of the text box.
10. Right-click on the form background. Although no menu bar should appear at the top of the form, your custom pop-up menu should appear. Make sure you use the alternate mouse button to get the menu.
11. The menu items can be clicked but have no code attached.
12. To see a built-in control pop-up menu, right-click on the `TextBox` control window. The menu should appear automatically.
13. Type some words into the text box. Within your pop-up menu, select to copy and paste some text. Notice that you did not program this functionality.
14. End the application.

This exercise demonstrated how to use both a customized pop-up menu and a built-in, pop-up menu. When customized menus are required, just build them as a regular menu and hide the top-level menu if desired. Built-in menus require no coding or trapping of the mouse but cannot be overridden in VB.

3.4 Create an Application That Adds and Deletes Menus at Runtime

In this exercise, you create a form that adds and deletes menus at runtime. To create this template, follow these steps:

Estimated time: 25 minutes

1. Start Visual Basic 6.
2. Create a Standard EXE project.
3. Under the Project menu, choose Components.
4. Check the Microsoft Common Dialog Control. A new tool icon should be added to the toolbox.
5. Add the new tool to `Form1`. It should appear as a small gray square. This control provides the Open, Save As, Color, Font, Printer, and Help dialog boxes from Windows. The control is invisible at runtime.
6. Change the `Name` property for the object to CDC1.
7. Open the code window and, in the General Declarations section for `Form1`, enter the following code:

```
Private iItem as Integer
```

8. Create the menu structure for `Form1` as shown in Table 3.4.

APPLY YOUR KNOWLEDGE

TABLE 3.4

FORM1 MENU STRUCTURE

Caption	*Menu Name*	*Position*
&File	mnuFile	Top-level
&Open...	mnuFileOpen	Sub-menu of File
&Close	mnuFileClose	Sub-menu of File
-	mnuFileItem	Sub-menu of File
-	mnuFileSep1	Sub-menu of File
&Exit	mnuFileExit	Sub-menu of File

9. Using the Menu Editor, select `mnuFileItem`; set the `Index` property to `0` and make the menu item invisible. This will be your first element in the menu array, and it will be invisible until an array element has been assigned. After completing this action, the Menu Editor should appear, as in Figure 3.18.

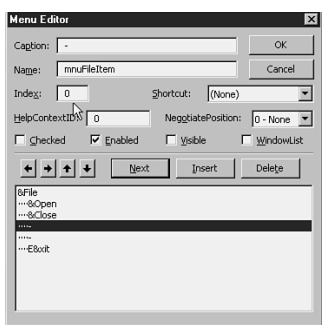


FIGURE 3.18
Menu Editor with completed menus for Exercise 3.4 and `Index` property set for `mnuFileItem`.

10. Open the File menu while in Design mode. Notice that the Open, Close, Sep1, and Exit menu items are visible, but the first array element—Separator—is not.

11. Select the File, Open menu item. In the Code window, enter the following code:

```
Sub mnuFileOpen_Click()
    Dim iLoop As Integer
    cdc1.ShowOpen
    If cdc1.filename <> "" Then
        mnuFileItem(0).Visible = True
        For iLoop = 0 To iItem
            If mnuFileItem(iLoop).Caption =
➥cdc1.filename Then
                Exit Sub
            End If
        Next iLoop
        iItem = iItem + 1
        Load mnuFileItem(iItem)
        mnuFileItem(iItem).Caption =
➥cdc1.filename
        mnuFileItem(iItem).Visible = True
    End If
End Sub
```

12. Add a `CommandButton` to `Form1`. Change the `Caption` property to &Clear File List.

13. Open the Code window for the Clear File List button, and enter the following code:

```
Sub Command1_Click()
    Dim iLoop as Integer
    mnuFileItem(0).Visible = False
    For iLoop = 1 to iItem
        Unload mnuFileItem(iLoop)
    Next iLoop
    iItem = 0
End Sub
```

14. Run the project.

15. From `Form1` select File, Open. Choose any directory, and then choose any file. The file will not be opened, but the path and filename will appear under the File menu. This is similar to the Most Recently Used list found in Applications.

16. Using File, Open, select as many different filenames as you like. If the path and filename are already in the list, they will not be added again.

APPLY YOUR KNOWLEDGE

17. After you have selected a few different files, use the Clear File List button. This button will use a form-level variable which indicates the number of menu array elements and unloads each element.
18. After the list is cleared, the Open menu can be used again followed by the Clear File List button.
19. End the application.

This exercise demonstrated how runtime menus are created based on a design time template. The template is the first item in a control array, and new menu elements are loaded dynamically at runtime. Unloading the items just requires the array element's index and the `Unload` statement to remove the menu from memory.

3.5 Test Control Events

In this exercise, you create a form that tests the event procedures of many of the events discussed in this chapter. To create the application, follow these steps:

Estimated time: 35 minutes

1. Start Visual Basic 6.
2. Create a Standard EXE project and populate the default startup form with controls as shown in Figure 3.19. Note the upper control in the frame for the Mouse events is a Label with its `BorderStyle` property set to 1-Fixed Single.
3. You should name the controls according to their respective captions as shown in the figure: `cmdCommand1Value`, `Command1`, `cmdOption1ValueTrue`, `cmdCheck1Value0`, `cmdCheck1Value1`, `cmdCheck1Value2`, `Option1`, `Option2`, `Check1`, `cmdClickAndDblClick`, `txtKeyStroke`, `lblClickAndDblClick`, and `cmdClickAndDblClick`. Name the two `CommandButtons` on the lower right corner of the form cmdClearForm and cmdQuit respectively.

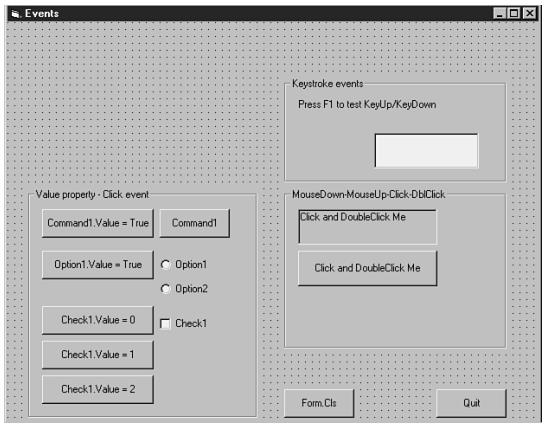


FIGURE 3.19
Form and controls for Exercise 3.5.

4. Enter the following code into `cmdClearForm` and `cmdQuit`'s `Click` event procedures:

```
Private Sub cmdClearForm_Click()
    Me.Cls
End Sub
Private Sub cmdQuit_Click()
    Unload frmEvents
    Set frmEvents = Nothing
    End
End Sub
```

5. `CmdQuit` will allow you to gracefully terminate the application by closing the form properly. `CmdClearForm` will clear the results of old calls to the form's `Print` method from the form's surface. You will want to use this button frequently in the following steps so that you can clearly see when certain event procedures run and when they do not.
6. Enter the following code into the indicated event procedures of `txtKeyStroke`:

```
Private Sub txtKeyStroke_KeyDown(KeyCode As
➥Integer, Shift As Integer)
    If KeyCode = vbKeyF1 Then
        Me.Cls
```

APPLY YOUR KNOWLEDGE

```
        Me.Print "KeyDown."
    End If
End Sub
Private Sub txtKeyStroke_KeyPress(KeyAscii
➥As Integer)
    KeyAscii = Asc(UCase(Chr(KeyAscii)))
End Sub
Private Sub txtKeyStroke_KeyUp(KeyCode As
➥Integer, Shift As Integer)
    If KeyCode = vbKeyF1 Then
        Me.Cls
        Me.Print "KeyUp."
    End If
End Sub
Private Sub txtKeyStroke_Change()
    Beep
End Sub
```

7. Run the app and experiment with keyboarding into the `TextBox`. Notice that the state of the Caps Lock key doesn't matter, because everything always comes through as upper case. Notice the annoying beep every time you do anything to change the contents of the `TextBox`. Finally notice that, upon pressing the F1 key, the `KeyDown` event fires, and only when you release it does the `KeyUp` event fire.

8. Stop the app and enter the following code into `lblClickAndDblClick's` event procedures:

```
Private Sub lblClickAndDblClick_Click()
    Me.Font.Bold = True
    Me.Print "Label Click"
    Me.Font.Bold = False
End Sub
Private Sub lblClickAndDblClick_DblClick()
    Me.Print "Label DblClick"
End Sub
Private Sub
lblClickAndDblClick_MouseDown(Button As
➥Integer, Shift As Integer, X As Single, Y
➥As Single)
    Me.Cls
    Me.Print "Label MouseDown"
End Sub
Private Sub
lblClickAndDblClick_MouseUp(Button As
➥Integer, Shift As Integer, X As Single, Y
➥As Single)
    Me.Print "Label MouseUp"
End Sub
```

9. Run the app and experiment with clicking and double-clicking the mouse over this `Label`. Try disabling the `Label` to verify that it does not then receive these events.

10. Stop the app and enter the following code into the event procedures of `cmdClickAndDblClick`:

```
Private Sub cmdClickAndDblClick_Click()
    Me.Font.Bold = True
    Me.Print "CommandButton Click"
    Me.Font.Bold = False
End Sub
Private Sub
cmdClickAndDblClick_MouseDown(Button As
➥Integer, Shift As Integer, X As Single, Y
➥As Single)
    Me.Cls
    Me.Print "CommandButton MouseDown"
End Sub
Private Sub
cmdClickAndDblClick_MouseUp(Button As
➥Integer, Shift As Integer, X As Single, Y
➥As Single)
    Me.Print "CommandButton MouseUp"
End Sub
```

11. Run the app and notice the different behavior of the `CommandButton`'s mouse events compared to the `Label`.

12. Stop the app and in `Command1`'s and `cmdCommand1Value`'s `Click` event procedures, enter the following code:

```
Private Sub cmdCommand1Value_Click()
    Command1.Value = True
End Sub
```

13. Run the app and notice that setting `Command1`'s `Value` property has the same effect as clicking it.

14. Stop the app and in the `Click` event procedures of Option1 and Check1, enter the following code:

```
Private Sub Option1_Click()
    Me.Cls
```

APPLY YOUR KNOWLEDGE

```
    Me.Print "Option1_click"
End Sub
Private Sub Check1_Click()
    Me.Cls
    Me.Print "Check1_Click"
End Sub
```

15. Run the app and notice that the `Click` events only run when you change the value of the `OptionButton` or `CheckBox`.

16. Stop the app and in the `Click` event procedures of `cmdOption1ValueTrue`, `cmdCheck1Value0`, `cmdCheck1Value1`, and `cmdCheck1Value2`, enter the following code:

```
Private Sub cmdOption1Value_Click()
    Option1.Value = True
End Sub
Private Sub cmdCheck1Value0_Click()
    Check1.Value = 0
End Sub
Private Sub cmdCheck1Value1_Click()
    Check1.Value = 1
End Sub
Private Sub cmdCheck1Value2_Click()
    Check1.Value = 2
End Sub
```

17. Run the app and notice the effect of setting the Values of the `OptionButton` and `CheckBox`.

3.6 Use Common Control Properties

In this exercise, you create a form that uses many of the control properties discussed in this chapter. To create the application, follow these steps:

Estimated time: 10 minutes

1. Start Visual Basic 6.

2. Create a Standard EXE project and populate the default startup form with controls as shown in Figure 3.20. Name the `TextBoxes` on the form `txtName`, `txtDepartment`, `txtLoginID`, and `txtPassword` and name the `CommandButtons` `cmdOK` and `cmdQuit`. You can leave the `Labels` with their default names.

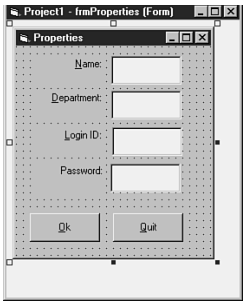


FIGURE 3.20
Form and controls for Exercise 3.6.

3. Modify the `TabStop` property of `txtPassword` so that the user cannot access it with the Tab key.

4. Convert the first letter of each of the first three Labels into an access key for the `TextBox` immediately to the right of the respective `Label`. Remember that the `TabIndex` property of the `Label` must immediately precede the `TabIndex` of the `TextBox` for which the `Label` provides the access key.

5. Change the `PasswordChar` property of txtPassword so that the user only sees asterisks (*) when typing into this control. Change the `MaxLength` property of `txtPassword` so that the user can enter no more than eight letters.

6. Experiment with the `Locked`, `Enabled`, `MultiLine`, and `Scrollbars` properties of the other three `TextBoxes`.

7. Put the following code into the `Click` event procedures of the `CommandButtons`:

```
Private Sub cmdOK_Click()
    MsgBox "It's OK"
End Sub
Private Sub cmdQuit_Click()
    MsgBox "Abandoning User Changes"
End Sub
```

APPLY YOUR KNOWLEDGE

8. Set the Default property of `cmdOK` to `True` and the `Cancel` property of `cmdQuit` to `True`. Now run the app, make sure that the cursor is in one of the `TextBox`es, and observe the effects of pressing the Enter key and the Esc key. Notice that the `CommandButton`s do not receive focus even though their event procedures run. This is because the events themselves do not fire when Enter or Esc is pressed—only the event procedures run.

Review Questions

1. Visual Basic forms can be used to host a menu bar. What two methods can be utilized by Visual Basic to create these menus?
2. When designing application menu bars, a variety of terminologies are used to refer to the different levels and positions held by a menu item. What is the name of a menu item that appears directly under a window's title bar?
3. When using the Menu Editor, how can the programmer define which menus will be at the top, which will be sub-items, and how the order is controlled?
4. Pop-up menus are displayed when the user selects the right mouse button. This displays a menu with options specific to the area that the user has selected. In which event, in Visual Basic, would a programmer trap for the use of the right mouse button?
5. Right-clicking on certain objects provides the user with a menu. What names are used to refer to this menu?
6. Menus can be created dynamically in Visual Basic by declaring a menu array and then loading a new element into the menu array. Once loaded, the new menu can have its properties set. True or False? Why?
7. The `Unload` statement is used to remove menu items created at runtime. Can the `Unload` statement also be used for removing the design time menu template that is the first element in the array?
8. Describe two mouse techniques for adding controls to a form.
9. Name three objects that can contain controls.
10. What are the default properties, respectively, of `CommandButton`s, `Labels`, and `TextBox`es. What does each of these properties represent?
11. What is the syntax for changing a control property's value in VB code?
12. What problem might happen to a control's event procedure when the programmer renames the control?

Exam Questions

1. Menus in Visual Basic can have their appearance changed to reflect the state of the application. To change menu items in VB at runtime, which methods can be used?

   A. The Menu Editor

   B. Program code

   C. Program code and the Menu Editor

   D. Menu property of a form

   E. Negotiate property of a form

APPLY YOUR KNOWLEDGE

2. The Menu Editor enables the programmer to create menu objects. These objects can have a variety of properties set and changed. Which property cannot be affected by using the Menu Editor?

   A. `Checked`

   B. `Visible`

   C. `Picture`

   D. `Shortcut Key`

   E. `Enabled`

3. In certain applications, objects provide a shortcut menu by using the right mouse button. This pop-up menu can be used to provide common commands for the selected object. Which item best describes the requirements to create a pop-up menu?

   A. Define the top-level and sub-level menu item, trap for the right mouse button in the `MouseUp` event, and call the form `PopupMenu` method.

   B. Define the menu items, trap for the right mouse button in the `MouseUp` event, and call only the selected menu items.

   C. Use the pop-up property of the object.

   D. Assign a top-level menu to the pop-up menu property of the object.

   E. Use the Menu Editor, define the top-level menu name as "pop-up," and in the `MouseUp` event call the menu.

4. Assume that the Menu Editor has already been used to create a menu structure. For the menus to provide the functionality required, how are the menu items assigned program code?

   A. By using the `ItemCode` property in the editor

   B. By using the `Menu` property in the Properties window

   C. By using the `ItemCode` property in the Properties window

   D. By selecting the menu item from the form and entering code into the opened Code window

   E. None of these

5. When a menu control array is created, runtime menu items can be added to the array. When the element's properties are set, where in the menu structure do the new items appear?

   A. In the order specified by the `NegotiatePosition` property

   B. In the order specified by the array element

   C. Immediately below the preceding array element

   D. At the bottom of the menu specified at design time

   E. None of these

6. You can specify that a menu control will be a separator bar by

   A. Supplying a space(" ") as the `Name` property

   B. Supplying a dash ("—") as the `Caption` property

   C. Supplying a dash ("—") as the `Name` property

   D. Supplying an underscore ("_") as the `Caption` property

7. If `Text1` is selected and the following line of code runs

```
Text1.SelText = "Hello"
```

APPLY YOUR KNOWLEDGE

A. The text "Hello" replaces the previously selected text.

B. The text "Hello" inserts into the `TextBox` in front of the previously selected text.

C. `Text1.SelLength` becomes 5

D. Everything visible in `Text1` disappears and is replaced by "Hello."

8. The value of the `Name` property

A. Can't be read in code at runtime

B. Provides a default value for the `Caption` property for all controls having a `Caption`

C. Takes its default from the `Caption` property

D. Can't be changed at runtime

9. To prevent the user from being able to give focus to a control under any circumstances, you may (select all that apply)

A. Set the control's `TabIndex` property to `0`.

B. Set the control's `TabStop` property to `0`.

C. Set the control's `TabStop` property to `False`.

D. Set the control's `Enabled` property to `False`.

10. Which of these statements will assign the value of the `CommandButton`'s `Caption` property to the `TextBox`'s `Text` property?

A. Text1 = Command1

B. Text1 = Command1.Caption

C. Text1.Text = Command1

D. Text1.Text = CStr(Command1)

11. If a `TextBox`'s `Enabled` property is `False`, then

A. The text in the box will be grayed, and the user won't be able to set focus to the `TextBox`.

B. The text in the box will be grayed, the user can still set focus to the `TextBox`, but the user won't be able to make changes to the text.

C. The text in the box will be grayed, and the user can still make changes to the text.

D. The text in the box will appear normal, the user can set focus to the `TextBox`, but the user can't make any changes.

12. An access key for a `TextBox` control

A. Can be provided in the `TextBox`'s `Caption` property

B. Can be provided in the `TextBox`'s `Text` property

C. Can be provided in an accompanying `Label` control

D. Can be provided in the `TextBox`'s `Label` property

Answers to Review Questions

1. The two methods used by Visual Basic to create menus for an application are the VB Menu Editor and the Win32 API. Both methods can be used from VB to generate menu systems. The built-in Menu Editor is a simple dialog box that enables the user to create a hierarchy of menu items and menu item order. The Win32 API is an external set of functions provided by the operating system and allows for a wide variety of functions. See "Understanding Menu Basics."

APPLY YOUR KNOWLEDGE

2. The term used to refer to a menu item found directly under a window's title bar is a top-level menu. This menu item is used to group other items into a sub-menu, which will appear under the top-level item when it is selected. See "Knowing Menu Terminology."

3. When creating menus with the Menu Editor, the programmer uses the left and right arrows. The arrows allow the menu hierarchy to be customized as required. The up and down arrows of the Editor allow the items to be ordered from top to bottom. See "Using the Menu Editor."

4. The `MouseUp` event can be used to determine which mouse button has been clicked. By using a specific object's `MouseUp` event, the programmer can determine whether the right mouse button was used. If so, the form `PopupMenu` method can be called. See "Determining the Mouse Button."

5. The menu provided when a user right-clicks on an object has a variety of names. One of the most common terms used is the "pop-up menu." Another term is the "context-sensitive menu." Also used is the "right mouse menu." All terms refer to the same menu. Certain objects and the operating system provide the menu, or they can be created in Visual Basic. See "Adding a Pop-Up Menu to an Application."

6. True. This is one way to allow for the customization of menu items on a menu bar. By setting the index value of one menu item at design time, new items can be dynamically loaded and controlled through code. See "Creating an Application that Adds and Deletes Menus at Runtime."

7. No. The `Unload` statement can only be used to remove instances of menu items created at runtime. If the first element in the array—assuming it is the design time item—is passed to `Unload`, an application error will occur: `Can't unload controls created at design time`. See "Removing Runtime Menu Items."

8. First technique: Double-click on the control's icon in the `ToolBox`. Second technique: Single-click on the control and then use the mouse to draw its rectangular outline on the container's surface. See "Adding Controls to Forms."

9. Some objects that can contain controls are: `Forms`, `PictureBoxes`, and `Frames`. Note: `Image` controls cannot contain other controls. See "Adding Controls to Forms."

10. `Value` property, `Caption` property, and `Text` property, respectively. The `CommandButton`'s `Value` property is a `True/False` property that you can set in code to fire the `CommandButton`'s `Click` event. The `Label`'s `Caption` property represents the text that the user sees on the `Label`'s surface. The `TextBox` control's `Text` property represents editable text that appears in the `TextBox`. See "Referring to a Property Within Code," "Important Properties of the `CommandButton`," "Important Properties of the `TextBox` Control," and "Important Properties of the Label Control."

11. `ControlName.PropertyName = NewValue`. See "Referring to a Property Within Code."

12. When you re-name a control, the control gets a brand-new event procedure with a new name. If you wrote code that you wrote in the event procedure before the name change, that code stays in the procedure with the old name. Therefore, any code that you wrote in the old event procedure before you changed the control's name is no longer associated with the control's event. See "Changing a Control Name After You Assign Code to the Event Procedure."

APPLY YOUR KNOWLEDGE

Answers to Exam Questions

1. **B**. The menu items can only be affected by program code at runtime. If an application is to have dynamic menus that reflect the state of the application, the program code will allow the menu objects to be controlled. The Menu Editor is only available for application forms that have a menu and are in Design mode. For more information, see the section "Using the Menu Editor."

2. **C**. The `Picture` property cannot be accessed by using the Visual Basic Menu Editor. This property is one of the special features of a menu that requires using the Win32 API. By using a special external call from Visual Basic, a menu can be assigned a picture. The new applications from Microsoft, such as Word 97 and Excel 97, show this capability. For more information, see the section "Using the Menu Editor."

3. **A**. The top-level menu is created first, and the pop-up menu items are then created as sub-level menu items of the top-level menu. The next step is to test for a right-click on the selected object. After the right-click has been detected, the form `PopupMenu` method is called, and the top-level menu that has been defined is passed as an argument. For more information, see the section titled, "Defining the Pop-Up Menu."

4. **D**. You can bring up a menu item's code window by selecting the menu item from the design time form with the mouse or keyboard. The code window will contain the procedure stub for the menu object's `Click` event. The programmer just has to enter the desired code to be run. For more information, see the section titled, "Attaching Code to a Menu Item's `Click` Event Procedure."

5. **B, C**. When runtime menus are created, they are displayed according to their element number and appear directly below the menu with the preceding element number. The only exception is if the element is not displayed because the `Visible` property has been set to `False`. Runtime menus will always be on the same menu where they were created at design time, directly below the array elements that were created at design time. For more information, see the section titled, "Creating an Application that Adds and Deletes Menus at Runtime."

6. **B**. You can specify that a menu control will be a separator bar by specifying a dash as the `Caption` property. This is the only way to get a separator bar, so none of the other options will work. See "Using the Menu Editor."

7. **A**. The code Text1.Seltext="Hello" causes "Hello" to replace the previously selected text and sets the value of Text1.SelLength to 0 (after SelText is set, no text is selected). Setting the value of SelText replaces the currently selected text with the new string, leaving the new string as the currently-selected text. Thus the length of the newly-selected text would be the same as the length of the string. Answer D would be true only if everything in the `TextBox` were selected before the line of code ran. See "Important Properties of the `TextBox` Control."

8. **D**. The value of the `Name` property can't be changed at runtime. In some earlier versions of VB, the `Name` property couldn't be read at runtime, but now it can (this has been true for several versions already). The `Name` property does not provide a default value for the `Captions` of Menu items (you must always type in their names manually).

APPLY YOUR KNOWLEDGE

In other cases the `Name` property definitely serves as the default value for the `Caption` and not the other way around. See "Name" under the section "Important Common Properties of `CommandButtons`, `TextBoxes`, and `Labels`."

9. **D**. To prevent the user from being able to give focus to a control under any circumstances, you may set the control's `Enabled` property to `False`. All the other options will only affect the user's navigation with the Tab key. The user could still use the mouse as long as the `Enabled` property were `True`. See "Enabled" under the section "Important Common Properties of `CommandButtons`, `TextBoxes`, and `Labels`."

10. **B**. The line `Text1 = Command1.Caption` would set the `TextBox`'s `Text` property to the `CommandButton`'s `Caption`. `Text1 = Command1` would not work because the `CommandButton`'s `Default` property is the `Value` property. `Text1.Text = Command1` would not work for the same reason. Finally `Text1.Text = CStr(Command1)` would set `Text1.Text` to the string "True" or "False" because, once again, it would convert the `Default` property of `Command1` (the `Value` property, which is Boolean) into a string. See "Referring to a Property Within Code."

11. **A**. When a `TextBox`'s `Enabled` property is `False`, the text in the box will be grayed and the user won't be able to set focus to the `TextBox`. Option D describes the behavior of a `TextBox` when the `Locked` property is `True`; Option B describes the behavior if the `ForeColor` property were gray and `Locked` were `True`, and C would be the situation if `ForeColor` were gray, `Enabled` were `True`, and `Locked` were `False`. See "Enabled" under the section, "Important Common Properties of `CommandButtons`, `TextBoxes`, and `Labels`" and also see "Important Properties of the `TextBox` Control."

12. **C**. An access key for a `TextBox` control can be provided in an accompanying `Label` control, provided the `Label` immediately precedes the `TextBox` in the Tab order. As for the other answers: `TextBoxes` have no `Caption` property, their `Text` property is constantly changed by the user, and they have no `Label` property. See "Assigning an Access Key to a `TextBox` Control Through a `Label`'s Caption."

OBJECTIVES

This chapter helps you prepare for the exam by covering the following objectives:

Add an ActiveX control to the ToolBox (70-175 and 70-176).

- See the objective explanation for "Create data input forms and dialog boxes."

Create data input forms and dialog boxes (70-175 and 70-176).

- **Display and manipulate data by using custom controls. Controls include `ListView`, `ImageList`, `ToolBar`, and `StatusBar`.**
- **Create an application that adds and deletes controls at runtime.**
- **Use the Controls Collection to manipulate controls at runtime.**
- **Use the Forms Collection to manipulate forms at runtime.**

- The exam objectives for this chapter broaden the focus of the objective of the previous chapter by adding more elements of user interface programming:
- The first objective (Add an ActiveX control to the ToolBox) and the first subobjective (Display and manipulate data by using custom controls) of the second objective (Create data input forms and dialog boxes) focus on several controls that are not found in the standard VB toolbox. You must know how to add such custom controls to the VB toolbox so that you can program with them, and then, of course, you must know something about each control's object model and design- and runtime behaviors. We discuss each of the controls listed in the second objective in this chapter.
- The remaining subobjectives deal with broader issues of programmatic manipulation of VB objects in the visual interface. These objectives apply to any type of control in any VB application.

CHAPTER 4

Creating Data Input Forms and Dialog Boxes

OUTLINE

Adding an ActiveX Control to the ToolBox **128**

Using ActiveX Controls to Create Data Input Forms and Dialog Boxes **129**

Using the `ImageList` Control 129
Using the `TreeView` Control 134
Using the `ListView` Control 139
Using the `ToolBar` Control 147
Using the `StatusBar` Control 153
Techniques for Adding and Deleting Controls Dynamically 160
More on Creating Data Input Forms and Dialog Boxes 172
Using the Forms Collection 179

Chapter Summary **185**

STUDY STRATEGIES

- For the first objective (Display and manipulate data by using custom controls), you should perform Exercises 1 through 4 at the end of this chapter.
- For the second objective (Create an application that adds and deletes controls at runtime), see Exercise 5 at the end of this chapter. Programmers familiar with earlier versions of VB should *not* depend solely on their knowledge of previous versions for this objective, since VB6 now allows you to add controls dynamically without having a design time control array to provide initial templates for the new controls.
- For the third objective (Use the Controls Collection to manipulate controls at runtime), see Exercise 6 at the end of this chapter.
- For the fourth objective (Use the Forms Collection to manipulate forms at runtime), see Exercise 7 at the end of this chapter.

INTRODUCTION

This chapter extends the ideas introduced in Chapter 3, "Implementing Navigational Design," and describes further steps you can take to provide the user with a functioning data input form or dialog box.

You may include several standard 32-bit ActiveX controls on your form to enhance your Visual Basic application's user interface. These controls organize data and provide different ways of presenting information to the user, give you additional means of displaying information about the environment, and also provide the means for you—as a developer—to manipulate data and controls.

Here is a brief description of what the ActiveX controls discussed in this chapter do:

- The `ImageList` control gives you a means of loading graphics files, such as icons and bitmaps, into your application for use with other controls.
- You can use the `ListView` control to organize data in lists.
- The `ToolBar` control lets you quickly build toolbars in your application, giving users an alternative to the menu for performing actions.
- You can add the `StatusBar` to a form to present information about the environment to the user through text messages and progress bars.

We then discuss control arrays that allow your application to dynamically create and destroy controls as the application runs.

Next we examine the related concept of a form's Controls Collection that you can use to manipulate controls without referencing each by name.

Turning our attention to the manipulation of forms, we first discuss the different techniques for managing forms programmatically.

Finally we talk about how to manipulate the loaded forms in an application without referencing each form by name. As you might expect from the earlier description of the Controls Collection, we will use the Forms Collection.

This chapter examines the following topics:

- Adding an ActiveX control to the ToolBox
- Using the `ImageList` control
- Using the `TreeView` control
- Using the `ListView` control
- Using the `ToolBar` control
- Using the `StatusBar` control
- Adding and deleting controls dynamically
- Using the Controls Collection
- Manipulating forms in code to provide data input forms and modal dialogs
- Using the Forms Collection

ADDING AN ACTIVEX CONTROL TO THE TOOLBOX

▶ Add an ActiveX control to the ToolBox.

The controls discussed in this chapter are available through the Microsoft Windows Common Controls Library 6.0 (comctl32.ocx) that you can select from the Project Components dialog box, as shown in Figure 4.1.

Generally custom controls that you will add to a VB application are implemented in files with the extension .ocx.

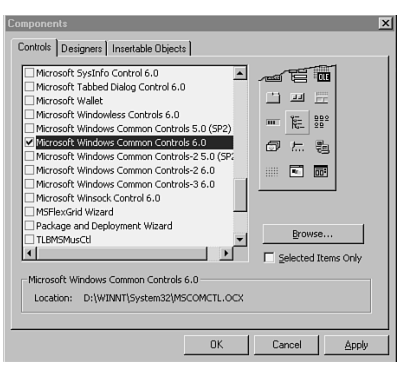


FIGURE 4.1
Adding the Common Controls to a project.

USING ACTIVEX CONTROLS TO CREATE DATA INPUT FORMS AND DIALOG BOXES

▶ Create data input forms and dialog boxes.

The various ActiveX controls and their uses are discussed in this section.

Using the `ImageList` Control

The `ImageList` is a control that enables you to store graphic images in an application. Other controls can then use the `ImageList` as a central repository for the images that they will use. Both bitmaps (*.bmp files) and icons (*.ico files) can be stored in the `ImageList` control. At runtime the `ImageList` is invisible, just like a `Timer` or a `CommonDialog` control, so you can place it anywhere on a form without interfering with the user interface.

After an `ImageList` control has been added to a form, images can be added to the control at design time through the Custom Properties dialog box. The first tab of the dialog box, as shown in Figure 4.2, enables you to change general properties of the `ImageList`. On this tab, you can select the style and size of a graphic that will be included in the `ImageList`. Three of the resolutions, 16×16, 32×32, and 48×48, refer to the resolution of icon files. The Custom option lets you include bitmap images in the `ImageList` as well as icon images. You do not need to choose a resolution, height, and width for the ImageList. As soon as you add an image to the control, Visual Basic automatically determines the properties for you. After you have placed an image in the `ImageList`, you cannot change the `resolution` property. It is locked for as long as there is an image in the `ImageList` control.

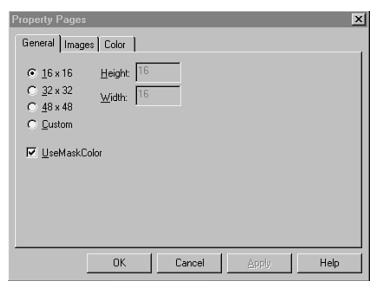


FIGURE 4.2
The General tab of the Property Pages dialog box for the `ImageList` control.

The list of images contained in the `ImageList` control can be managed through the Images tab of the Property Pages dialog box, as shown in Figure 4.3. This tab enables you to add and remove images from the control as well as set additional information about each image. The Index of each image is assigned by Visual Basic. The Index starts at 1 for the first image and increments by 1 for each additional image.

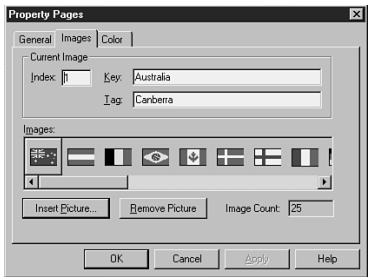


FIGURE 4.3
The Images tab of the Property Pages dialog box for the `ImageList` control.

You can use the `Key` property to refer to an image in the `ImageList`'s collection of images. The `Key` property is a string value that you can use in place of the `Index` property to refer to an image in the list.

The easiest way to add images to the `ImageList` is to use the Images tab at design time. Just click on the Insert Picture button, and you can browse for the *.ico and *.bmp files that you want to add to the control. After you have added images to the `ImageList` control, they are available for your application to use in other controls. You can load images into `PictureBoxes` and `Image` controls from the `ImageList`. The images can also be used by the other controls discussed later in this chapter, such as the `ToolBar` and `ListView` controls.

Although you can add all the images you may need in your application to the `ImageList` at design time, there are times when you will have to manipulate these images at runtime. This can be done through the `ListImages` Collection of the `ImageList` control. The `ListImages` Collection contains `ListImage` objects.

`ListImage` Object and `ListImages` Collection

All the images contained in the `ImageList` control are stored in the `ListImages` Collection. Each icon and bitmap in the collection is a separate `ListImage` object. You can refer to each image in the list by its index:

```
ImageList1.ListImages(1).Picture
```

or, if a key (for example, "Smiley") were assigned to a particular `ListImage` object, you could refer to it by its key value:

```
ImageList1.ListImages("Smiley").Picture
```

You can use the `ListImages` Collection to loop through all the images in the list. If you wanted to display all the stored images in a `PictureBox`, one after another, you could use the code of Listing 4.1.

LISTING 4.1

LOOPING THROUGH THE IMAGES IN AN IMAGELIST CONTROL

```
Dim picImage as ListImage

For Each picImage in ImageListImageList1.ListImages
     Picture1.Picture = picImage.Picture
Next
```

If you knew the image you wanted to move to a `PictureBox`, you could just code this:

```
Picture1.Picture = ImageList1.ListImages(3).Picture
```

or this:

```
Picture1.Picture = ImageList1.ListImages("Smiley").Picture
```

In the first example, the third image in the `ImageList` control would be loaded into `PictureBox Picture1`, and in the second example, the image whose key is FirstImage would be loaded.

`Add` and `Remove` Methods

You can use the `Add` method of the `ListImages` Collection to add images to the `ImageList` at runtime. The syntax for `Add` is as follows:

```
ImageList1.ListImages.Add([index], [key], picture)
```

Here `ImageList1` is the name of the `ImageList` control on your form. `Index` and `Key` are optional parameters. If `Index` is specified, it will be the location at which the image is loaded into the `ListImages` Collection. If `Index` is omitted, the new image will be inserted at the end of the collection. `Key` is a string value that you can use to refer to an image in the list without knowing its index value. For example if an image were added as follows:

```
ImageList1.ListImages.Add(, "folder icon",
➥LoadPicture("folder.ico"))
```

you might not know the index value of the new image, but you could still refer to it in this way:

```
Picture1.Picture = ImageList1.ListImages("folder
➥icon").Picture
```

Using the `ListImages` key makes your code more readable than if you refer to images with the `Index` property.

You can remove a `ListImage` from the `ListImages` Collection with the `Remove` method. You can use `Remove` either by specifying the index of the image:

```
ImageList1.ListImages.Remove 1
```

or by providing the image's key value:

```
ImageList1.ListImages.Remove "key value"
```

`Draw` Method

You can use the `Draw` method of the `ListImage` object to draw an image on another object. As discussed earlier, an image from the `ImageList` control can be loaded into a `PictureBox` by setting the `Picture` property:

```
Picture1.Picture = ImageList1.ListImages(1).Picture
```

You can also use the `Draw` method to accomplish the same thing. With `Draw`, however, you have some additional options. The syntax for `Draw` is as follows:

```
ImageList1.ListImages(Index).Draw (HDC, x,y, style)
```

where `Index` identifies the image to be drawn. The `Key` value can be used in place of the `Index` as well. `HDC` is the device context ID of the destination device. If you were to draw an image onto a form, for example, you could specify Form1.HDC for the `HDC` property. This tells Windows where to put an image. The optional parameters, `x` and `y`, identify the coordinates within the destination at which the image will be drawn. The `Style` property can take one of the following four values:

- **`imlNormal (0)`** The image will appear normally—with style = 1.
- **`imlTransparent (1)`** The part of the image will appear to be transparent. Transparency is determined by the `MaskColor` property (set on the Color tab of the Custom Properties dialog box).
- **`imlSelected (2)`** The image will be dithered with the system highlight color.
- **`imlFocus (3)`** The image appears as if it has focus.

`Overlay` Method

You can use the `Overlay` method of the `ImageList` control to return a combination of two images from the `ListImages` Collection. The method is a function and returns the result as a picture. The syntax for the `Overlay` method is as follows:

```
Set Object = ImageList1.Overlay(index1,index2)
```

where `index1` and `index2` refer to two images in the `ListImages` Collection and Object refers to an object such as an `Image` or `PictureBox`. Either the Index or the Key for a `ListImage` can be used.

The resulting picture can be used as any other picture object in Visual Basic. It can be placed on a destination object, such as a `Form` or `PictureBox`, or can even be loaded into another `ImageList` control. The following code

```
Form1.Picture = ImageList1 (1, 2)
```

combines the first and second images from the `ImageList1.ListImages` Collection and places the resulting picture on `Form1`.

`ImageHeight` and `ImageWidth` Properties

The `ImageHeight` and `ImageWidth` properties of the `ImageList` control identify the height and width in ***pixels*** (not in twips!) of images belonging to the `ListImages` Collection. These properties are read/write at design time (from the Custom Properties window) and at runtime. The `Height` and `Width` properties identify the size of an image in pixels.

Note that after the first image is added to the `ListImages`, all other images must be the same height and width; otherwise an error will occur. If you need to include different-sized icons in an application, you can use multiple `ImageList` controls, one for each size needed.

`ListImages` Property

The `ListImages` property returns a reference to the collection of `ListImage` objects (`ListImages` Collection) contained within an `ImageList` control.

`MaskColor` and `UseMaskColor` Properties

`MaskColor` is a read/write property used to identify the color that will be used to create masks for the `ImageList` control. The `MaskColor` can be set at design time on the Color tab of the Custom Properties dialog box for the `ImageList` control. It can also be set and read at runtime as follows:

```
ImageList1.MaskColor = vbBlack
```

You can set the `MaskColor` property by using the Visual Basic color constants, the `QBColor` function, or by using the `RGB` function. The `MaskColor` is used with the `Draw` and the `Overlay` methods to determine the parts of an image that will be transparent.

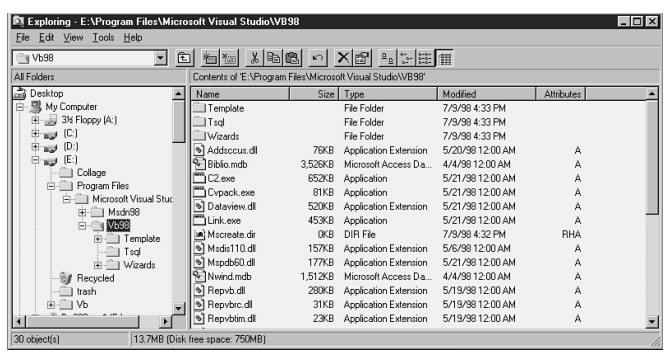
`UseMaskColor` determines whether the `MaskColor` property will be used as part of a `Draw` or `Overlay`. It takes either a `True` or a `False` value and is also available at either design time or runtime.

Using the `TreeView` Control

The purpose of a `TreeView` control is to display information in a hierarchy. A `TreeView` is made up of *nodes* that are related to each other in some way. Users can look at information, or objects, presented in a `TreeView` control and quickly determine how those objects are bound together.

Figure 4.4 shows a good example of `TreeView` in the Windows Explorer. The left side of Explorer shows information about drives in a hierarchical layout. Explorer starts its tree with a node called Desktop. From there you can see several nodes indented below the Desktop node. The indentation and the lines connecting nodes show that the My Computer node is a child of the Desktop node. My Computer also has *children*—the A: drive, C: drive, and so on. Children of the same parent are often referred to as *siblings*. The A: drive and B: drive, for example, both have My Computer as a parent and are called siblings.

FIGURE 4.4
Windows Explorer as an example of a `TreeView`.



The `TreeView` control is available as part of the Common Controls component in Visual Basic. You can use the `TreeView` anytime that you need to display data to a user as a hierarchy.

`Node` Object and `Nodes` Collection

A `TreeView`'s information is contained in a Nodes Collection of `Node` objects. Just as image information is stored in the `ListImages` Collection for the `ImageList` control, node information is stored in the `Nodes` Collection of the `TreeView` control. You can get to the information in a specific node in the tree by referring to that node by index, as follows:

```
TreeView1.Nodes(index)
```

where "index" is an integer identifying a node in the collection, or by referring to the node with a key value, as follows:

```
TreeView1.Nodes("key")
```

where "key" is the string value of the node's key property. The `Node` object represents a single node from the `Nodes` Collection:

```
Dim objNode as Node
Set objNode = TreeView1.Nodes(1)
```

After the above code executes, `objNode` will have the same properties as the node identified by `TreeView1.Nodes(1)`.

Add and Remove Methods

Nodes can be added to the `TreeView` control by using the `Add` method of the `Nodes` Collection. The syntax for `Add` is as follows:

```
TreeView1.Nodes.Add(relative, relationship, key, text,
➥image, selectedimage)
```

All the arguments for the `Add` method are optional.

The `Relative` argument gives VB the Key or Index of an existing node, and the `Relationship` parameter tells VB where to place the new node in relation to the relative node.

If the `Relative` and `relationship` arguments are not specified, the new node will be placed at the top level in the tree after all other existing nodes at that level.

The values for the `relationship` argument are as follows:

- **`tvwFirst (0)`** The new node is placed at the same level in the tree as the "relative." It will be added as the first node of that level.

WARNING

Key Values for `TreeView` Nodes
Throughout Visual Basic, you will find many collections that can be referenced by key, which is a string value. In most of these collections, if you want to use a number as the key, you can just convert the number to a string by using the `Str$()` function or by enclosing the number in double quotation marks. You should be aware, however, that `Nodes` Collection of the `TreeView` is an exception to this. Converting a number to a string and attempting to use that string as the key to a `TreeView Node` object will generate a runtime error.

- **`tvwLast (1)`** The new node is placed at the same level in the tree as the "relative" but will be added after the last existing node at the same level as "relative."
- **`tvwNext (2)`** The new node will be placed at the same level in the tree as the "relative," immediately following that node.
- **`tvwPrevious (3)`** The new node will be placed at the same level in the tree as the "relative," immediately preceding that node.
- **`tvwChild (4)`** The new node will be a child of the "relative" node.

The `Key` property identifies the new node in the tree. If provided as an argument, it must be a unique string, not used as a key by any other node in the tree. The key is used to retrieve or to find this node when the index is not known.

The last three arguments of the `Add` method define the appearance of the new node. The text that appears next to a node in the `TreeView` is specified by the `Text` argument, a string value. If you want to have icons appear in the `TreeView` alongside the `Text`, you must first have an `ImageList` control on your form. When you set up a `TreeView` and define its properties through the Property Pages dialog box, you can bind an `ImageList` to the `TreeView`. Figure 4.5 shows an example of this.

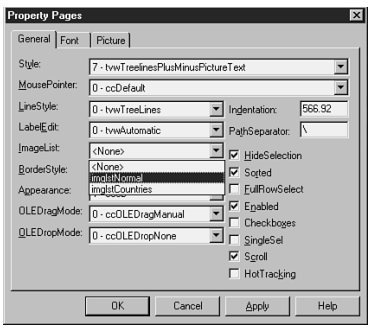


FIGURE 4.5
Binding an `ImageList` to a `TreeView` control.

To include an icon with the node text, you can use the `Image` argument. This argument has an integer value that corresponds to the index of an image in the bound `ImageList` control. The `ImageList` has to be set up first so that the index values are available for use in the `TreeView` control. If you want a node to have a different icon displayed when that node is selected by the user, you can specify a second icon with the `SelectedImage` argument. This is also an integer argument identifying an image in the same `ImageList`.

`EnsureVisible` and `GetVisibleCount` Methods

If at some point during the execution of your program, you need to make sure that a node in the `TreeView` is visible, you can use the `EnsureVisible` method of a specific node:

```
TreeView1.Nodes(23).EnsureVisible
```

This makes the node with an index of 23 visible to the user even if the node is several levels deep in a tree that is completely collapsed. The `EnsureVisible` property expands the tree to make the node visible.

To get the number of nodes visible at any one time, you can use the `GetVisibleCount` method of the `TreeView` control. By using

```
TreeView1.GetVisibleCount
```

you will get a count of nodes visible within the `TreeView`. The count will only include those nodes that are visible, including any nodes partially visible at the bottom of the control. The count does not include any nodes expanded below the bottom edge of the `TreeView` control.

`TreeView` Properties

Numerous properties for the `TreeView` control and for `Node` objects define the appearance of the tree and give access to nodes within the tree. You can use many of these properties to navigate through a `TreeView`, as follows:

- **`Child`** Returns a reference to the first child of a node. The `Child` property can be used to set a reference to a node:

  ```
  Dim objNode as Node
  objNode = TreeView1.Nodes(1).Child
  ```

  This sets `objNode` equal to the first child of the node with index 1. Operations can also be performed directly on the reference to the child:

  ```
  TreeView1.Nodes(1).Child.Text = "This is the first child."
  ```

  This changes the text for the first child node of node 1.

- **`FirstSibling`** Returns a reference to the first sibling of the specified node. `FirstSibling` is a node at the same level as the specified node.

- **`LastSibling`** Returns a reference to the last sibling of the specified node. `LastSibling` is a node at the same level as the specified node.

- **`Parent`** Returns a reference to the parent of the specified node.

- **`Next`** Identifies the node immediately following the specified node in the hierarchy.

- **`Previous`** Identifies the node immediately preceding the specified node in the hierarchy.

- **`Root`** Provides the root node, or top-level node, in the tree for the specified node.
- **`SelectedItem`** Returns a reference to the node currently selected in the `TreeView` control.
- **`Nodes`** Returns a reference to the entire `Nodes` Collection for the `TreeView`.

You can also use the following additional properties to control the behavior and appearance of the `TreeView`:

- **`Children`** Returns the total number of child nodes for a given node.
- **`Selected`** A `True` or `False` value indicating whether a particular node is selected.
- **`Expanded`** A `True` or `False` value indicating whether a particular node is expanded (that is, its child nodes are visible).
- **`FullPath`** Returns a string value depicting the entire path from the root to the current node. The full path is made up of the concatenated text values of all the nodes from the root, separated by the character specified by `PathSeparator` property.
- **`PathSeparator`** Identifies the character used as a separator in the `FullPath` property.
- **`LineStyle`** Determines the appearance of the lines that connect nodes in a tree. `LineStyle` can have two values 0 (tvwTreeLines) and 1 (tvwRootLines). If `LineStyle` is 0, there will be lines connecting parents to children and children to each other. If `LineStyle` is 1, there will also be lines connecting the root nodes.
- **`Sorted`** A `True` or `False` value for the `TreeView`. If `Sorted` is `True`, the root nodes will be sorted alphabetically by the `Text` property of each node. Child nodes will also be sorted alphabetically within each parent. If `Sorted` is `False`, the nodes in the `TreeView` will not be sorted.

  When the `Sorted` property is set to `True`, the nodes that already exist in the `TreeView` will be sorted. If any additional nodes are added, they will not be sorted into the existing nodes.

The `Sorted` property will have to be set to `True` again for these new nodes to appear in sorted order.

`TreeView` Events

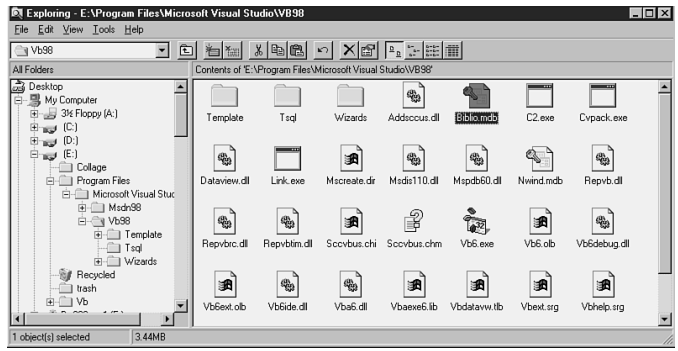
You can use several events of the `TreeView` control to code for actions taken by the user or to handle actions caused by code execution. In addition to standard control events such as `Click` and `DblClick`, the `TreeView` control has these following additional events:

- **`Collapse`** Generated whenever a node in a `TreeView` control is collapsed. This event occurs in one of three ways: when the `Expanded` property of a node is set to `False`; when the user double-clicks on an expanded node; or when the user clicks on the +/- image for a node to collapse that node. The node that was collapsed is passed in as an argument to the event.
- **`Expand`** Generated when a node in a `TreeView` is expanded. Like the `Collapse` event, `Expand` occurs in one of three instances: when the user double-clicks on a node that has children to expand that node; when the user clicks on the +/- image to expand a node; or when the `Expanded` property for the node is set to `True`. `Expand` also has one argument, the node object that was expanded.
- **`NodeClick`** Occurs when a `Node` object is clicked by the user. The node that was clicked is passed in as an argument to the event. If the user clicks anywhere on the `TreeView`, other than on a node, the `Click` event for the `TreeView` control is fired instead.

Using the `ListView` Control

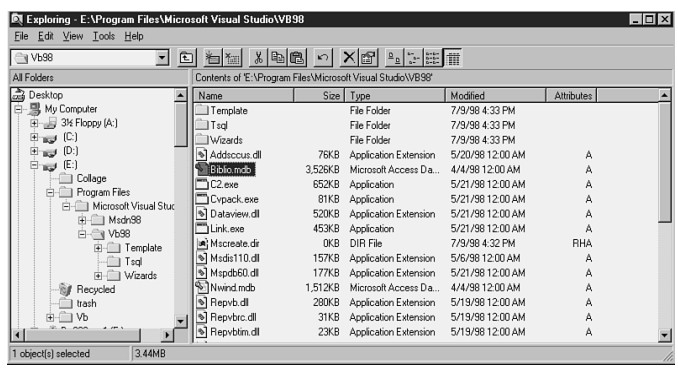
The `ListView` control displays lists of information to a user. As with the `TreeView` control, Windows Explorer provides an example of the `ListView` control. The left side of Windows Explorer contains a tree of all the directories on a drive. The right side contains a list of items within a directory. To get an idea of the ways in which a `ListView` control can be used, you just need to look at the way a list of files appears and behaves in Windows Explorer (see Figure 4.6).

FIGURE 4.6▶
Example of a `ListView` in Windows Explorer showing large icons.



Objects can be displayed in one of four ways with the `ListView`. You can use either large or small icons to represent each item in the list, along with accompanying text information, where multiple items can appear on a single row. You can also display items in a list with one item per line. Finally you can show items as a columnar report in the `ListView` control with one item appearing on each line and sub-item information displaying in additional columns in the control, as shown in Figure 4.7.

FIGURE 4.7▶
Example of a `ListView` in Windows Explorer showing the report style.



As you learn about the `ListView`, you will see similarities to the `TreeView` control. The behavior and appearance of the two controls can be manipulated in many of the same ways. The major difference that you will notice is that objects in a `ListView` are not related to each other, as are objects in a `TreeView`.

ListItem Object and ListItems Collection

Each item in the ListView is called a ListItem object. If you look at Figure 4.6, for example, you will see a list of files in the Visual Basic directory. Biblio.mdb, Readme.hlp, and each of the other files in the right side of the Windows Explorer is a ListItem.

The ListView control organizes all the ListItem objects into a single collection called ListItems. This is similar to a TreeView, which organizes each Node object in a tree into the Nodes Collection. The ListItems Collection can be used to cycle through all the objects in the control for processing, just as with any other collection (see Listing 4.2).

LISTING 4.2

PROCESSING THE ITEMS CONTAINED IN A LISTVIEW CONTROL

```
Dim objItem as ListItem
For Each objItem in ListView1.ListItems
   ' do some processing÷
Next
```

Index and Key Properties

You can refer to Items in a ListView by either the Index property or the Key property. The Index property is an integer expression typically generated by Visual Basic when a ListItem is added to the ListView. You can refer to a specific item by number:

```
Msgbox ListView1.ListItems(1).Text
```

Alternatively you can loop through the collection of ListItems, referring to each by number, as illustrated in Listing 4.3.

LISTING 4.3

LOOPING THROUGH A LISTVIEW CONTROL'S LISTITEMS BY NUMBER

```
For I = 1 To ListView1.ListItems.Count
        Msgbox ListView1.ListItems(I).Text
Next I
```

Unlike the `Index` in the `TreeView`, you have some control over the value of the `Index` property in the `ListView` control. Specifying an `Index` number is discussed later in this section.

A more convenient way to reference an item in the list is by `Key`. The `Key` property is a string expression—assigned by you as the developer (or by the user if you desire)—that can also be used to access an item in the list. The `Key` property is included as part of the `Add` method (discussed later) when an item is inserted into the list. As a developer you usually know the value for the `Key` and can access a node directly. It is easier to refer to the node you want by using a meaningful text string than by using the `Index` property that is determined by Visual Basic.

Unlike the `TreeView` you can store numbers in the `Key` property of each `ListItem` if necessary. You just have to convert the number to a string by using the `Str$()` function, and the string value will be accepted as the `Key` by Visual Basic.

`View` Property

The overall appearance of a `ListView` is determined by the `View` property. The `View` property can have one of the following four values:

- **`lvwIcon (0)`** Display item text along with the regular icon with one or more `ListItems` per line.
- **`lvwSmallIcon (1)`** Display item text along with the small icon with one or more `ListItems` per line.
- **`lvwList (2)`** Display the small icon with the text to the right of the icon. One `ListItem` will appear per line.
- **`lvwReport (3)`** Display the small icon with the text to the right of the icon and sub-item information to the right of the text displayed in columns. One `ListItem` will appear per line.

`Add` and `Remove` Methods

`ListItems` are inserted into a `ListView` by using the `Add` method. The `Add` method has the following syntax:

```
ListView1.ListItems.Add(index, key, text, icon, smallIcon)
```

All the arguments for the `Add` method are optional.

The `Index` argument is an integer value that you can use to specify the position of the new item being added to the list. If the `Index` argument is omitted, Visual Basic places the new item at the end of the list.

The `Key` property is a unique string that you can use to identify an item in the list instead of using the `Index` to that object.

The last three arguments—`Text`, `Icon`, and `SmallIcon`—determine the appearance of the new item in the `ListView`.

If the `Text` argument is given, that text will appear with the item in the `ListView`.

The `Icon` and `SmallIcon` arguments are integers referring to an icon in an `ImageList` control. As with the `TreeView`, the `ListView` needs at least one `ImageList` control on the form for icons to be displayed. References to `ImageLists` are set through the Property Pages dialog box for the `ListView`, as shown in Figure 4.8. You can set references to two different `ImageLists` for a `ListView`: one reference for regular icons (when the `ListView`'s View property is lvwIcon) and a second list for small icons (when the View property is set to any other value).

Because an `ImageList` control can only contain images of a single size, two `ImageLists` are required if you will be using both regular and small icons. Typically regular icons will be 32×32, and small icons will be 16×16.

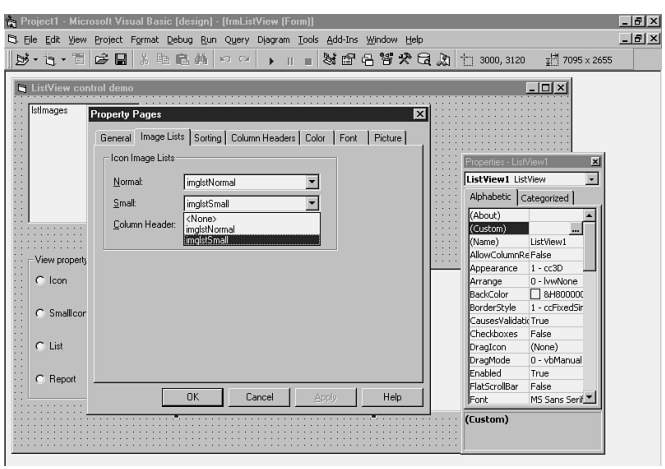


FIGURE 4.8
Setting an `ImageList` reference for a `ListView` control.

Icon and SmallIcon Properties

After ListItems have been added to a ListView, their references to icons in an ImageList can be read or set through the Icon and SmallIcon properties. These properties have integer values that correspond to the index value of images in the ImageLists to which the ListView is bound. If you create a form with ListView1, ImageList1, and ImageList2 controls, for example, and you use ImageList1 for regular icons and ImageList2 for small icons, the following:

```
x = ListView1.ListItems(1).Icon
```

returns an integer that corresponds to an index value of an image in ImageList1 while:

```
x = ListView1.ListItems(1).SmallIcon
```

gives you the index of an image in ImageList2.

FindItem Method

After items have been added to a list, you may have a need to find one or more of those items. The FindItem method of the ListView control enables you to search through the items and return the desired ListItem object. FindItem has the following syntax:

```
ListView1.FindItem (string, value, index, match)
```

The string argument, which is required, specifies the string for which you are searching in the list. The value argument tells Visual Basic how to search for the string. Value can be one of the following:

- **lvwText (0)** Search for the String argument in the Text property of the ListItems.
- **lvwSubItem (1)** Search for the String argument in the sub-items of the ListItems.
- **lvwTag (2)** Search for the String argument in the Tag property of the ListItems.

The Index argument can be used to indicate the start position for the search. This argument can have either an integer or a string value. If the value is an integer, Visual Basic starts the search at the ListItem with an Index of that value. If the argument is a string, the search begins at the item having the same Key value as the argument value. If the Index argument is not specified, the search starts at the first item in the list.

The final argument, `match`, determines how Visual Basic will select a `ListItem` that matches the string argument. `Match` can have the following two values:

- **`lvwWholeWord(0)`** A match occurs if the `String` argument matches the whole word that starts the `Text` property of the item.
- **`lvwPartial (1)`** A match occurs if the `String` argument matches the beginning of the `Text` property regardless of whether it is the whole word.

`Arrange` Property

The `Arrange` property of the `ListView` determines how items are arranged within a list. Arrange can have one of the following three values:

- **`lvwNone (0)`** Items are not arranged within the `ListView`.
- **`lvwAutoLeft (1)`** Items are arranged along the left side of the `ListView`.
- **`lvwAutoTop (2)`** Items are arranged along the top border of the `ListView`.

The `ListItem`'s appearance in a `ListView` is also determined by the sorting properties.

`Sorted`, `SortKey`, and `SortOrder` Properties

Three properties—`Sorted`, `SortKey`, and `SortOrder`—determine the order of `ListItems` in a `ListView` control.

The `Sorted` property can have one of two values, `True` or `False`. If `Sorted` is False, the items in the `ListView` are not sorted. If the `Sorted` property is set to `True`, the `ListItems` will be sorted with an order that depends on the other two properties: `SortKey` and `SortOrder`. The values of `SortOrder` and `SortKey` are ignored unless the `Sorted` property is set to `True`.

`SortOrder` specifies whether the `ListItems` appear in ascending (`lvwAscending`) or descending (`lvwDescending`) order. Ascending order is the default for `SortOrder`. `SortKey` identifies on what the items in the list will be sorted. By default the items in the list will be sorted by the `Text` property of the `ListItems`.

If your `ListView` is in a report format with multiple columns, you can sort by using the text value of any of the columns by setting the `SortKey` property to the desired column. If `SortKey` is 0, the list will be sorted by the `ListItems().Text` property. If `SortKey` is greater than 0, the sort will take place using one of the additional report columns in the control (described in the following section).

`ColumnHeader` Object and `ColumnHeaders` Collection

If you want to use a `ListView` with a report format that has multiple columns for each `ListItem`, you must work with the `ColumnHeaders` Collection. `ColumnHeaders` is a collection of `ColumnHeader` objects. Each `ColumnHeader` object identifies one column in a `ListView`. The `ColumnHeaders` Collection always contains at least one column for the `ListItem` itself. Additional columns can be added using the `ColumnHeaders.Add` method or removed with the `ColumnHeaders.Remove` method.

If you look back at the example of the Windows Explorer as an `ImageList`, you will see that it uses four columns. The first column—the icon and the filename—is the `ListItem`. This would be the first entry in the `ColumnHeaders` Collection. Additional columns are included for the file size, type, and modification date.

`SubItems` Property

Once additional columns have been added to a `ListView`, those columns can be accessed through the `SubItems` property of a `ListItem`. The text that appears in a column for a particular `ListItem` can be read or set using:

```
Msgbox ListView1.ListItems(1).SubItems(2)
```

For this example the above code will display the text that appears in the third column, `SubItems(2)`, of the first `ListItem`, `ListItems(1)`.

`ItemClick` Event

Most code associated with a `ListView` control appears in either the `ItemClick` event or the `ColumnClick` event. The `ItemClick` event occurs when the user clicks on a `ListItem` within the `ListView` control. The `ListItem` that was clicked will be passed into the event as an argument.

The `ItemClick` event occurs only when an item in the list is clicked. If the user clicks anywhere in the `ListView` control other than on an item, the regular `Click` event is fired.

`ColumnClick` Event

The `ColumnClick` event is fired when the user clicks on the column header of the `ListView`. The `ColumnHeader` object that was clicked is passed into the event as an argument.

The code that is typically placed in the `ColumnClick` event is the code to sort the `ListItems` by that column. This is the normal behavior expected by users in the Windows environment.

The `ColumnHeaders` is one-based, meaning that the first column in the `ColumnHeaders` collection has an index of 1. The `SortKey` property, however, uses 0 as the index for the first key. Therefore if you want to match up a `ColumnHeader` with its corresponding `SortKey`, you must subtract 1 from the `ColumnHeader` index, as in Listing 4.4.

LISTING 4.4

MATCHING A COLUMNHEADER WITH ITS CORRESPONDING SORTKEY

```
Private Sub ListView1_ColumnClick _
    (ByVal ColumnHeader As ComctlLib.ColumnHeader)
'Change the SortKey of the ListView
'to correspond to the SubItem in the just-clicked ColumnHeader
    'MsgBox ColumnHeader.Index
    ListView1.SortKey = ColumnHeader.Index - 1
    ListView1.Sorted = True
End Sub
```

Using the `ToolBar` Control

In earlier 16-bit releases of Visual Basic, it was difficult to implement a toolbar for your users. You had to place a `PictureBox` control on a form and then add `CommandButton` controls to the `PictureBox` to simulate the toolbar. Starting with the 32-bit version of Visual Basic 4.0 and continuing with all subsequent versions of VB, you have the `ToolBar` ActiveX control that you can add to your forms to easily implement toolbar functionality for your users.

A toolbar is becoming a standard feature of Windows applications. Toolbars provide functionality to a user through an easily accessible, graphical interface. For common functions the user does not need to navigate through a menu or remember shortcut keys to use an application. Applications can expose their most common features through buttons on a toolbar.

Setting Custom Properties

The `ToolBar` control is available through the Windows Common Controls, along with the `TreeView`, `ListView`, and `ImageList`. After you have drawn a toolbar on a form, you will usually start by setting properties through the Property Pages dialog box for the control.

On the first tab of the Property Pages dialog box, as shown in Figure 4.9, one of the options you will set most often is the `ImageList`. Like the `TreeView` and the `ListView` controls, the toolbar gets images from an `ImageList` control. If you are building a toolbar that will have graphics on its buttons, you will first need to add an `ImageList` control to the form and then bind that `ImageList` to the toolbar through the Property Pages dialog box.

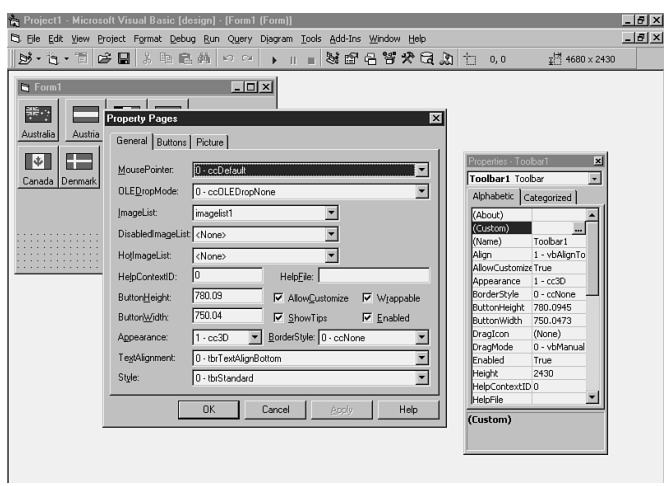


FIGURE 4.9
General tab of the Custom Properties of a toolbar.

Several other properties are unique to the toolbar and can be set on the General tab. The `ButtonHeight` and `ButtonWidth` properties determine the size of buttons that appear on the toolbar. All buttons will have the same height and width. The number of buttons that can appear on a toolbar is determined by the size of the buttons and the width of the window. If you want the toolbar and buttons to wrap when the window is resized, you can set the `Wrappable` property of the toolbar to `True`.

If you want to add ToolTips to your `ToolBar` control, you must set the `ShowTips` property to `True`. The actual tips that appear are tied to each button. You can allow the user to customize the toolbar by setting the `AllowCustomize` button to `True`. These properties are discussed later in the section "Customizing Toolbars."

`Button` Object and `Buttons` Collection

Command buttons on a toolbar are called `Button` objects, and all the `Button` objects are stored in the `Buttons` Collection. This is similar to the way that the `TreeView Node` objects are contained in a `Nodes` Collection. Each `Button` object has many properties that control its appearance and functionality. These properties can be set either at design time through the Buttons tab of the Property Pages dialog box (see Figure 4.10) or at runtime.

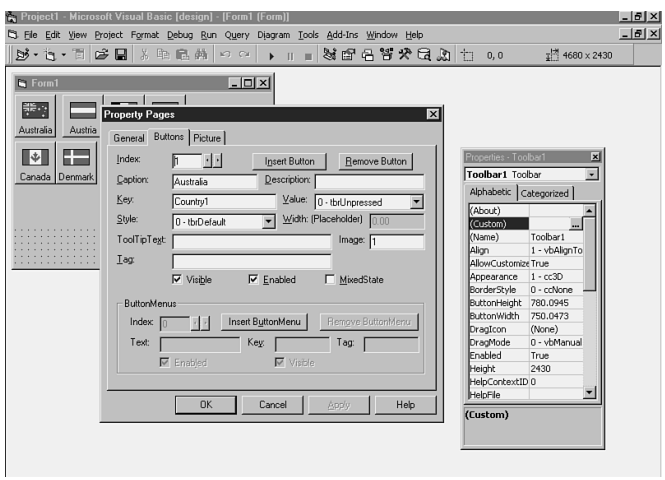


FIGURE 4.10
Buttons tab of the Custom Properties of a toolbar.

WARNING

Key Property of a Toolbar's `Button` Object The `Key` property of a `Button` object cannot contain a numeric expression even if you convert the number using the `Str$()` function first. If you try to use a number converted to a string as the key to a `Button` object, you will receive a runtime error indicating an invalid key value.

As with the other controls discussed in this chapter, individual `Button` objects can be referenced by either the `Index` property or the `Key` property. The `Index` property has an integer value that refers to a button's position on a toolbar. The first button on a toolbar will have an `index` value of 1. The `Key` property is a unique string expression that can be used to access a particular button on a control.

`Style` Property

The `Style` property of the `Button` object determines how a button performs on a toolbar. The values that the `Style` property can take are as follows:

- **`tbrDefault (0)`** This is the default value for the `Style` property. When buttons have this style, they appear and behave as standard `CommandButton` controls.
- **`tbrCheck (1)`** The button will function as a check box. This style goes along with the `Value` property (`Pressed` or `Unpressed`). When this button is clicked, it will stay pressed or indented until it is clicked a second time.
- **`tbrButtonGroup (2)`** Buttons with this style function as `OptionButton` controls. Buttons on a toolbar can be grouped together, and all have a style of `tbrButtonGroup` if you want to give the user mutually exclusive options. When the user selects a button in the group, that button will stay depressed (value of 1) until another button in the group is clicked. Button groups are separated from the rest of the buttons in the toolbar by a separator (see following item).
- **`tbrSeparator (3)`** If you use this style for a button, it will not appear as a button at all but as a space between buttons. The space will be a fixed eight pixels wide.
- **`tbrPlaceholder (4)`** This style will allow a button to act like a separator, but the width is adjustable.

`Appearance` Properties

Several other properties of the `Button` object control the appearance of each button on the toolbar.

The first of these is the `Image` property. The value of `image` is an integer that maps to the index of an image in an `ImageList` control. The `ImageList` used is determined by the `ImageList` selected on the General tab of the Property Pages dialog box.

In addition to an image, you can place a caption on each button of your toolbar by setting the `caption` property. If you have both an image and a caption on a `Button` object, the caption will show below the image.

ToolTips can be added to each button on a toolbar by setting the `ToolTipText` property.

If the `ShowTips` property of the toolbar is set to `True`, the `ToolTipText` of a button will appear to the user when the mouse pointer is rested over that button. This is valuable for new users of an application who do not yet know the purpose of each button.

`Add` and `Remove` Methods

Buttons can be added and deleted from a toolbar at runtime by using the `Add` and `Remove` methods. The `Add` method of the `Buttons` Collection has the following syntax:

```
Toolbar1.Buttons.Add(index, key, caption, style, image)
```

`Index` is the position at which you want to insert the new button. If you do not specify an index, the new button will appear at the right end of the existing buttons. The `key` is a string value that can be used to reference the new button. The `caption` argument will be any text that you want to appear on the new button. `Style` specifies how a button will behave (as described earlier). Finally, the `image` argument is an integer value that corresponds to an image in the `ImageList` connected to the `ToolBar` control.

To delete an existing button from the toolbar, you can use the `Remove` method:

```
Toolbar1.Buttons.Remove index
```

where the `Index` argument is either the integer value for the `Index` of the `Button` object you wish to remove or a string value for the `Key` of the `Button` object.

`ButtonClick` Event

The `ButtonClick` event for the `ToolBar control` will fire whenever a user clicks on a `Button` object. For each toolbar there is a single `ButtonClick` event, not one for each button on the toolbar. The event is fired as follows:

```
Private Sub Toolbar1_ButtonClick(ByVal button As Button)
```

where the `Button` argument is a reference to the `Button` object that has been clicked. You can use this argument to determine which button was clicked by the user and what to do as a result by using a `Select Case` statement, as shown in Listing 4.4.

LISTING 4.4

REACTING TO A USER CLICK ON A TOOLBAR BUTTON

```
Private Sub Toolbar1_ButtonClick(ByVal button As Button)

    Select Case button.Key
        Case "Open"
          ' open an existing file
        Case "New"
          ' create a new file
        Case "Exit"
          ' close the application
    End Select

End Sub
```

Customizing Toolbars

Many applications are now allowing users to customize toolbars with their own preferences. You can also provide your users the ability to customize the toolbars you create within your application. The first thing you must do before a user can change your toolbar is to set the `AllowCustomize` property to `True`. As long as `AllowCustomize` is `False`, the user cannot make changes.

After `AllowCustomize` is set to `True`, customization takes place through the Customize Toolbar dialog box, as shown in Figure 4.11. This dialog box becomes available to the user in one of two ways:

- The user double-clicks on the `ToolBar control`.
- The `Customize` method of the toolbar is called.

FIGURE 4.11
The Customize Toolbar dialog box.

If you will be allowing a user to make changes to a toolbar, you should consider using two additional methods: `SaveToolbar` and `RestoreToolbar`. With these two methods, you can save settings from the toolbar in the Windows Registry and then read them back at a later time to restore the appearance of a toolbar either to the original settings or to the user's personal settings when the application is restarted.

The `SaveToolbar` method has the following syntax:

```
Toolbar1.SaveToolbar(key As String, subkey As String, value
➥As String)
```

The `RestoreToolbar` has the following syntax:

```
Toolbar1.RestoreToolbar(key As String, subkey As String,
➥value As String)
```

The `key` and `subkey` arguments are the key and subkey of the Windows Registry at which you are storing information or from where you are retrieving information. The `value` argument is the information that you are saving or reading.

Using the `StatusBar` Control

The `StatusBar` control is another ActiveX Control available through the Windows Common Controls components. With the `StatusBar`, you can easily display information about the date, time, and other details about the application and the environment to the user.

`Panel` Object and `Panels` Collection

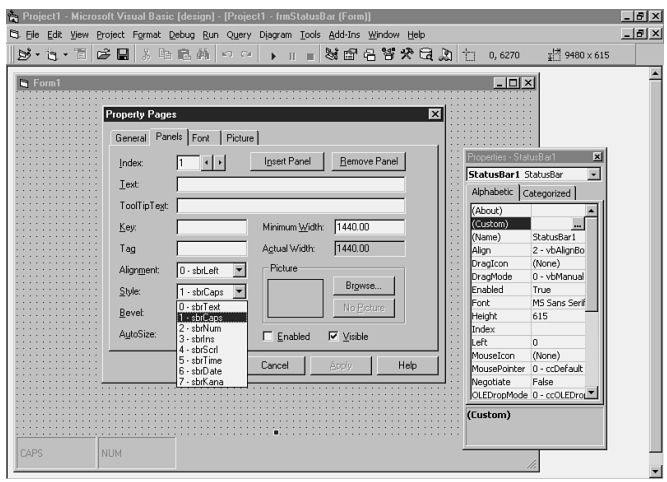
A `StatusBar` is made up of `Panel` objects, each of which displays different types of information. All the `Panel` objects are contained in the `Panels` Collection of the `StatusBar`. The appearance and purpose of each `Panel` is determined by the `Style` property of that `Panel`. The following styles are available:

- **`sbrText (0)`** The Panel will display text information. The information displayed is in the Text property of the Panel.
- **`sbrCaps (1)`** The `Panel` will contain the string "Caps". It will be highlighted if Caps Lock is turned on or disabled if Caps Lock is turned off.
- **`sbrNum (2)`** The `Panel` will contain the string "Num". It will be highlighted if Num Lock is turned on or disabled if Num Lock is turned off.
- **`sbrIns (3)`** The `Panel` will contain the string "Ins". It will be highlighted if Insert mode is turned on or disabled if Insert mode is turned off.
- **`sbrScrl (4)`** The `Panel` will contain the string "Scrl". It will be highlighted if Scroll Lock is turned on or disabled if Scroll Lock is turned off.

- **`sbrTime (5)`** The `Panel` will display the current time in the system format.
- **`sbrDate (6)`** The `Panel` will display the current date in the system format.
- **`sbrKana (7)`** The `Panel` will contain the string "Kana". It will be highlighted if Kana (a special character set only available in the Japanese PC market) is enabled or disabled if Kana is disabled.

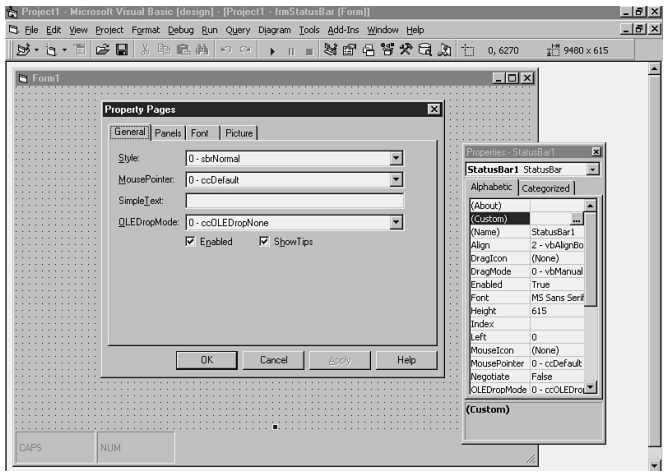
Figure 4.12 shows an example of a status bar displaying all the different styles.

FIGURE 4.12
Different styles of the `Panel` object.



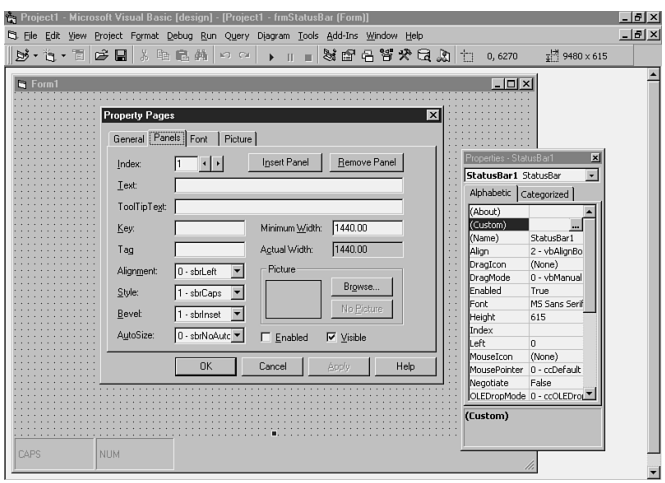
As with the other controls discussed in this chapter, the properties of the `StatusBar` and the `Button` objects can be set either through the Property Pages dialog box at design time or at runtime. The properties of the `StatusBar` control can be set on the General tab, as shown in Figure 4.13.

The `Style` and `SimpleText` properties of the toolbar determine the appearance of the control at runtime. The `Style` property can have one of two values: `Simple` or `Normal`. When the `Style` property is `Simple`, only one panel of the `StatusBar` is visible, and the text in the `SimpleText` property is displayed. When the `Style` property is `Normal`, the `StatusBar` appears with multiple panels.



◀**FIGURE 4.13**
General properties of the `StatusBar`.

Further properties of the `Panel` objects give more control over the appearance of a `StatusBar` control. These properties can be set on the Panels tab of the Property Pages dialog box, as shown in Figure 4.14, or at runtime.



◀**FIGURE 4.14**
Custom Properties of the `Panel` object.

The use of ToolTips is available to you with the `StatusBar` control. Behavior of ToolTips for the `StatusBar` is similar to the behavior of ToolTips for the `ToolBar` control (discussed earlier in "Using the `ToolBar` Control").

An individual `ToolTipText` property is available for each panel of the `StatusBar` control. When the `ShowTips` property of the `StatusBar` is set to `True` and the user rests the mouse pointer over a panel in the `StatusBar`, the `ToolTipText` is displayed at the mouse pointer.

Several other `Panel` properties affect the appearance of each panel. The `Alignment` property determines whether text displayed in a panel is left-justified, right-justified, or centered. The `MinWidth` property specifies the minimum width of a panel. The `ScaleMode` of the container on which the `StatusBar` is located determines the units for the `MinWidth`. Each panel also has a `Width` property that determines the starting width of the object.

The `AutoSize` property of a panel can also have an impact on its width. This property can have the following values:

- **`sbrNoAutoSize (0)`** The panel size will not be changed automatically.
- **`sbrSpring (1)`** If the size of the `StatusBar` increases as the size of the container grows, the width of the `Panel` will also increase. As a `StatusBar` grows smaller, the size of the `Panel` will not go below the `MinWidth` value.
- **`sbrCntents (2)`** The `Panel` will be resized to fit its contents but will never fall below the `MinWidth`.

Another property that affects a panel's appearance is the `Picture`.

Unlike the other controls discussed in this chapter, the `StatusBar` does not get images from an `ImageList`.

If you wish to include images—either bitmaps or icons—in a panel, you can add them through the Property Pages dialog box by specifying the file directly. You can also add them at runtime with the `LoadPicture` function, which takes the image file path as its argument, or by setting the Picture property to one of the `ListImages` in an `ImageList` control or to the appropriate bitmap property of some other control.

`Add` and `Remove` Methods

The `Add` method of the `Panels` Collection can be used to add panels to the `StatusBar` at runtime. The syntax is similar to the other `Add` methods discussed in this chapter:

```
StatusBar1.Panels.Add(index, key, text, style, picture)
```

`Index` and `Key` are the unique identifiers that enable you to access a particular `Panel` object. The `Text` argument is the text that appears in a panel. How a `Panel` object appears and what data it contains is determined by the `Style` argument. Finally if you want to add an image in the new panel, you can use the `LoadPicture` function in the `Picture` argument to include that image along with any other text that appears.

Panels can be removed at runtime by using the `Remove` method of the `StatusBar`. To remove a `Panel` object, you would use the following:

```
StatusBar1.Remove index
```

where the `Index` argument is either an integer equal to the `Index` property of the panel that you are removing or a string value equal to the `Key` of the panel.

Using the Controls Collection

The Controls Collection is a built-in collection belonging to a form and is maintained by Visual Basic. The Controls Collection contains references to all the controls on that form. Each member of the collection points to a different object on that form—for example, a `TextBox`, `ComboBox`, `ListView`, and so on.

Each form in a project has its own Controls Collection that you can use to navigate through the different controls on a form. You can refer to each control within the collection as follows:

```
Form.Controls(index)
```

where `Form` is the form name, and `Index` is an integer value referring to the desired control. Since the Controls Collection is zero-based, Index values range from 0 up through a number that is one less than the total number of controls on the form. If you have 10 controls on a form, the Index values for the Controls Collection would range from 0 through 9.

You will usually use in a Controls Collection when you need to process all the controls or the majority of the controls on a given form. You can loop through a Controls Collection either with a `For Next` loop, as in Listing 4.5, or with a `For Each` loop, as in Listing 4.6.

LISTING 4.5

Looping Through a Controls Collection With a For Next Loop

```
For I = 0 To Form1.Controls.Count - 1
    ' process Form1.Controls(I)
Next
```

LISTING 4.6

Looping Through a Controls Collection With a For Each Loop

```
For Each obj In Form1. Controls
        ' process obj
Next
```

A common use of the Controls Collection is to clear all the controls on a form. If the user wants to reset all the fields on a data entry form, you can code a Clear button with code such as that found in Listing 4.7.

LISTING 4.7

Code to Reset All Fields on a Data Entry Form

```
Private Sub cmdClear_Click()

    Dim objControl As Control
    Dim sTemp As String

    For Each objControl In Me.Controls

        If TypeOf objControl Is TextBox Then
            ' clear the text
            objControl.Text = ""
        ElseIf TypeOf objControl Is ComboBox Then
            ' reset the listindex
            objControl.ListIndex = -1
        ElseIf TypeOf objControl Is Label Then
            ' leave labels as is
        ElseIf TypeOf objControl Is CommandButton Then
            ' leave commandbuttons as is
```

```
        ElseIf TypeOf objControl Is MaskEdBox Then
            ' clear the masked edit control
            sTemp = objControl.Mask
            objControl.Mask = ""
            objControl.Text = ""
            objControl.Mask = sTemp
        Else
            ' leave any other control alone
        End If
    Next

End Sub
```

This code in the Listing loops through all the controls on a form and clears each control based on its type. To clear a `TextBox` control, for example, you would set the `Text` property to a 0-length string. To clear a `CheckBox`, you would set the `value` property to `vbUnchecked`.

Notice that the code in the Listing uses an `If` statement to determine the type of control being cleared. You must make sure you know the type of control you reference to avoid runtime errors. Errors occur if you try to reference a property of a control that is not a valid property. If the first control on a form is a `TextBox`, you can clear that control by using:

```
Me.Controls(0).Text = ""
```

If, on the other hand, the first control is a `CheckBox` that does not have a `Text` property, with the same code, you will get a runtime error from Visual Basic. The Clear button code example avoids this problem by checking for specific control types within the Controls Collection. The syntax for the `If` statement is as follows:

```
If TypeOf objectname Is objecttype Then
   ' code
EndIf
```

where `objectname` is the object or control (that is `TextBox`, `ComboBox`, and so forth) on which you are working and `objecttype` is the class of the object. Using this `If` structure before you reference a control through the Controls Collection will help you avoid runtime errors.

Techniques for Adding and Deleting Controls Dynamically

Occasionally you will create forms for which you do not always have a fixed number of controls. You may need to add controls at runtime or remove some of the controls that you have created. Visual Basic lets you create and destroy controls dynamically at runtime based on the needs of your application.

With VB6, you have at your disposal two major techniques for dynamic control creation:

- Control arrays
- Direct manipulation of the Controls Collection through the `Add` and `Remove` methods

We discuss these techniques in the following two sections.

Adding and Deleting Controls Dynamically Using Control Arrays

You must follow several rules to be able to create and remove controls with control arrays.

First you must have a control of the type that you will be adding placed on the desired form at runtime. If you will be adding `TextBox` controls to a form, for example, you must have at least one text box drawn on that form at design time. That control can be invisible, and there are no restrictions for the size or placement of that control, but it must be on the form at design time. This control will be the template for text boxes that you will add at runtime.

The second requirement for using dynamic control arrays is that the template object that you draw at design time must be part of a control array. Usually it is the only control of its type with its `Index` property set to a value of 0. Continuing with the text box example, you can have a form with only one text box as a template, and that text box must have its `Index` property set to some integer value (typically 0 or 1). If your application required it, you might have additional text boxes with `Index` values of 1, 2, 3, and so on. As long as you have a control array with at least one object in it, you can create additional instances of that object dynamically at runtime.

After you have built a form with a control that has its `Index` property set, you can add additional controls to the control array at runtime.

Assume, for example, that you have an application with one form, `Form1`, and one text box, `Text1`. `Text1` has an `Index` value of 0. At runtime, you can create additional instances of `Text1` on your form with code such as:

```
Load Text1 (index)
```

where `Index` is an integer value that will be used as the index of the new text box. When you run the application and want to create the first dynamic instance of `Text1`, you would use:

```
Load Text1 (1)
```

because `Index` value 0 is already in use by the `Template` control.

After you have loaded the new control on the form, you must set the appearance as desired. You have to set the `Visible` property to `True`, if you want the control to appear to the user. Also you must change either the `Left` or `Top` properties in order to place the control on the form (otherwise the control will show up in exactly the same spot as the previous control). Finally you can change the `Height`, `Width`, and any other property that can be altered at runtime.

Code for dynamically adding labels to a control array might look like the code in Listing 4.8.

NOTE

`Index` Property Is Blank by Default
By default, the `Index` property of a control is blank, meaning that it is not part of a control array.

LISTING 4.8

LOADING LABELS INTO A CONTROL ARRAY

```
' load a new control into the control array
    i = i + 1
    Load lblTemplate(i)

    ' position the new control on the form and add a caption
    lblTemplate(i).Left = lblTemplate(0).Left
    lblTemplate(i).Top = lblTemplate(i - 1).Top + _
                         lblTemplate(i - 1).Height + 100
    lblTemplate(i).Caption = "index = " _
                             & Str$(i)

    ' make the control visible
    lblTemplate(i).Visible = True
```

After you have added controls to a control array on a form, you may need to remove them at some point. To remove a control that you have dynamically added to a control array, you simply code the following:

```
Unload Text1(index)
```

where `Text1` is the name of the control you loaded previously, and `Index` is the integer value of that control's `Index` property.

It is important to remember that you cannot use the `Unload` statement to remove a control on the form that was added at design time—this will cause a runtime error. Only controls added dynamically can be removed with `Unload`.

Also, you can't add a control twice using the same `Index` value, and you can't delete a control with an `Index` value that isn't in use. Trying either one of these stunts in code will generate a runtime error.

Adding and Deleting Controls Dynamically Using the Controls Collection

The `Add` and `Remove` methods of the Controls Collection are new to VB6. You can use these methods to add and delete controls from a form instead of using the control array technique described in the previous section.

Following is an overview of the general steps that you need to take in order to dynamically add and remove controls with the Controls Collection (more detailed discussion is given in the following sections):

1. Find out the control's ProgID, a unique string used by the Windows operating system (and stored in the Windows registry) for identifying the control's type.
2. If the control is an intrinsic VB control, declare an object variable of the appropriate control type using WithEvents and program the resulting object's event procedures. If the control is an intrinsic VB control, then ignore steps 3, 6, and 7 that only apply to non-intrinsic controls.
3. If the control is an ActiveX control (i.e., not an intrinsic VB control) then you must declare the type of its object variable as VBObjectExtender and place code in its ObjectEvent procedure to trap for the various events that you're interested in.

4. Use the `Add` method of the Controls Collection to initialize the control with the ProgID that you determined in step 1, and set the result of the method to the object variable you declared in step 2 or 3. Set the control's `Visible` property to True and set any other properties that need to be changed. If the control is an ActiveX control, you'll need to refer to its members through the `Object` property of the control object variable.
5. Use the `Remove` method of the Controls Collection to remove the control from the Controls Collection when your program is finished using the control.
6. If an ActiveX control is in the Toolbox but is not otherwise referenced in your project with a design time instance on the surface of a form, then you must make sure that your project's properties are set appropriately to allow information about unused ActiveX controls to remain in the project.
7. If an ActiveX control requires a license, then you must detect the control's license ID in your design time test environment and use that license ID to initialize the control in the compiled application that you distribute to end users. In order to do this legally, you must be licensed to use and distribute this control.

We discuss these steps in the following six sections.

Getting a Control's ProgID

A Control's ProgID, as mentioned above, is a unique system-wide string that the Windows operating system can use to identify your control's type. You must know the ProgID of any control that you want to add to the Controls Collection because the `Add` method will require it as an argument.

For VB intrinsic controls the ProgID is usually formed by combining the characters VB. with the VB name of the control's type (CommandButton, TextBox, or Label for example). So, for example, the ProgID for CommandButtons is VB.CommandButton and the ProgID for Labels is VB.Label.

If you want to add an ActiveX control (a non-intrinsic control) to the Controls Collection, you'll either need to have some documentation at hand or you'll need to do some investigation in the Windows Registry.

In order to find the ProgID for a control's type in the Windows Registry, you can run Regedit from the Start/Run menu option of your Windows desktop. Perform an Edit/Search in Regedit for the name of the control. Repeat the search with the F3 key until you find the ProgID entry (see Figure 4.15). You can use this key with the `Add` method.

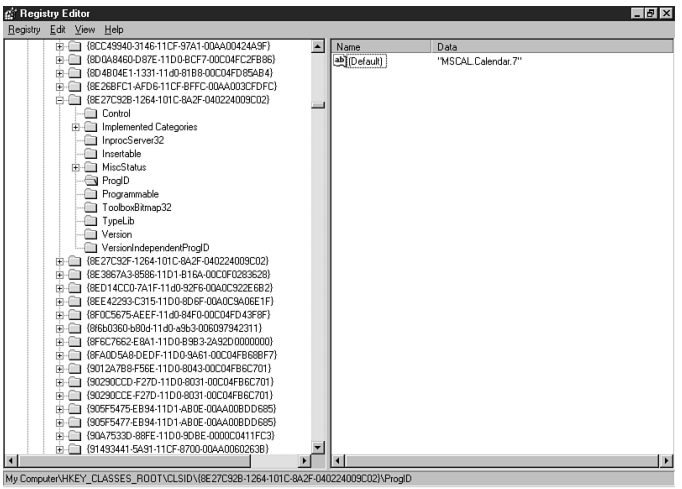


FIGURE 4.15
Finding a control's ProgID in the Windows Registry using the Regedit utility.

NOTE

Limitations on the Use of `WithEvents`
You can only use the `WithEvents` keyword in the General Declarations sections of designer-type objects, such as forms, `UserControls`, and `Active Documents`. If you try to use `WithEvents` in the General Declarations of a Standard (.bas) Module or in a local variable declaration within any kind of procedure, you'll receive a compiler error.

Declaring an Intrinsic Control and Programming Its Events

Having determined the ProgID for your intrinsic control, you can declare a variable of the desired control type, as in the examples of Listing 4.9:

LISTING 4.9

DECLARATIONS OF OBJECT VARIABLES FOR INTRINSIC VB CONTROLS THAT WILL BE ADDED TO THE CONTROLS COLLECTION AT RUNTIME

```
Option Explicit
Private WithEvents cmdMyButton As CommandButton
Private WithEvents txtMyTB As TextBox
Private WithEvents lblMyLabel As Label
```

Notice that the three examples all use the `WithEvents` keyword. This means that you'll be able to find the declared objects in a code window's drop-down list and that each object will further provide its event procedures in the rightside drop-down of the code window for you to program, as illustrated in Figure 4.16.

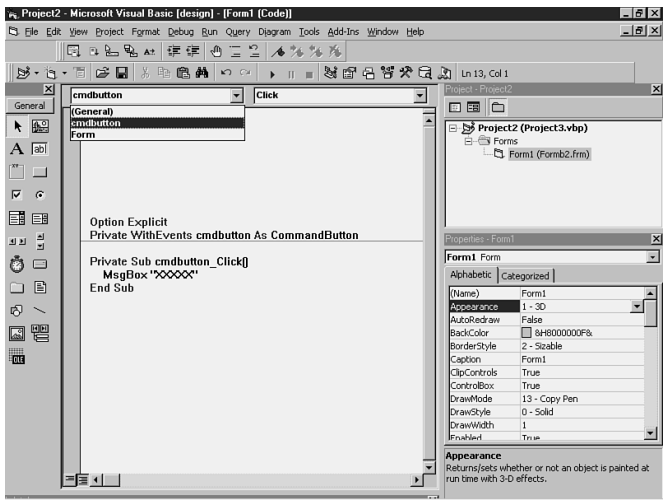


FIGURE 4.16
Once you've declared your control using `WithEvents`, you can see the control and its events in the drop-down lists of the Code window.

You can then write your own event-handling code in the event procedures.

Declaring an ActiveX Control and Programming Its Events

You can also declare an ActiveX (non-intrinsic) control using the `WithEvents` keyword and then write program code to react to its events.

However, the techniques that you must use to program with an ActiveX control and its events are quite different from those used for an intrinsic control.

First, you must declare the control's object variable as of type `VBObjectExtender`. Use the `WithEvents` keyword in the declaration if you wish to program with the control's events.

In order to program the control's event procedures, you will have to use a general event handling procedure belonging to the `VBObjectExtender` control object known as the `ObjectEvent` procedure.

You can find this procedure among the object's event procedures by navigating to it in the Code window, just as you would navigate to an intrinsic control object's event procedures (see the previous section).

The `ObjectEvent` procedure takes a single parameter known as `Info`. The `Info` parameter's type is `EventInfo`. The `EventInfo` type contains two properties:

- **`Name`** This property gives the name of the Event that has fired.
- **`EventParameters`** The fact that its name is a plural word should tip you off that the `EventParameters` property is a collection of `EventParameter` objects. Like all VB collection objects, `EventParameters` has a `Count` property. If `Info.EventParameters.Count` is greater than 0, then the event that's just fired has parameters. Each `EventParameter` object has a `Name` property and a `Value` property. If you know the name of a parameter that interests you, then you can use that name in a string variable to index the `Info` object's `EventParameters` collection and so read or write the value of the parameter in question (use the `Value` property of the `EventParameter` object).

The example in Listing 4.10 shows code that will tell you the name of any event that fires for the control that the event procedure belongs to. If the event supports one or more parameters (the code checks the `Count` property of the `EventParameters` collection), then the code will show the names and values of those parameters by looping through the `EventParameters` collection.

LISTING 4.10

`ObjectEvent` PROCEDURE CODE THAT WILL DISPLAY THE NAME OF ANY EVENT THAT FIRES, AS WELL AS THE NAMES AND VALUES OF ANY PARAMETERS THAT THE EVENT SUPPORTS

```
Private Sub mscCalendar_ObjectEvent(Info As EventInfo)
    'Display the name of the event
    'that has just fired
    Me.Print Info.Name

    'If the event has any parameters
    If Info.EventParameters.Count > 0 Then
        'set up a variable to hold each
        'object visited in the For...Each loop
        Dim evinf As EventParameter
```

```
        'Loop through the EventParameters collection
        For Each evinf In Info.EventParameters

            'For each Parameter object found
            'Display its name and value
            Me.Print , evinf.Name, evinf.Value
        Next evinf
    End If
End Sub
```

The code in Listing 4.11 was written for an `MSCalendar` control's `ObjectEvent` procedure and will prevent the user from setting the date to the first of the month. The code in the `Calendar` control's `ObjectEvent` procedure checks to see if the `BeforeUpdate` event has fired. If the event at hand is the `BeforeUpdate` event, then the code checks to see if the user is trying to set the day of the month to the first. If the user is indeed trying to set the day of the month to the first, then the code uses the `Info` object's `EventParameters` Collection to set the `Cancel` parameter's value to `True`. Setting the `Cancel` parameter to `True` will cancel the user's action.

LISTING 4.11

`ObjectEvent` PROCEDURE CODE FOR AN MS CALENDAR CONTROL THAT WILL PREVENT THE USER FROM SETTING THE DAY OF THE MONTH TO THE FIRST

```
Private Sub mscCalendar_ObjectEvent(Info As EventInfo)
    'Get the name of the event that has fired
    Dim strEventName As String
    strEventName = UCase$(Trim$(Info.Name))

    'If it's the BeforeUpdate event
    If strEventName = "BEFOREUPDATE" Then

        'If the user is trying to set the
        'day-of-month to be the first, then
        If mscCalendar.object.Day = 1 Then

            'Set the Cancel parameter to True
            'to undo the user's change
            Info.EventParameters("Cancel").Value = True
        End If
    End If
End Sub
```

Note in Listing 4.11 that any reference to the control's custom members (refer to the `Day` property in the example) must be made through the `Object` property of the `VBObjectExtender` variable.

If the code referred to standard members (such as Top, Left, or Visible), it would not use the `Object` property. In fact you would receive a runtime error if you attempted to refer to a standard property through the `Object` property.

Adding and Removing a Control in the Controls Collection

Once you've declared the intrinsic or ActiveX control object and programmed its events, you can add it to the Controls Collection with a line of the format:

```
Set objvariable = Controls.Add(strControlType,
➥strControlName)
```

where *objvariable* is the name of the variable holding the control's instance, *strControlType* is a string containing the `ProgID` for this control's type, and *strControlName* is a string containing a unique name for the control (can be the same as *objvariable*).

Once you've added the control, you need to set its object variable's properties appropriately to the needs of the application and set the `Visible` property to True. The method for referring to a control object's members varies slightly, depending on whether the control is an intrinsic or an ActiveX control:

- If the control is an intrinsic VB control, then simply refer to the control's members with the familiar *objvariable.member* syntax. Listing 4.12 gives an example that sets the properties of an intrinsic control that's just been added to the Controls Collection.

LISTING 4.12

REFERRING TO AN INTRINSIC CONTROL'S MEMBERS

```
Set cmdbutton = Controls.Add( _
                "VB.CommandButton",  _
                "CmdButton")
cmdbutton.Caption = "XXXXX"
cmdbutton.Left = 500
cmdbutton.Top = 500
cmdbutton.Visible = True
```

- If the control is an ActiveX control, then you must refer to the control's *custom* members through the control's object property with syntax in the format *objvariable*.`object`.*member*. Listing 4.13 gives an example that sets the properties of an ActiveX control that's just been added to the Controls Collection. Note in the example that you do *not* use the `Object` property when referring to a *standard* property of the VB environment. You only use the `Object` property when referring to *custom* members. Referring to *custom* members without the `Object` property or to *standard* members *with* the `Object` property will cause a runtime error.

LISTING 4.13

REFERRING TO AN ACTIVEX CONTROL'S STANDARD AND CUSTOM MEMBERS

```
'Add the control
Set mscCalendar = Controls.Add( _
                     strControlType,  _
                     "mscCalendar")

'Set a custom property
mscCalendar.object.Day = 15

'Set standard properties
mscCalendar.Top = 600
mscCalendar.Left = 500
mscCalendar.Visible = True
```

See Exercise 4.9 for examples of the manipulation of both intinsic and ActiveX control object members.

When you're finished with the control, you can take it out of the Controls Collection with a line of the format:

```
Controls.Remove objvariable
```

where objvariable is the name of the variable holding the control's instance.

Exercise 4.9 illustrates the use of the `Add` and `Remove` methods of the Controls Collection.

Keeping a Reference in the Project to an ActiveX Control

When you create an exe or run your project in the IDE, VB6 by default removes references to ActiveX controls that are in the Toolbox but not used by the program. VB is able to remove this extraneous reference information by checking to see whether instances of a control exist on the surfaces of forms or other container objects.

However, VB6 cannot detect when an ActiveX control of a particular type is going to be added to the control array, because the type of such a control is declared as VBObjectExtender. The control object is therefore late bound and the compiler cannot verify its existence in the project.

Thus, the VB6 compiler may remove information about a control type that you want to add to the Controls Collection if you have placed the control in the Toolbox and you have not put design time instances of that control in your project.

When your code attempts to initialize such a control, the compiler will give you an error message warning that a reference is lacking (see Figure 4.17).

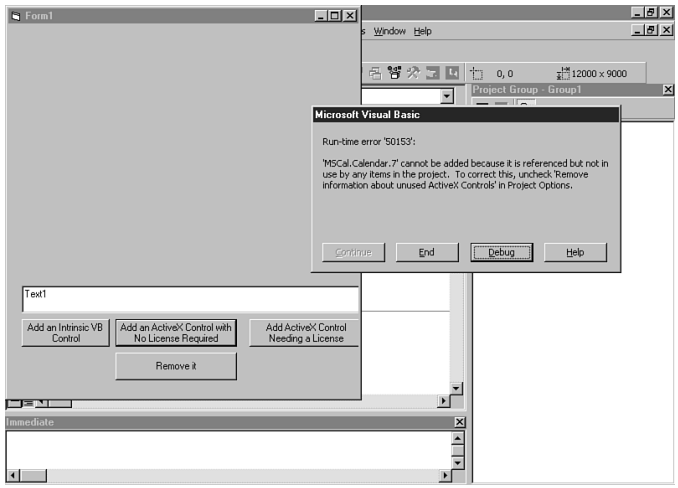


FIGURE 4.17
Compiler error for a deleted reference to an ActiveX control.

The warning illustrated in Figure 4.17 also tells you what to do in order to keep the reference to the control in your application:

Change the default compiler behavior by un-checking the option labeled Remove information about unused ActiveX Controls in the Make tab of the Project Properties dialog box (see Figure 4.18).

If you un-check this option, then you will not receive the message of Figure 4.17 again.

Of course another way to avoid this situation is by simply removing the control from the Toolbox if you don't plan to place design time instances of it in your project (uncheck the control's library in the Project Components dialog). The system will still recognize your reference to the control's ProgID in the `Controls.Add` method and will instantiate the correct control type.

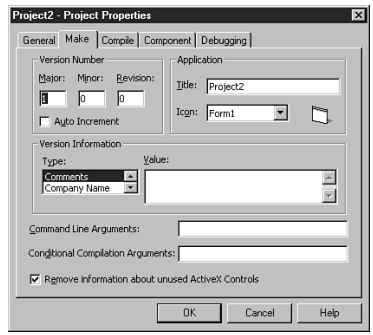
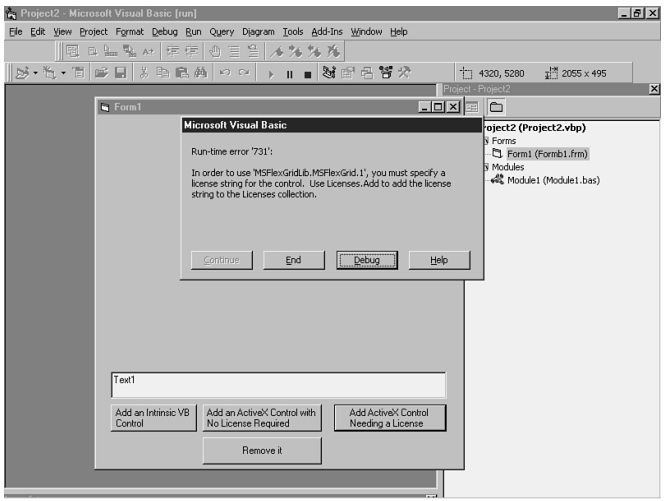


FIGURE 4.18▲
The Make tab of the Project Properties dialog showing the option that controls references to unused ActiveX controls.

Managing the License for an ActiveX Control

Many ActiveX controls cannot be freely distributed to end users but instead require a license before the system will allow them to be used in applications.

VB will raise a runtime error if an ActiveX control requires a license key and you attempt to add an instance of that ActiveX control to the Controls Collection without providing the license key (see Figure 4.19).



◀FIGURE 4.19
A runtime error caused by the lack of a license key for an ActiveX control added to the Controls Collection.

To provide a license key to the runtime environment, you must determine the contents of the key and then add the key to the `Licenses` Collection before trying to add an instance of the control to the Controls Collection. Add the key to the `Licenses` collection with a call to its `Add` method using the following syntax:

```
Licenses.Add Controltype, LicenseKey
```

where *ControlType* is the string representing the control's ProgID and *LicenseKey* is the string representing the license key for that type of control.

So how do you find out the license key for a particular control type?

In your development environment, you can make a call to `Licenses.Add` that automatically adds the control's License key to the Licenses Collection without having to know the License key beforehand, and at the same time provides the license key in a string that you can examine. The syntax for such a call would be:

```
StrLicenseKey = Licenses.Add(ControlType)
```

where *strLicenseKey* is a string variable that will hold the license key for the control type. When you run this line in your development environment, you can then examine the contents of *strLicenseKey* (using a Debug.Print message or some other display technique) to find out the contents of the license key for that control type.

When you release your product to end users, you can change your code to the first syntactic format listed above to allow your application to load the control on the user's workstation. You would hard-code the key that you discovered as the second argument to the `Licenses.Add` method.

More on Creating Data Input Forms and Dialog Boxes

This section examines the use of VB `Form` objects within a project. Topics covered include:

- Loading form objects into memory and unloading them
- Showing and hiding forms from the screen
- Showing how VB `Form` objects are stored

When a project's design specifies multiple forms, it is the programmer's job to ensure a clean, easy-to-use interface. The knowledge of how forms are loaded into memory, the effect when forms leave memory, when to expect a form to be displayed, and when it is removed are critical elements in designing a professional-looking and behaving application. Also knowing how information is stored at design time will help to ensure forms that can be used throughout various projects or moved from one version of VB to another.

Forms are used to contain other controls for user interaction. When one object has the capability to host other controls, this object is known as a `Container`. When controls are placed on a form, they are affected when the form is moved or resized. This allows the form to act as a `Container` object. The `Picture` control is also a `Container` and can have other controls placed on it.

This section looks at how to load and unload forms from memory, how to show and hide forms from the screen, and how design time forms are stored.

Loading and Unloading Forms

Visual Basic projects have a special object that can be automatically loaded when the program is run. This object is referred to as the *Startup object*. Figure 4.20 shows an example of a `Startup` object. The `Startup` object can now vary depending on the type of project you are creating. A form can be selected, the `Sub Main` procedure, or—new in VB5 and VB6—nothing. You would specify Nothing when the program does not require an interface. An example of this type of project might be an ActiveX component or an ActiveX control.

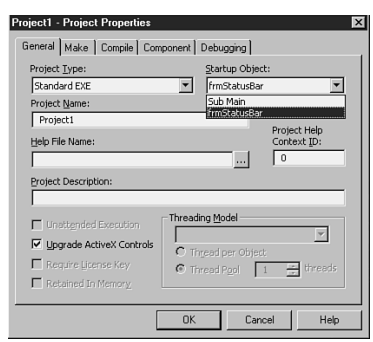


FIGURE 4.20
Selecting the project `Startup` object.

When a form is specified as the `Startup` object, that form automatically loads into memory when the application starts. No other form loads unless it is referenced in program code or is explicitly loaded into memory.

Use the `Load` statement to load a form into memory without making it visible yet. The `Load` statement will take only one argument: the name of the object to be loaded. Take a look at the following code:

```
Load Form1
Load frmTest
```

The `Load` statement in both cases accepts a valid object name. This causes the object to load into memory. Although the object loads into memory, this does not mean that the object will be visible to the user. This enables the programmer to load multiple forms that may be required and to prepare them with information before display. Once loaded into memory, the form's controls and any program code can be used.

When working with forms in VB, it is important to note that any reference to an object will cause that object to load. The `Load` statement does not have to be explicitly used before an object can be used. An example of this would be if a form's `Caption` property were set, as follows:

```
Form1.Caption = "My Notepad"
```

Notice that this is the only line of code that you need to be concerned with. There is no `Load` statement before the `Caption` property is set. This code directly sets the form's property. This single line of code automatically causes the `Form1` object to be loaded into memory. This is often referred to as *implicit loading*. Because the object must be loaded to set the property, VB does exactly that.

Implied loading can often cause problems when working on a multiform project. The programmer does not notice that one form calls or sets information on another form. The form then automatically loads. Later when you attempt to unload by name all forms that you remember using, your project continues to run.

The `End` statement can be used to force the application to terminate, regardless of which forms were explicitly or implicitly loaded.

This is a fail-safe in case the program missed a reference or miscounted how many forms were loaded.

However, the `End` statement will have an undesirable effect because it will end the application so abruptly that the `QueryUnload` and `Unload` events of forms will not have a chance to run.

For a more acceptable method to unload all forms, see the section in this chapter "Using the Forms Collection to Unload All Forms."

When an individual form is no longer required, you can unload it from memory. This will release the graphic components from memory. The following code unloads two forms:

```
Unload Form1
Unload frmTest
```

The `Unload` statement accepts a valid object name. This causes the design time graphic components of a form to be released. Although the form has been unloaded from memory, it is very important to note that any Public procedures or variables belonging to the form are still in memory. The graphical controls are no longer available, but any Public members of the form can still be called, using the syntax for calling any object variable's members, *formname.procname*. Remember, a form's Public procedures and variables are considered to be members (i.e., methods and properties) of the instance of the form object. Of course what's just been said for Public variables also goes for a form's Custom Properties implemented with Property Get and Property Let/Set procedures.

`Load` and `Unload` are used to control the memory status of a form. These two statements always appear before the name of the object to be affected. They are often confused with the `Show` and `Hide` methods that take place after the object name. `Show` and `Hide` are used to control whether a form is visible to the user.

After you unload a form, the form's Public procedures and its Public variables will still be resident as mentioned above. To completely reset the contents of these elements, call the line

```
Set FormName = Nothing
```

in your code just after calling the `Unload` statement.

Showing and Hiding a Form

Loading and unloading only bring the form into memory or remove the form from memory. If the programmer wants to display the form for the user to interact with, another set of commands must be used. The `Show` and `Hide` methods affect the form's visibility. Unlike the `Load` and `Unload` statements, which you write in code before the form name, `Show` and `Hide` follow standard method syntax and precede the object name, separated from it by a period (.).

When a form is to appear on-screen for the user to interact with, the `Show` statement causes the form to appear. `Hide` will make the form invisible, but allows it to remain in memory.

If the form is to be directly shown to the user, only the `Show` method is required. The loading of the `Form` object will take place automatically. The following lines of code are all that is required to both load the form and have it displayed:

```
Form1.Show
```

To understand why the form does not have to be explicitly loaded, first always remember that any programmatic reference to an object will cause it to automatically load. That explains why using the object name followed by the `Show` method causes the form to load first and then display onscreen. For further discussion of implicit loading, see the previous section in this chapter "Loading and Unloading Forms."

Calling a form's `Show` method will also cause the form to become the application's *active* form—that is the form where focus resides in the application. Usually the form itself does not have focus but rather a control on the form. This is because a form object itself cannot get focus unless the form contains no controls that are currently able to receive focus.

If the form is no longer required to be on-screen, it can be removed from display by just using the `Hide` method. This keeps the form loaded but removes it from display. The following code demonstrates this:

```
Form1.Hide
frmTest.Hide
```

The first line of code removes the form named Form1 from the on-screen display. The second line of code removes the object named frmTest. Both prevent user interaction with the form and help you avoid a very busy screen.

Forms can be hidden instead of being unloaded. If a form is hidden without being unloaded and destroyed, then the values of controls will remain as the user entered them. Other code in the project can then refer to the contents of the controls on that form.

If the form were unloaded, this would re-initialize the controls on the form. Every time the user wanted the settings, they would either have to be loaded from an external source or reset by the user.

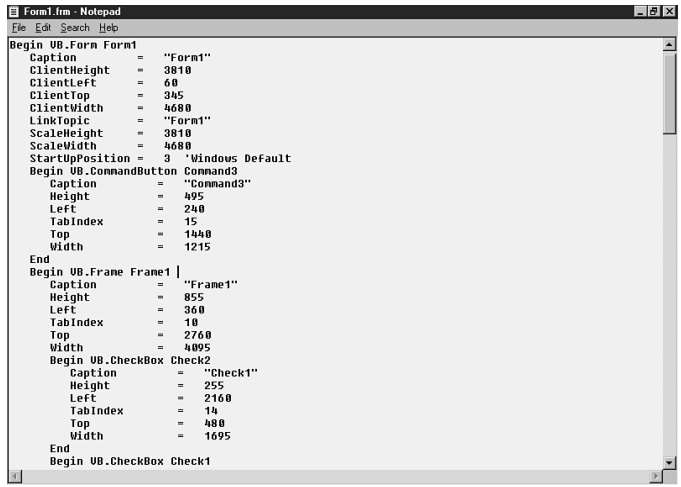
The relationship between `Show` and `Hide` methods and the Loading and Unloading of forms is also discussed in Chapter 6, "Writing Code that Processes Data Entered on a Form," in the sections "Show/Hide Methods Versus Load/Unload Statements" and "How Significant Form Methods Affect Form Events." It is also discussed in this chapter in the previous section "Loading and Unloading Forms."

Design-Time Storage of Form Objects

If you know how Form objects are stored, you will be able to move them from one project to another or from one version of VB to another.

After a form has been designed and saved, the information from the form is separated into two components. The first component is a text description of the form itself—all objects contained on the form, all object properties and their values, as well as any code associated with the form and the objects. This text-based information is stored in an ASCII text file with the extension .FRM. Figure 4.21 shows a sample of a text-based .FRM file. Any word processor or text editor can open this file.

FIGURE 4.21
Sample .FRM file containing the form information.



The second file is .FRX. It contains all graphics-related information required by the form. If a `Picture` control or `Image` control is used, and a bitmap graphic is referenced by the control, that image must be stored internally. This information is stored in the separate .FRX file and is in a binary format. The .FRM file will contain references to the .FRX file and a number indicating the image to be referenced for specific controls.

Knowing how the form objects are stored after designing them can be very useful. If a form is taken to another project and doesn't seem to load properly, you could open the text file and determine whether that form uses paths for files that are not the same in the new environment. Often as forms are moved between projects, the references to various controls are not always the same. You can use a text editor to open the file and verify the references that should be included within that project.

Another reason that you may need to look at the form's text file would be that of an old form from a previous version of Visual Basic being used within a newer version. By opening the text file with an editor, any problem lines can be removed.

The same technique is probably even more useful when taking Form objects from a newer version of VB and going to an older version of the software.

This section has covered how forms are brought into memory and removed from memory. If you need to display a form, use the `Show` method. Use the `Hide` method to make a form invisible. The storage of the design-time form is separated into a text-description file with the extension .FRM. Graphics information is stored in an .FRX file. Both files together implement the form.

Using the Forms Collection

This section discusses the use of VB's Forms Collection. Topics covered include using methods and properties of the collection, referencing individual forms, looping through the collection, verifying loaded forms, and unloading forms from the collection.

In Visual Basic, a collection is just a group of objects. If you have more than one form in a project, you would have a Forms Collection that you can use to easily manipulate all forms or just one specific form.The Forms Collection is a built-in or intrinsic collection of VB. Another example of a collection built into VB is the Controls Collection that will return all the controls contained on the specified form.

The Forms Collection contains the loaded forms of all VB form types. MDI Parent forms, MDI Child forms, and non-MDI forms are all included in the collection, along with any forms created at runtime.

The Forms Collection only contains the forms that are currently loaded in memory. If a project contains multiple forms and only the `Startup` form has been loaded, the collection will only have one item. To be included in the Forms Collection, all forms must be loaded. They do not have to be shown onscreen, only loaded.

If a form is the unloaded from memory, that form is no longer part of the Forms Collection. The collection will reduce its count by one, and that form will no longer be available in the collection.

Using Methods and Properties of the Forms Collection

Most VB collections have various methods or properties that will return information regarding the collection of objects. The Forms Collection has only one property, `Count`. This property gives the total number of forms currently loaded by the application.

To find the total number of forms in the project, just use code such as the following:

```
iFormCount = Forms.Count
```

To test the `Forms.Count` property in VB, use a single form and place one command button on the form. Open the code window for the command button and type the code shown in Listing 4.14.

LISTING 4.14

PROGRAMMING WITH THE FORMS.COUNT PROPERTY

```
Sub Command1_Click()
    Msgbox "The Forms Collection has " &  _
            Forms.Count &  _
            " forms currently",  _
            vbInformation, "Forms Count"
End Sub
```

This code will show a message box with a string that contains the number of forms in the collection. Remember that only loaded forms are in the collection. Any form that has not been loaded or that has been unloaded will not be part of the collection.

Using Specific Items Within the Forms Collection

Each object in the collection can also be referred to by its specific ordinal number. This is an item number automatically assigned as the new items are added to the collection.

When referring to the ordinal number of a form, you must always remember that the Forms Collection is 0-based. The first item in the collection will always have the ordinal value of 0. The following lines demonstrate using the item value:

```
Forms(0).Caption
Forms(1).Name
Forms(2).Width
```

This code assumes that a VB project has three forms and that all forms have been loaded into memory. The first line will return the `Caption` of the very first item (item 0) in the collection. The second line will return the `Name` property of `Form(1)`. The third line will return the value of the last form's width.

These code examples show how easy referencing specific items in the collection can be. This can be an alternative method to using the specific form names in a given project. With the Forms Collection, you also do not have to know the name of a given form to control it. This will assist in manipulating forms generated at runtime.

Looping Through the Forms Collection

Another way to use the Forms Collection is with various looping techniques. By using `For...Next`, `For...Each`, or `Do...While` statements, the programmer can loop through the collection to affect all the forms, search for specific forms, and search for specific properties or even values. This will assist in searching for information without having to program every form name individually.

A simple example would be to retrieve each form's `Caption`, as shown in Listing 4.15.

LISTING 4.15

LOOPING THROUGH THE FORMS COLLECTION WITH FOR...NEXT

```
Dim iLoop as Integer
    For iLoop = 0 to Forms.Count - 1
        MsgBox Forms(iLoop).Caption
    Next iLoop
```

This code declares an integer variable for looping and then sets up the `For...Next` statement. Notice `iLoop = 0`. You must start at 0 because collections are 0-based. Because you will not always know the amount of forms in the collection, you can take the `Forms.Count` and subtract 1 from the total. You must remove 1 because the `Count` starts at 1 and the `Collection` starts at 0. When this sample code is run, a message box displays with the `Caption` property of every form in the collection.

An alternative to the `For... Next` loop would be the `For...Each` loop, which does not depend on an integer counter (see Listing 4.16).

LISTING 4.16

LOOPING THROUGH THE FORMS COLLECTION WITH FOR...EACH

```
Dim frmCurr as Form
    For each frmCurr in Forms
        MsgBox FrmCurr.Caption
    Next frmCurr
```

This is probably preferable to the `For...Next` loop because you don't have to keep track of position in the collection.

Using the Forms Collection to Verify Whether a Form Is Loaded

When a form is referenced in code, it automatically loads the form into memory. An advantage of the Forms Collection is that it contains only forms that have been loaded into memory. This enables the programmer to test for a specific form by looping through the collection and testing the `Name` property of every item in the collection.

An example of this code is found in Listing 4.17.

LISTING 4.17

VERIFYING WHETHER A PARTICULAR FORM IS LOADED

```
Sub Command1_Click()
        Dim frmCurr as Form
        For each frmCurr in Forms
```

```
        If frmCurr.Name = "Form3" Then
            MsgBox "Form3 has been loaded"
        End If
    Next frmCurr
End Sub
```

This code declares a form variable for looping and then sets up the `For...Each` statement. Inside the `For...Each` statement is an `If` test. With this code you are looking through each item in the collection and testing the property `frmCurr.Name`. This allows frmCurr to represent every form that is loaded in the collection. If the `Name` property is equal to `Form3`, a message box displays informing us that `Form3` has been loaded into memory.

This code sample requires a project with at least three forms. The default form names are expected, and the `Form_Load` event is responsible for loading all forms into memory. Remember that if the forms are not loaded at the moment the code runs, they will not be included in the collection.

Using the Forms Collection to Unload All Forms

Another common use of the Forms Collection is to unload all forms.

Although the `End` statement would provide a faster, simpler way to terminate the application, it can have unwanted effects because a call to `End` will immediately terminate the application without allowing any further events (such as `Unload` events) to run. Looping through the Forms Collection to unload all members might seem like a more complicated way to do the task, but it allows normal processing of `Unload` and other events.

The code of Listing 4.18 shows one way to control the loop. First, you must determine the total number of elements in the collection then reduce that number by one to accommodate for that fact that the collection is 0-based. The loop unloads collection members in reverse order from the highest-numbered member to the lowest. This is due to the fact that, after each unload of a collection member, the number of members in the collection decreases by one.

LISTING 4.18

Unloading All Forms With a Descending For...Next Loop

```
Sub cmdClose_Click()
        Dim iLoop As Integer
        Dim iHighestForm as integer
        iHighestForm = Forms.Count — 1
        For iLoop = iHighestForm To 0 Step — 1
            Unload Forms(iLoop)
        Next iLoop
    End Sub
```

A more elegant way to unload all forms would be with a `For...Each` loop, as in the example of Listing 4.19.

LISTING 4.19

Unloading All Forms With a For...Each Loop

```
Sub cmdClose_Click()
        Dim frmCurr as Form
        For each frmCurr in Forms
            Unload frmCurr
        Next frmCurr
    End Sub
```

Both of the above methods assume that at least one form has been loaded in the collection and that a command button named `cmdClose` is on one of the loaded forms. After run, all forms should be unloaded, and the project will terminate if no other forms are in memory.

CHAPTER SUMMARY

This chapter covered the following topics:

- Adding an ActiveX control to the ToolBox
- Programming the `ImageList` control
- Programming the `TreeView` control
- Programming the `ListView` control
- Programming the `ToolBar` control
- Programming the `StatusBar` control
- Using the Controls Collection
- Adding and deleting controls with control arrays
- Adding and deleting controls with the Controls Collection
- Managing visibility and loading of forms
- Using the Forms Collection

KEY TERMS

- ActiveX control
- BMP
- Controls Collection
- Collection
- Forms Collection
- ICO
- Intrinsic control
- OCX
- Pixel
- Prog ID
- ToolBox
- ToolTip
- Twip

APPLY YOUR KNOWLEDGE

Exercises

4.1 Using the `ListView` and `ImageList` Controls

In this exercise, you use a `ListView` and two `ImageList` controls to display a list of items. The `ImageList` controls will contain images that will be shown in the `ListView`. You will bind the controls together and allow a user to add `ListItems` to the form.

You will put the `ImageList` controls on a separate form because you will use that form in projects for several of the other exercises.

As a reference, use sections "The `ImageList` Control" and "The `ListView` Control."

Estimated Time: 30 minutes

1. Start a new project in Visual Basic and create a form named frmListView like the one in Figure 4.22. Add the Microsoft Windows Common Controls Library 6.0 (comctl32.ocx) to the Toolbox using the Project Components menu option. The form will contain two `ImageList` controls, a `ListView` (with a default name of ListView1), `OptionButtons` that will be used to change the appearance of the list, and text boxes for the user to enter the `Key` and `Index` values of `ListImages`. Rename the two `ImageList` controls on the Form to `imglstNormal` and `imglstSmall`.

2. Add images to the `ImageList` controls. Right-click on the `ImageList` control named `imglstNormal` and select `Properties`. Choose 32×32 and then click on the Images tab. Now insert several images into the `ImageList` and also be sure to enter in some unique text for the Key property of each image. Choose icon (*.ico) files from either the Visual Basic graphics directories or by selecting any other *.ico files from your PC. This `ImageList` will contain the regular icons for `ListView1` (see Figure 4.23).

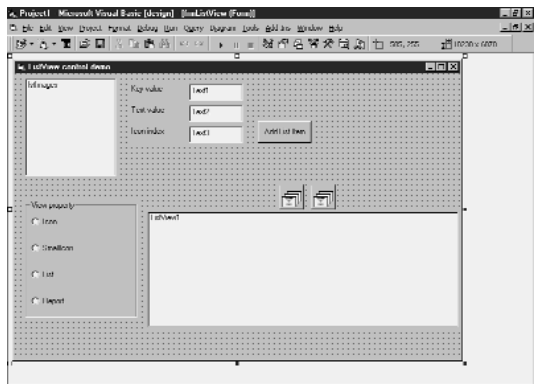


FIGURE 4.22▲
Form and controls for Project 4.1.

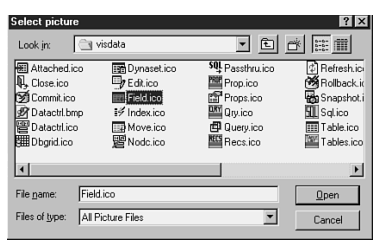


FIGURE 4.23▲
Manipulating `imglstNormal` at design time.

3. For the `ImageList` control named `imglstSmall` (for small icons), choose 16×16 as a size but don't insert any images manually. Instead add code to the `Form_Load` event procedure that will traverse all the `ListImages` in the first control, imglstNormal, and, for each control encountered, will add a new `ListImage` to `imglstSmall`, making sure to use the same `Picture` and `Key` properties from the first `ImageList`.

APPLY YOUR KNOWLEDGE

4. Bind `ListView1` to the `ImageLists`. Bring up the Custom Properties window for `ListView1`. Click on the Image Lists tab. Select `imglstNormal` for the normal icons and `imglstSmall` for small icons, as in Figure 4.24.

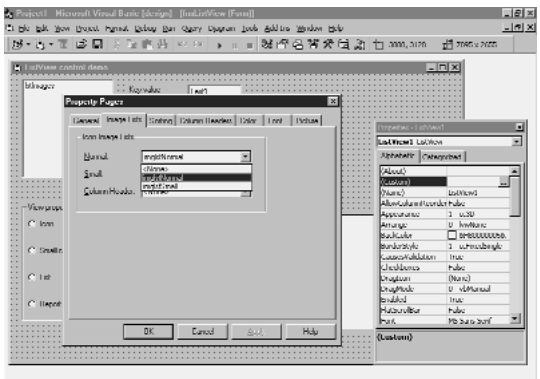


FIGURE 4.24
Manipulating the `ListView`'s design-time properties.

5. Add code to the `OptionButtons` that will change the appearance of the `ListView` between `Normal`, `Small Icons`, `List`, and `Report` by setting the `View` property of `ListView1`.
6. Use text boxes to allow the user to enter the Index value for a specific `ListImage` of the `ImageList`.
7. Place code in the Add button that will insert an item into `ListView1` based on the contents of the three `TextBox` controls.
8. Use the `Add` method of the `ListItems` Collection to do this. Use the input from the form to set the `Key`, `Text`, `Normal Icon`, and `Small Icon` properties.
9. Run and test the application.
10. If you want to put more information in the `ListView` for each item, add ColumnHeaders to the `ListView` from the Custom Properties dialog box and add extra TextBoxes to the form.

4.2 Using a `TreeView` Control

In this exercise, you create a program to execute basic operations on a `TreeView` control.

This exercise covers information from the section "The `TreeView` Control."

Estimated Time: 30 minutes

1. Start a new project in Visual Basic and create a form named frmTreeView like the one in Figure 4.25. Add the Microsoft Windows Common Controls Library 6.0 (comctl32.ocx) to the Toolbox using the Project, Components menu option. The form will contain: an `ImageList` control; a `TreeView` control; `OptionButtons` that will be used to determine the way a new node will be inserted; three text boxes for the user to enter the `Key`, `Text`, and `Index` values of the `ListItems` (and three Labels to the left of each to explain their respective contents); and Labels that will hold information about the last selected `TreeView` node.
2. Assign icons to the `ImageList` control and make the `TreeView` point to the `ImageList` by setting its `ImageList` property.
3. Name one of the Labels `lblSelectedNodeIndex` and another `lblSelectedNodeText`. These labels will hold the Text and Index of the Node that will be used as a reference point when new Nodes are inserted.

APPLY YOUR KNOWLEDGE

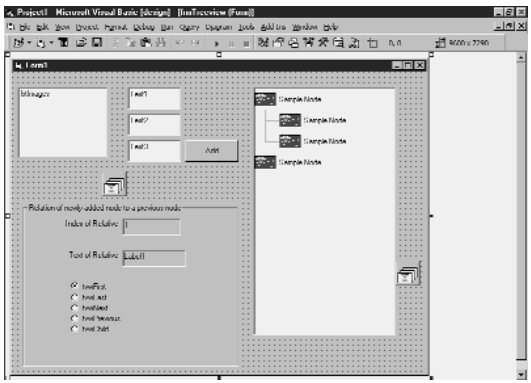


FIGURE 4.25▲
The form for Exercise 4.2.

4. In the `NodeClick` event of the `TreeView` control, set the `lblSelectedNodeIndex.Caption` property to be the Node's `Index` property, and set `lblSelectedNodeText.Caption` property to be the Node's `Text` property.

5. In the `Click` event of the `CommandButton`, first check to see which `OptionButton` is checked to determine which option to use as the `Relation` argument to the `Nodes` collection's `Add` method. If there are no nodes already in the `TreeView`, or if `lblSelectedNodeIndex` contains a non-numeric value, add a node without specifying the first two parameters. If there are already nodes in the `TreeView` parameter, let the first parameter be the value in `lblSelectedNodeIndex` and the second parameter be the option specified by the `Relation` `Optionbuttons`.

6. Run and test the application.

4.3 Creating a Toolbar

In this exercise, you create a toolbar, add buttons, and change settings to see how a toolbar appears and performs at runtime. You will be able to standardize buttons, use button groups, and add separators to space out buttons on the toolbar.

This exercise covers information from the section "The `ToolBar` Control."

Estimated Time: 20 minutes

1. Start a new project in Visual Basic with a single default startup form, as in Figure 4.26. Add the Microsoft Windows Common Controls Library 6.0 (comctl32.ocx) to the Toolbox using the Project, Components menu option. Add an `ImageList` and a toolbar to the form.

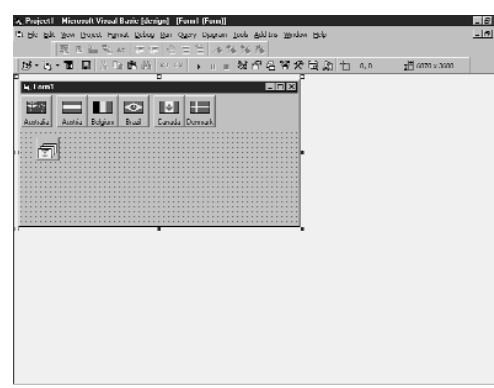


FIGURE 4.26▲
The form for Exercise 4.3.

2. Add some icons to the `ImageList` control with the Custom Properties dialog box (see the section "The `ImageList` Control" for help and also see Exercise 4.1). You can select any icons that you have available.

APPLY YOUR KNOWLEDGE

3. Open the Property Pages dialog box for the toolbar. On the General tab, set a reference to your `ImageList`.
4. Go to the Buttons tab and add a button to the control. Set `Style` to be the `Default`. Enter an image number and add any text you wish.
5. Add another button to the control and set `Style` to be a `Separator`. Then add three more buttons and set their `Style` to `ButtonGroup`. To the three buttons in the group, add images and text.
6. Add another separator button and then add two more buttons. For these two, set `Style` to `Check`. Again add images and text to the buttons.
7. Run the application. Click on the buttons one at a time to see how they perform. Click on the buttons in the group. Can you select more than one at a time? Click the button that is a Check button. Does it stay depressed after it is clicked? Click on it again.

4.4 Creating a `StatusBar`

In this exercise, you create a `StatusBar` control in an application, as discussed in the section "Using the `StatusBar` Control."

Estimated Time: 20 minutes

1. Start a new project in Visual Basic with a single default startup form, as in Figure 4.27. Add the Microsoft Windows Common Controls Library 6.0 (comctl32.ocx) to the Toolbox using the Project, Components menu option. Add a `StatusBar` to the form. Also add a `Label`, a `TextBox`, and two `CommandButtons` to the form and name them, respectively, `lblPanelText`, `txtPanelText`, `cmdAddPanel`, and `cmdClearPanels`.

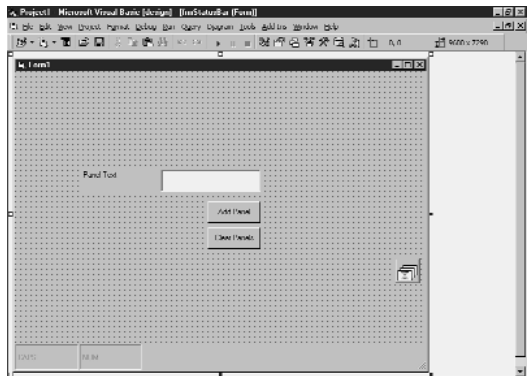


FIGURE 4.27
The form for Exercise 4.4.

2. Call up the Custom Properties dialog box of the `StatusBar` and add several panels. Experiment with the various properties and run the application to test them.
3. Add the following code to the form replacing anything that might already be there. Run and test the project.

```
Option Explicit

Private Sub cmdAddPanel_Click()
    If txtPanelText.Text <> "" Then
        Dim pnl As Panel
        Set pnl = StatusBar1.Panels.Add(,
➥txtPanelText, txtPanelText, sbrContents)
        pnl.Style = sbrText
        pnl.MinWidth = 100
        pnl.AutoSize = sbrContents
        pnl.Enabled = True
    End If
End Sub

Private Sub cmdClearPanels_Click()
   StatusBar1.Panels.Clear
End Sub

Private Sub StatusBar1_PanelClick(ByVal Panel
➥As ComctlLib.Panel)
    Panel.Enabled = Not Panel.Enabled
End Sub
```

APPLY YOUR KNOWLEDGE

4.5 Using the Controls Collection

In this exercise, you use the Controls Collection of a form to clear input fields. You will code a `For Each` loop to cycle through all the controls on a form, and you will use the `If TypeOf` statement to clear the value of the controls on the form.

Refer to the section "Using the Controls Collection" for help with this exercise.

Estimated Time: 20 minutes

1. Open a new project in Visual Basic. Add several input controls including `TextBoxes`, `CheckBoxes`, `OptionButtons`, and `ComboBoxes`, as in Figure 4.28. Add at least one item to each `ComboBox`.

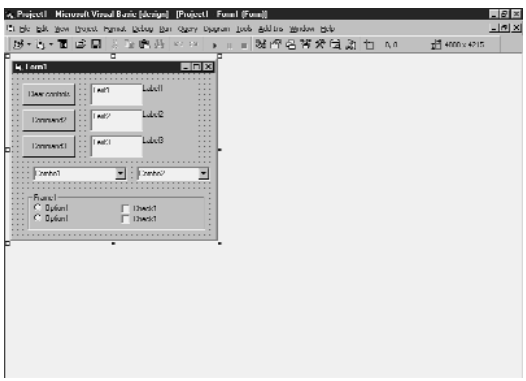


FIGURE 4.28
The form for Exercise 4.5.

2. Add a command button to the form. Add code to the command button that will loop through all the controls on the form, see "Using the Controls Collection," and clear each input control that was added in step 1.

   Use a `For Each` loop to navigate through the Controls Collection. Be sure to check the class of each control by using the `If TypeOf` statement to make sure you are clearing each one properly.

3. Run the application. Fill in all the input fields. Select values from the `OptionButtons` and `ComboBoxes`. Check the `CheckBoxes`.

4. Click on the Clear button. Did all the controls clear? Did they clear without any runtime errors?

4.6 Loading and Unloading Forms

In this exercise, you create a two-form project that will load and unload forms. The `Forms Count` property will also be used to show the forms that are loaded. This exercise will also show how unloaded form code stays resident in memory.

Estimated Time: 45 minutes

To create this project, follow these steps:

1. Start Visual Basic 6.
2. Create a Standard EXE project with its default startup form.
3. Add a second form to the project.
4. On `Form1`, create five command buttons as shown in Figure 4.29.
5. Change the `Command1` button caption to `Show Form2`.
6. Open the Code window for the `Show Form2` button on `Form1` and enter the following code:

```
Sub Command1_Click()
    Form2.Show
End Sub
```

7. Change the `Command2` button caption to `Count Forms`.

APPLY YOUR KNOWLEDGE

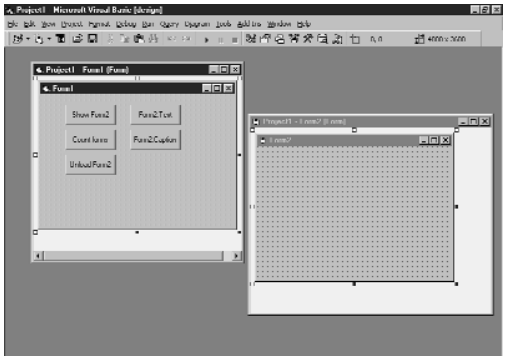


FIGURE 4.29
The forms for Exercise 4.6.

8. Open the Code window for the `Count Forms` button on `Form1` and enter the following code:

```
Sub Command2_Click()
    MsgBox "The Forms Collection contains " _
         & Forms.Count & " loaded forms."
    End Sub
```

9. Change the `Command3` button caption to `Unload Form2`.

10. Open the Code window for the `Unload Form2` button on `Form1` and enter the following code:

```
Sub Command3_Click()
    Unload Form2
End Sub
```

11. Change the `Command4` button caption to `Form2.Test`.

12. Open the Code window for the `Form2.Test` button on `Form1` and enter the following code:

```
Sub Command4_Click()
    Form2.Test
End Sub
```

13. Change the `Command5` button caption to `Form2.Caption`.

14. Open the Code window for the `Form2.Caption` button on `Form1` and enter the following code:

```
Sub Command5_Click()
    Form2.Caption = "Hello World"
End Sub7
```

15. Open the Code window for `Form2`. Under the General Declaration section, enter the following code:

```
Public Sub Test()
    Msgbox "Code from Form2"
End Sub
```

16. Run the project.

17. Click on the `Show Form2` button. `Form2` should be displayed with nothing on it.

18. Return to `Form1` and click the `Count Forms` button. The message box should indicate that the Forms Collection contains two loaded Forms.

19. Click on the `Unload Form2` button. `Form2` should be unloaded from memory at this point.

20. Click on the `Count Forms` button. The message box should indicate that the Forms Collection contains one loaded form.

21. Click on the `Form2.Test` button. The message box should indicate the message `Code from Form2`. Remember that `Form2` has been unloaded at this point.

22. Click on the `Count Forms` button to see how many forms are currently loaded.

23. When the message box appears, it indicates that only one form has been loaded. Procedures are not unloaded from forms. It is only programmatic reference to the form's properties that will cause the load to occur.

APPLY YOUR KNOWLEDGE

24. Click on the `Form2.Caption` button. This will reference `Form2`'s `Caption` property and set it to `"Hello World"`.
25. Click on the `Count Forms` button. The message box should indicate that the Forms Collection has two loaded forms although `Form2` is not visible.
26. Close `Form1`. Notice that the VB environment is still in runtime. This is due to `Form2` being loaded but not shown.
27. Press the End button to end the application.

This exercise demonstrated how loading and unloading forms is performed and the effect that unloading has on form-level code. The Forms Collection was used to prove how many forms were present in memory.

4.7 Using the Forms Collection

In this exercise, you use the Forms Collection to return information about the forms that have been loaded.

Estimated Time: 45 minutes

To create this project, use the following steps:

1. Start Visual Basic 6.
2. Create a Standard EXE project with a standard default startup form.
3. Add three additional new forms. There should be a total of four forms, as shown in Figure 4.30.
4. Add four command buttons to `Form1`.
5. Change `Command1`'s button caption to `Count Forms`.
6. Change `Command2`'s button caption to `Show Forms`.
7. Change `Command3`'s button caption to `Set Captions`.

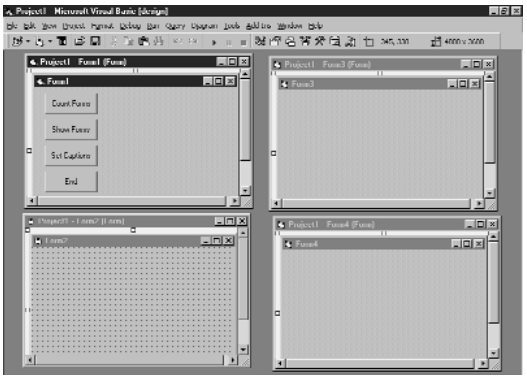


FIGURE 4.30
The forms for Exercise 4.7.

8. Change `Command4`'s button caption to `End`.
9. Open the Code window for `Count Forms` and enter the following code:

```
Sub Command1_Click()
    MsgBox "The Forms Collection contains " _
           & Forms.Count & " loaded forms."
End Sub
```

10. Open the Code window for `Show Forms` and enter the following code:

```
Sub Command2_Click()
    Form2.Show
    Form3.Show
End Sub
```

11. Open the Code window for `Set Captions` and enter the following code:

```
Sub Command3_Click()
    Dim iLoop as Integer
    For iLoop = 0 to Forms.Count — 1
        Forms(iLoop).Caption = "I have set
➥Form" & iLoop & "'s caption."
    Next iLoop
End Sub
```

APPLY YOUR KNOWLEDGE

12. Open the Code window for `End` and enter the following code:

```
Sub Command4_Click()
    Dim frmCurr as Form
    For Each frmCurr in Forms
        Unload frmCurr
    Next frmCurr
    End
End Sub
```

13. Run the project.
14. Move `Form1` to the top-right corner of the screen. This will allow the new forms to be created and not be placed right on top of `Form1`.
15. Click on `Count Forms`. A message box should indicate that the Forms Collection has only one form.
16. Click on `Show Forms`. Forms 2 and 3 should now be loaded and displayed, but Form 4 has not been loaded.
17. Click on `Count Forms`. A message box should indicate that the Forms Collection now has three forms. Form 4 was not loaded and is not part of the collection.
18. Click on `Set Captions`. This should loop through all forms in the collection, starting at item 0 and increasing to the total amount of forms. Each form should now have a caption with its item number as part of the form name.
19. Click on `End` and the keyword `END` will close all forms and end the application.

This exercise demonstrated using the Forms Collection to count the total amount of loaded forms and using an item number to set specific form's properties.

4.8 Adding and Deleting Controls Dynamically With a Control Array

In this exercise you use a control array to dynamically add, manipulate, and remove controls at runtime, as discussed in the section "Adding and Deleting Controls Dynamically Using Control Arrays."

Estimated Time: 45 minutes

To create this project, use the following steps:

1. Create a VB Project and add a second form to the project in addition to the first default form. On the surface of `Form1`, place `CommandButtons`, `cmdAdd` and `cmdRemove`, with appropriate Captions as shown in Figure 4.31.

   In `Form1`'s `Load` event procedure, write the line

   `Form2.Show`

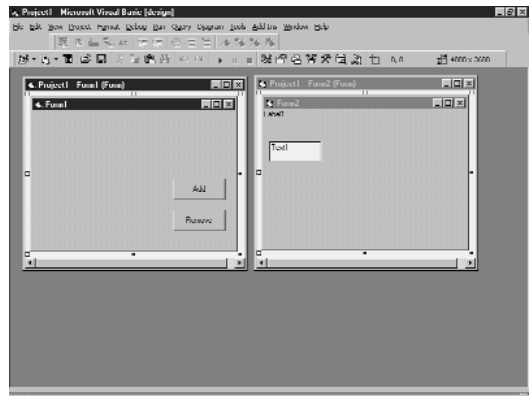


FIGURE 4.31
The forms for Exercise 4.8.

APPLY YOUR KNOWLEDGE

2. On Form2's surface place a `Label` and a `TextBox` control. Change the `Index` property of each control to 0 and the `Visible` property of each control to False.

3. In Form1's General Declarations section, declare a `Private` variable to keep track of the number of elements in the control array:

```
Option Explicit
Private iSequence As Integer  'keep track of
➥ctrls created
```

4. In cmdAdd's `Click` event procedure, put code to load and re-position a `Label` and `TextBox` control array element, augmenting the form-wide variable to keep track of the number of elements loaded:

```
Private Sub CmdAdd_Click()
  Static lNewTop As Long       'top for next
➥ctrl
  Static lNewLeft As Long      'left side of
➥next ctrl
  Static lMaxWidth As Long     'keep track of
➥widest ctrl so far
  iSequence = iSequence + 1
  Dim sCaption As String

  'Add new controls to the control arrays
  Load Form2.Label1(iSequence)
  Load Form2.Text1(iSequence)

  'Top & left are next available positions
➥for label
  Form2.Label1(iSequence).Top = lNewTop
  Form2.Label1(iSequence).Left = lNewLeft

  'Position matching textbox in relation to
➥label
  Form2.Text1(iSequence).Left = lNewLeft + _
  Form2.Label1(iSequence).Width
  Form2.Text1(iSequence).Top = lNewTop

  'form unique label from sequence #
  sCaption = "Control" & iSequence
  Form2.Label1(iSequence).Caption = sCaption

  'make visible
  Form2.Label1(iSequence).Visible = True
  Form2.Text1(iSequence).Visible = True

  'Adjust top for next label-text pair
  lNewTop = lNewTop +
➥Form2.Label1(iSequence).Height

  'But if it will be past the form
  If lNewTop > Form2.ScaleHeight Then

    'Put next label-text pair at top of form
    lNewTop = 0

    'and put its left side to right of
➥current
    'column of controls
    lNewLeft = Form2.Label1(iSequence).Width
➥+ _
    Form2.Text1(iSequence).Width + lNewLeft
  End If

  'No more changes if form is full
  If lNewLeft > Form2.ScaleWidth Then
    cmdAdd.Enabled = False

  End If
End Sub
```

5. In cmdRemove's `Click` event procedure, put code to unload the highest elements in the Label and `TextBox` control arrays and decrement the number of elements:

```
Private Sub cmdRemove_Click()
    If iSequence < 1 Then Exit Sub
    Unload Form2.Label1(iSequence)
    Unload Form2.Text1(iSequence)
    iSequence = iSequence - 1
End Sub
```

6. Run the application to test the loading of control array elements. `Form2` should look like Figure 4.32. Then click the Remove button to repeatedly remove control array elements.

APPLY YOUR KNOWLEDGE

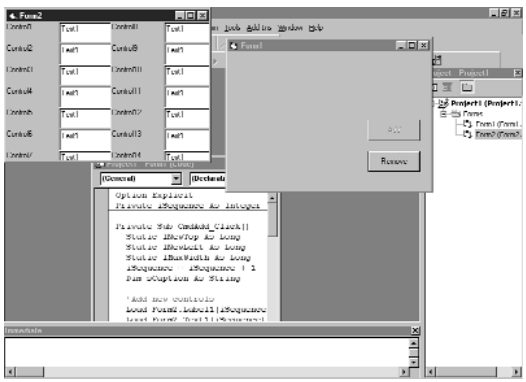


FIGURE 4.32▲
The runtime appearance of the second form in Exercise 4.8 after it has been filled with control array elements.

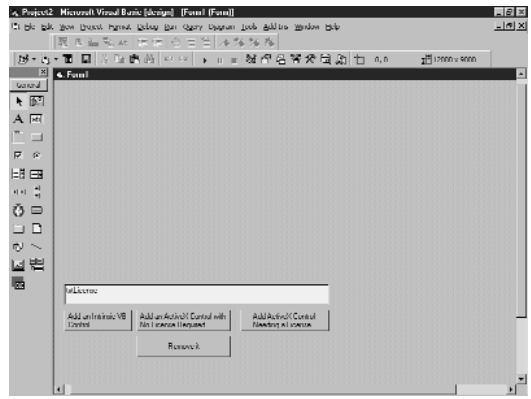


FIGURE 4.33▲
The form for Exercise 4.9.

4.9 Adding and Deleting Controls Dynamically With the Controls Collection

In this exercise, you use the Controls Collection to dynamically add, manipulate, and remove controls at runtime, as discussed in the section "Adding and Deleting Controls Dynamically Using the Controls Collection."

Estimated Time: 45 minutes

To create this project, use the following steps:

1. Add a `CommandButton` to the form named cmdAddIntrinsic, and set its Caption to "Add an Intrinsic VB Control" (see Figure 4.33). Declare a form-wide variable of type `CommandButton` and add code to `cmdAddIntrinsicVB_Click()` as follows:

```
Option Explicit
Dim WithEvents cmdbutton As CommandButton
```

```
Private Sub cmdAddIntrinsicVB_Click()
    Set cmdbutton = Controls.Add
➥("VB.CommandButton", "CmdButton")
    cmdbutton.Caption = "XXXXX"
    cmdbutton.Left = 500
    cmdbutton.Top = 500
    cmdbutton.Visible = True
    cmdAddIntrinsicVB.Enabled = False
End Sub
```

2. Since you declared cmdButton using `WithEvents`, you can navigate to it in the Object list of a Code window. Do so and open its `Click` event procedure, adding the following test code:

```
Private Sub cmdbutton_Click()
    MsgBox "XXXXX"
End Sub
```

3. Add two more declarations to the General Declarations of this form, as follows:

```
Dim WithEvents mscCalendar As
➥VBControlExtender
Dim dt As Date
```

APPLY YOUR KNOWLEDGE

4. Add two more `CommandButtons` named `cmdAddAXNoLicense` and `cmdRemoveAXNoLicense` to the form, setting their captions as shown in Figure 4.33. In their `Click` event procedures, enter the following code:

```
Private Sub cmdAddAXNoLicense_Click()
    Dim ctrl As Object
    Dim strControlType As String
    strControlType = "MSCal.Calendar.7"
    Set mscCalendar = Controls.Add
➥(strControlType, "mscCalendar")
    mscCalendar.Top = 600
    mscCalendar.Left = 500
    mscCalendar.Visible = True
    DoEvents
    dt = DateSerial(mscCalendar.object.Year,
➥_
                mscCalendar.object.Month,
➥mscCalendar.object.Day)
    cmdRemoveAXNoLicense.Enabled = True
    cmdAddAXNoLicense.Enabled = False
End Sub

Private Sub cmdRemoveAXNoLicense_Click()
    Controls.Remove mscCalendar
    Set mscCalendar = Nothing
    Me.Cls
    Me.Print dt
    CmdRemoveAXNoLicense.Enabled = False
End Sub
```

5. In the Make tab of the Project Properties dialog, check the box labeled "Remove information about unused ActiveX controls," as shown previously in Figure 4.18.

   In the Project Components dialog, find the component for `MSCalendar` and check it to include it in the Toolbox.

6. Run the application and note the error message when you click `cmdAddAXNoLicense` as illustrated previously in Figure 4.17. To remedy this problem, stop the application and choose the Project Properties option from the Toolbox.

   In the Make tab of the Project Properties dialog, uncheck the box labeled "Remove information about unused ActiveX controls," as illustrated previously in Figure 4.18. Re-run the application and note that you're now able to add and remove an instance of the `Calendar` control.

7. In the Code window for the form, find the object `mscCalendar`, and then find that object's `ObjectEvent` event procedure. In the `ObjectEvent` procedure, enter the following code.

```
Private Sub mscCalendar_ObjectEvent(Info As
➥EventInfo)
    If UCase$(Trim$(Info.Name)) =
➥"AFTERUPDATE" Then
        dt = DateSerial(mscCalendar.object.
➥Year,  _
            mscCalendar.object.Month,
➥mscCalendar.object.Day)
    End If
End Sub
```

8. Run the application and note that now the date displayed at the top of the form when the calendar is removed correctly reflects the last date chosen by the user.

9. Add a button to the form named `cmdAddAXNeedsLicense` and give it the caption "Add an ActiveX Control Needing a License." Also add a `TextBox` named `txtLicense` to the form (see Figure 4.33). In the `Click` event procedure of `cmdAddAXNeedsLicense`, add the following code:

```
Private Sub cmdAddAXNeedsLicense_Click()
    Dim ctrl As Object
    Dim strControlType As String
    strControlType =
➥"MSFlexGridLib.MSFlexGrid.1"
    txtLicense = Licenses.Add(strControlType)
    Set ctrl = Controls.Add(strControlType,
➥"fgdMY")
    ctrl.Top = 600
    ctrl.Left = 500
    ctrl.Visible = True
    cmdAddAXNeedsLicense.Enabled = False
End Sub
```

APPLY YOUR KNOWLEDGE

10. Run the application and click the new `cmdAddAXNeedsNewLicense CommandButton`, noting that the `TextBox` is filled with license information for the control when you add the MSFlexGrid. If you were to distribute this application to end users, you would need to include the license string in your application as the second argument to `Licenses.Add`. The line would read:

    ```
    Licenses.Add(strControlType, strLicense)
    ```

    where `strLicense` represented the licensing string shown at design time in the TextBox.

11. Comment out the line

    ```
    'txtLicense = Licenses.Add(strControlType)
    ```

    and re-run the application. Notice that this time, you receive an error message complaining that the control needs a license.

Review Questions

1. What VB menu option must you use to add an ActiveX control to the ToolBox?
2. What are the four ways that items in a `ListView` can be displayed?
3. What image formats can be loaded together in an `ImageList` control. For example, can 16×16 icons be loaded with 32×32 icons? Can icons (*.ico files) be loaded with bitmaps (*.bmp) files?
4. What two properties of the `ToolBar` and `Buttons` Collections control the availability of ToolTips?
5. If you include a `Panel` on a `StatusBar` control to display the current time, what is the best way to update that time while the application is running?
6. How many Controls Collections are there in a Visual Basic application?
7. If you intend to dynamically create a `TextBox` control from a control array at runtime, what two things must you do at design time for this to work?
8. When using `Form` objects in a VB project, the `Load` statement is used to bring the form into memory. The `Show` statement is used to display the form on-screen. Are there any other statements that will cause a `Form` object to be loaded into memory?
9. Assume that a project contains a main form and an options form. Assuming that the user can alter the settings on the options form, would you use the `Hide` method or the `Unload` statement to ensure that the options form and its controls could still be referenced after the form is no longer visible?
10. When you create a form at design time and then save it, what are the file format and file extensions that are used to save the information contained on that form?
11. When creating multiple instances of a form at runtime, what is the keyword used to reference the currently running object?
12. What is the keyword used to create a runtime version of a form that was created at design time?
13. The Forms Collection contains references to Visual Basic `Form` objects. Does the collection contain references to forms that were created at design time, runtime, or both?
14. VB `collection` objects usually have various methods and properties. How many properties and methods does the Visual Basic Forms Collection have? What are their names?

APPLY YOUR KNOWLEDGE

15. The `Forms` Collection contains all forms that have been loaded into memory. When programming in VB, you might need to refer to the individual properties of a particular `form`. What does the collection provide to the programmer for the control of individual collection members?
16. What is the role of the `ObjectEvent` in programming with the `Controls.Add` method?
17. Describe the object model of the `ObjectEvent` procedure's `Info` parameter.
18. How can you tell which event was fired for a non-intrinsic control that you've added to the Controls Collection with the `Controls.Add` method?

Exam Questions

1. ActiveX controls are implemented in files with the extension:

   A. OCX

   B. VBX

   C. VBX and OCX

   D. OLB and OCX

2. What three properties affect the sort order of `ListItems` in a `ListView` control? (pick three)

   A. `SortKey`

   B. `SortItem`

   C. `Sorted`

   D. `SortOrder`

3. What image file formats can be loaded into the `ImageList` control? (Select all correct answers.)

   A. *.ico

   B. *.gif

   C. *.jpg

   D. *.bmp

4. If you want to have images displayed on the buttons of a toolbar, how do you add these images to the control?

   A. With the `LoadPicture()` function

   B. From an `ImageList` control

   C. From a *.bmp file

   D. All of these

5. What are the areas of a `StatusBar` control in which information is displayed?

   A. `Node` objects

   B. `Status` objects

   C. `ListItem` objects

   D. `Panel` objects

6. If you are removing controls from your form dynamically (at runtime), what controls can you remove?

   A. Controls you have created dynamically.

   B. Any controls on the form.

   C. Any controls that are part of a control array.

   D. You cannot remove controls dynamically.

7. If you are using a Controls Collection of a form to loop through and clear controls, how do you determine the class of each control?

   A. Using the `Select Case Type` statement.

   B. Checking the `Name` property of each control for standard control prefixes.

APPLY YOUR KNOWLEDGE

C. Using the `If TypeOf` statement.

D. You don't need to check the class of a control before clearing it.

8. What types of objects are contained in an `ImageList` control, and what is the collection that holds these objects?

   A. `Image` objects and the `Images` Collection

   B. `Picture` objects and the `Pictures` Collection

   C. `Bitmap` objects and the `Bitmaps` Collection

   D. `ListImage` objects and the `ListImages` Collection

9. When you add new nodes to a `TreeView` control, you can specify a `Relationship` argument that determines placement of the new node in relation to the `Relative` argument. Which of these is not a valid relationship?

   A. `tvwFirst`

   B. `tvwNext`

   C. `tvwChild`

   D. `tvwAfter`

10. Values of what data type can be used as the key value for an object in the `ListItems` Collection of a `ListView`?

    A. String

    B. Integer

    C. Long

    D. Variant

11. How are columns added to a `ListView` control?

    A. By incrementing the `Columns` property.

    B. By using the `InsertColumn` method.

    C. By adding `ColumnHeader` objects to the `ColumnHeaders` Collection.

    D. Columns cannot be added to a `ListView` control.

12. Which property of a `Button` object in a toolbar will determine the way in which a button performs (that is as a separator, part of a group, check, and so forth)?

    A. `Action`

    B. `Style`

    C. `Type`

    D. `Context`

13. If you want to reference all the controls on a form, `Form1`, which two `For` statements enable you to do this?

    A. `For I = 1 to Form1.Controls.Count`

    B. `For I = 0 to Form1.Controls.Count - 1`

    C. `For Each obj in Form1.Controls`

    D. `For Each obj in Form1.Controls.Count`

14. In the following code, which statement best describes the order of events that happen after this line of code is executed. Assume a VB project with two forms, `Form1` and `Form2`. `Form1` is the `startup` object.

```
Sub Command1_Click()
    Form2.Caption = "Visual Basic Notes"
End Sub
```

    A. When executed, the code will generate `Compile Error: Variable not defined.`

    B. `Form2` is automatically loaded, the caption is set, and `Form2` is shown.

APPLY YOUR KNOWLEDGE

C. `Form2` is automatically loaded, the caption is set, and `Form2` is not shown.

D. `Form2` is automatically loaded, the caption is set, and `Form2` is automatically unloaded.

E. When executed, the code will generate `Compile Error: Method or data member not found.`

15. Visual Basic allows a form to be loaded into memory without directly displaying the form. What statement is used to load a form without showing it?

A. `Load`

B. `Unload`

C. `Show`

D. `Hide`

E. `Dim`

16. In the following code, which statement best describes the order of events that occur while this code is executing. Assume that the project contains two forms, and that this procedure is in the `Load` event of the startup `Form` object.

```
Sub Form_Load()
    frmSplash.Show
    frmMain.Show
    Unload frmSplash
End Sub
```

A. The `Splash` form shows, `Main` form shows, `Splash` form unloads.

B. The `Splash` form loads, `Main` form loads, `Splash` form unloads.

C. The `Splash` form loads and shows, `Main` form loads and shows and causes a compile time error.

D. The `Splash` form loads and shows, `Main` form loads and shows, `Splash` form unloads.

E. The `Splash` form shows, `Main` form shows, `Splash` form causes a compile time error.

17. Visual Basic allows a `Form` object to be removed from the screen, but allows it to stay loaded in memory. This allows other components of the application to refer to controls, properties, and methods of the form. What statement in VB will remove a form from view but keep it in memory?

A. `Load`

B. `Unload`

C. `Dim`

D. `Show`

E. `Hide`

18. `Form` objects can be loaded into memory and used by an application. Using the following code, what type of loading is being performed by VB?

```
Sub Main()
    frmSetup.Show
End Sub
```

A. Forced

B. Manual

C. Implied

D. Inferred

E. Explicit

19. A VB application can define the very first object used when the program is run. This is referred to as the `startUp` object. Where in the VB IDE can the project's `startUp` object be specified?

A. Project, Components

B. Project, Project Properties

C. Project, References

APPLY YOUR KNOWLEDGE

D. Tools, Options

E. Tools, Settings

20. An application can create `Form` objects dynamically at runtime. To create dynamic forms, a `Form` object must be used as the template for the new object. What is the keyword used to create a dynamic runtime form based on another `Form` object?

    A. `Me`

    B. `The`

    C. `Object`

    D. `New`

    E. `Option`

21. From the following code, select all lines that would create a form at runtime based on a design time `Form` object template called `frmMainTemplate`. Assume that the project `startup` object is a `Sub Main` procedure and that this line of code will create the first form.

    A. `Dim frmMainTemplate as frmMainTemplate`

    B. `Dim frmMainTemplate as New frmMainTemplate`

    C. `Set x as frmMainTemplate`

    D. `Dim x as New frmMainTemplate`

    E. `Set x = New frmMainTemplate`

22. In the following code, which statement best describes the order of events that occur while this code is executing. Assume that the project contains two forms and that this procedure is in the `Load` event of the startup `Form` object.

```
Sub Form_Load()
    frmSplash.Show
    frmMain.Show
    Unload frmSplash
End Sub
```

    A. The `Splash` form shows, `Main` form shows, `Splash` form unloads.

    B. The `Splash` form loads, `Main` form loads, `Splash` form unloads.

    C. The `Splash` form loads and shows, `Main` form loads and shows and causes a compile time error.

    D. The `Splash` form loads and shows, `Main` form loads and shows, `Splash` form unloads.

    E. The `Splash` form shows, `Main` form shows, `Splash` form causes a compile time error.

23. When a `Form` object is created dynamically at runtime, all properties, methods, and events will be usable. What keyword can be used to refer to the active object rather than the object name?

    A. `Me`

    B. `Friend`

    C. `You`

    D. `Var`

    E. None of these

24. VB allows a set of similar objects to be grouped together. This allows easier access to the similar objects and also simplifies program code that can affect all members of the group. What is the term used to describe this function grouping?

    A. Friends

    B. Enemies

    C. Procedures

    D. Scopes

    E. Collections

APPLY YOUR KNOWLEDGE

25. Groups of objects simplify control and provide common functions. Which are groups of objects found within Visual Basic?

    A. Variables

    B. Friends

    C. Controls

    D. Variants

    E. Forms

26. When a collection is created in Visual Basic, a method can be used to determine how many objects are in the collection. What default method is used to determine this number?

    A. `Number`

    B. `Item`

    C. `Total`

    D. `Count`

    E. `Base`

27. The following code sample is missing a line of code. This code is to be used to unload all forms from the Forms Collection. Which line of code will complete this `sub` procedure best? Assume that the project contains six forms and that this procedure is in a command button on one of the forms:

```
Sub cmdClose_Click()
    Dim iLoop As Integer
    xxxx
        Unload Forms(iLoop)
    Next iLoop
End Sub
```

    A. `For iLoop = 0 To Forms.Count Step + 1`

    B. `For iLoop = 0 To Forms.Count - 1`

    C. `For iLoop = Forms.Count To 0`

    D. `For iLoop = Forms.Count - 1 To 0 Step - 1`

    E. `For iLoop = Forms.Count To 0 Step - 1`

28. On the Project Properties Make tab, the option Remove Information about unused ActiveX Controls

    A. Should be unchecked to prevent a runtime error when adding an element to the Controls Collection, if that control type is available in the Toolbox but has no design time instance on the `form`.

    B. Should be checked to prevent a runtime error when adding an element to the Controls collection, if that control type is available in the Toolbox but has no design time instance on the form.

    C. Should be unchecked to prevent a runtime error when adding an element to the Controls collection, if that control type is not available in the Toolbox.

    D. Should be checked to prevent a runtime error when adding an element to the Controls collection, if that control type is not available in the Toolbox.

29. When you use the `Controls.Add` method, you must specify the `ProgID` for (pick the best answer)

    A. Any ActiveX control type that is not included in the Toolbox

    B. Any ActiveX control type

    C. All control types

    D. Intrinsic control types

30. If `ctrl` is an object variable representing an ActiveX control object that has been added to the Controls collection, and `Viscosity` is a custom property of that object, which two lines will cause an error?

APPLY YOUR KNOWLEDGE

A. `ctrl.Object.Visible = True`

B. `ctrl.Object.Viscosity = 30`

C. `ctrl.Visible = True`

D. `ctrl.Viscosity = 30`

31. When you dynamically add controls to the `Controls` collection, you must use the `VBObjectExtender` data type for programming with

    A. Any ActiveX control that is not included in the Toolbox

    B. Any ActiveX control

    C. All controls

    D. Intrinsic controls

32. A license key to use with the `Controls.Add` method (pick all that apply)

    A. Can be most efficiently found by searching the Window Registry.

    B. Is the second argument to the `Licenses.Add` method.

    C. Must be licensed to you in order for you to use it legally with software that you distribute to end users.

    D. Need not be specified on user machines that already have the control licensed.

Answers to Review Questions

1. Project Components. See "Adding an ActiveX Control to the ToolBox."

2. The four styles of the `ListView` are `Icon`, `Small Icon`, `List`, and `Report`. They are set by using the `View` property of the object. See "Using the `ListView` Control."

3. Only a single image format can be used in an `ImageList`. 16×16, 32×32, 48×48 icons, and bitmaps must be loaded into separate `ImageLists`. After an image of one format is loaded, images of different formats cannot be loaded. See "Using the `ImageList` Control."

4. The `ShowTips` property of the toolbar dictates whether ToolTips are displayed. The `ToolTipText` property of the `Button` object on the toolbar identifies the text that will be displayed when the user rests the mouse pointer over a button. See "Using the `ToolBar` Control."

5. There is no need to use code that will update the current time in a `StatusBar` panel. The `StatusBar` itself will keep the time current while the application is running. See "Using the `StatusBar` Control."

6. Each Visual Basic application has one Controls Collection per form. Controls Collections are created and maintained for you automatically. See "Using the Controls Collection."

7. To create text boxes dynamically, you must place at least one text box on your form at design time, and you must also set the `Index` property of that control to 0 to create a control array. See "Adding and Deleting Controls Dynamically Using Control Arrays."

8. The `Load` statement and `Show` statement will both load a `Form` object into memory. `Load` will load the form but not show it. `Show` will automatically load the form and display the form onscreen. Another way to load a form into memory is to simply refer to that form. Any programmatic reference to a Form's intrinsic properties will force VB to load the indicated form. See "Loading and Unloading Forms."

APPLY YOUR KNOWLEDGE

9. The `Unload` method will remove a form from memory. When the form is reloaded, all controls contained on the form will be re-initialized. Therefore the `Hide` method would allow the `Options` form to be set by the user, hide it from the display, and still allow the programmatic reference to the controls as set by the user. See "Showing and Hiding a Form."

10. When forms are created then saved, two files can be generated. The first file is an .FRM ASCII text file that contains information related to the form, the form's objects, properties of the objects, and any source code for those objects. The second file is a binary file that contains graphic information related to the form. If a picture control is used, an .FRX file will contain the graphics information required by the control. See "Design Time Storage of Form Objects."

11. The keyword used to reference the currently running object is `ME`. This enables the programmer to reference multiple instances without having to worry about design time names and runtime object variables. If the design time name were used, that would be the only object that would be affected. See "Removing Runtime Forms."

12. The keyword used to create a runtime version of an object created at design time is `NEW`. An object variable is dimensioned as a `NEW` object. This tells VB to create another instance of the Form object at run time. The following code demonstrates this syntax: `Dim x as New Form1`. See "Creating a Runtime Form."

13. The Forms Collection contains references to forms that are loaded into memory through design time and runtime actions. If a project contains multiple forms but a particular form is not loaded into memory, the Forms Collection will not contain a reference to that form. See "Overview of the Forms Collection."

14. The Visual Basic Forms Collection has only one property, `Count`, and it returns the total number of forms that are currently loaded in memory. See "Using Methods and Properties of the Forms Collection."

15. A VB Collection provides an item number that is an ordinal index value assigned to each individual object as it is added to the collection. You can use the item number to programmatically access individual forms and their associated properties. For instance you could refer to the `Caption` property of the first Form in the collection with the syntax: `Forms(0).Caption`. See "Using Specific Items Within the Forms Collection."

16. `ObjectEvent` is an event procedure for objects of type `VBObjectExtender`. When you add an ActiveX control to a form with `Controls.Add`, you can declare the control to be of type `VBObjectExtender` using the `WithEvents` keyword. You can then program the `ObjectEvent` procedure to react to events raised by the control. See "Declaring an ActiveX Control and Programming Its Events."

17. The ObjectEvent event's single parameter, `Info`, is a complex object variable containing a `Name` property and a Collection of `EventParameter` objects. See "Declaring an ActiveX Control and Programming Its Events."

APPLY YOUR KNOWLEDGE

18. You can tell which event was fired for a non-intrinsic control that was added using `Controls.Add` by looking at the name passed in the `Info` parameter (`Info.Name`).

Answers to Exam Questions

1. **A**. ActiveX controls are implemented in files with the extension OCX. The VBX extension was used for 16-bit controls in older versions of VB. TLB files are Type Library files and don't implement ActiveX controls. For more information see the section "Adding an ActiveX Control to the ToolBox."

2. **A, C, D**. The `Sorted`, `SortKey`, and `SortOrder` properties determine whether and how `ListItems` will be sorted in a `ListView`. For more information see the section "Using the `ListView` Control."

3. **A, D**. Icons (*.ico) and bitmaps (*.bmp) files can be loaded into the `ImageList` control. For more information see the section "Using the `ImageList` Control."

4. **B**. Images used by the `ToolBar` control can only come from an `ImageList` control that has been placed on the same form. For more information see the section "Using the `ToolBar` Control."

5. **D**. Status information is displayed in one or more `Panel` objects on a `StatusBar`. For more information see the section "Using the `StatusBar` Control."

6. **A**. You can only remove controls that you have created dynamically. For more information see the section "Adding and Deleting Controls Dynamically Using Control Arrays."

7. **C**. The `If TypeOf` statement can be used to determine the class of an object. It is important to check the type before referencing any properties or method of an object to avoid runtime errors. For more information see the section "Using the Controls Collection."

8. **D**. An `ImageList` contains `ListImage` objects. `ListImage` objects are referenced through the `ListImages` Collection. For more information see the section "Using the `ImageList` Control."

9. **D**. The valid relationships are `tvwFirst`, `tvwLast`, `tvwNext`, `tvwPrevious`, and `tvwChild`; `tvwAfter` is not a valid relationship. For more information see the section "Using the `TreeView` Control."

10. **A**. Key values of the `ListItems` Collection (and all collections) must be strings. For more information see the section "Using the `ListView` Control."

11. **C**. The number of columns in a `ListView` control is controlled by the number of objects in the `ColumnHeaders` Collection. For more information see the section "Using the `ListView` Control."

12. **B**. The `Style` property controls the behavior of a button. Valid values include `tbrDefault`, `tbrCheck`, `tbrButtonGroup`, `tbrSeparator`, and `tbrPlaceholder`. For more information see the section "Using the `ToolBar` Control."

13. **B,C**. If you use a `For I = ...` statement, the index value of Controls Collection ranges from 0 to n-1, where n is the number of controls on the form. For more information see the section "Using the Controls Collection."

14. **C**. Any reference to a form object will cause an implied load in Visual Basic. After `Form2` has been loaded, the `Caption` property is set.

APPLY YOUR KNOWLEDGE

The form has been referenced, but the `Show` method was not executed. This form will remain loaded but not shown. For more information see the section "Loading and Unloading Forms."

15. **A, D**. The `Load` statement can be used to explicitly load a `Form` object into memory. If the `Hide` method of the form is used, an implied `Load` would happen first followed directly by hiding the form. Both could provide the desired result of loading a form into memory. For more information see the section "Loading and Unloading Forms."

16. **D**. Before a form can be shown it must be loaded into memory. Although the `Load` statement has not been used to explicitly load the form, the `Show` method uses implied loading. This allows the form to be directly shown to the user. The proper order of events would be to load the form and then show the form. After the two forms are shown, the `Splash` form is then unloaded. For more information see the section "Showing and Hiding a Form."

17. **E**. The `Hide` statement can be used to remove a `form` object from the screen view but keep it loaded in memory. The `Unload` statement will remove it from the screen and will also remove it from memory. For more information see the section "Showing and Hiding a Form."

18. **C**. If an explicit `Load` statement is not used before using an object, Visual Basic uses implicit loading. Any programmatic reference to an object will cause that object to be loaded into memory. For more information see the section "Loading and Unloading Forms."

19. **B**. To specify the `startup` object of a VB project, the VB IDE provides the Project, Project Properties dialog boxes. The `startup` object combo box can be found under the General tab. The menu selection Tools, Options is where the `startup` form could be selected in Visual Basic 4.0. For more information see the section "Loading and Unloading Forms."

20. **D**. The keyword `New` is used to create a runtime Form object based on a template form. This instructs VB to create another new object based on the object name following the `New` keyword.

21. **B, D**. When dimensioning an object variable, it can have any name you choose. The key is to Dim `'var'` as `New` object. The `New` keyword will create a runtime object from the ofject name following `New`. Both answers are correct, but both use a different object variable name.

22. **D**. Before a form can be shown, it must be loaded into memory. Although the `Load` statement has not been used to explicitly load the form, the `Show` method uses implied loading. This allows the form to be directly shown to the user. The proper order of program actions would be to load the form and then show the form. After the two forms are shown, the `Splash` form is then unloaded. For more information see the section "Showing and Hiding a Form."

23. **A**. The keyword used to refer to the active runtime object is `ME`. Instead of using the object name, the more generic keyword `ME` can be used to control and manipulate the active object.

APPLY YOUR KNOWLEDGE

`Friends` are a special type of procedure and are not used for object reference. For more information see the section "Removing Runtime Forms."

24. **E**. Visual Basic allows groups of similar objects to be used together in a collection. A `Friend` is a special type of procedure. Procedures are a generic term to classify code. Scope is used to refer to the lifetime of a variable or procedure. These are not related to collections. For more information see the section "Overview of the Forms Collection."

25. **C, E**. Similar objects that are grouped together are referred to as collections. Visual Basic provides the Controls and Forms intrinsic collections to keep the similar objects together. Friends are a type of procedure. For more information see the section "Overview of the Forms Collection."

26. **D**. The method of a collection used to determine how many objects are in the collection is `Count`. As objects are added, the `Count` property is used to keep track of the loaded objects. The `Count` property is included by default in new collections. Other methods can be programmed in, if required. For more information see the section "Using Methods and Properties of the Forms Collection."

27. **D**. `For iLoop = Forms.Count - 1 To 0 Step - 1` would correctly complete the code in the example. The form's `Count` property returns the total number of loaded forms in the collection. However the collection's first item index is 0, which means that the highest index in the Forms collection would be `Forms.Count—1`. Therefore your For loop must start at the value of `Forms.Count—1` and count down to 0. The Step—1 is used to cause the `for` loop count down by increments of one. All other code will produce a subscript out of range error. The Forms and Controls Collections always start at element 0; as items are removed, other items are reorganized to lower numbers. When deleting elements from a collection you should therefore begin at the highest number and work down to the lowest. For more information see the section "Using the Forms Collection to Unload All Forms."

28. **A**. On the Project Properties Make tab, the option Remove Information about unused ActiveX Controls should be unchecked to prevent a runtime error when adding an element to the Controls Collection, if that control type is available in the Toolbox but has no design time instance on the form. B is incorrect because checking the option will cause an error when your code tries to add an instance of the control to the Controls collection. C. and D. are incorrect because there is no problem of this nature when the control does not appear in the Toolbox. See "Keeping a Reference in the Project to an ActiveX Control."

29. **C**. When you use the `Controls.Add` method, you must specify the ProgID for all controls. When specifying the ProgID for most intrinsic controls, the ProgID is a string composed of "VB." plus the programmatic name of the control type (for example "VB.CommandButton" or "VB.Label"). You can find the ProgID for ActiveX controls by looking in vendor documentation or by searching the Windows Registry. See "Getting a Control's ProgID."

APPLY YOUR KNOWLEDGE

30. **A, D**. If `ctrl` is an object variable representing an ActiveX control object that has been added to the Controls collection, and `Viscosity` is acustom property of that object, then you will get an error by running either of the two lines:

    ```
    ctrl.Object.Visible = True
    ```

    or

    ```
    ctrl.Viscosity = 30
    ```

    This is because `Top` is a standard property and should not be accessed through the `Object` property that provides access to Custom Properties. `Viscosity`, on the other hand, is a custom property and so must be referred to through the `Object` property. See "Declaring an ActiveX Control and Programming its Events" and "Adding and Removing a Control in the Controls Collection."

31. **B**. When you dynamically add controls to the `Controls` Collection, you must use the `VBObjectExtender` data type for programming with any ActiveX control. When you program with intrinsic controls, you can use the object model provided for the control by the VB environment. See "Declaring an ActiveX Control and Programming its Events."

32. **B, C**. A license key to use with the Controls.Add method is the second argument to the `Licenses.Add` method and must be licensed to you in order for you to use it legally with software that you distribute to end users. A. is incorrect because, while it is possible to find the license key in the Windows Registry, this is a very tedious method compared to using the return value of `Licenses.Add` when running the application in your design environment. D. is incorrect because you must provide the license key by invoking `Licenses.Add` on all user machines. See "Managing the License for an ActiveX Control."

OBJECTIVE

This chapter helps you prepare for the exam by covering the following objective and its subobjectives:

Write code that validates user input (70-175 and 70-176).

- **Create an application that verifies data entered by a user at the field level and the form level.**
- **Create an application that enables or disables controls based on input in fields.**

▶ *Input validation* is an important part of any computer application that requires user interaction. In its broadest sense, the concept applies to anything that the application does to ensure that data entered by the user is acceptable for the purposes of the application.

Input validation can take place at various times in the data entry cycle. For example, the programmer can:

- Constrain the user's entry of data before it begins by providing very restricted data input fields that permit only valid choices. The most common way to do this is to provide standard controls that do not permit free keyboard entry, such as drop-down lists, option buttons, and check boxes.
- Constrain the user's entry of data at the moment that it occurs by monitoring every keystroke for validity and rejecting unwanted input as it's typed. For instance, a particular entry field might seem to the user to ignore anything but numeric characters.
- React to the user's entry of data after the user is finished, accepting or rejecting the contents of a data field when the user attempts to leave the field or close the screen.

CHAPTER 5

Writing Code that Validates User Input

OBJECTIVE

- Input validation can also have varying degrees of user participation. For instance, the program can
  - Automatically correct a user's mistakes without asking the user's opinion.
  - Warn the user of incorrect input and prompt the user to correct the input before allowing the user to continue with other activities.

The first exam subobjective for this chapter, verifying data entered by a user at the field level and the form level, deals mostly with the immediate validation of user input from the keyboard. In order to cover this objective, we discuss the three main `Keystroke` events, `KeyUp`, `KeyDown`, and `KeyPress`. These events can fire at the level of individual controls and also at the level of the form, thus allowing two levels of validation.

An application also usually validates field-level data when the user finishes entry in each field and attempts to leave the field by setting focus away from it. We also discuss the `Validate` event and `CausesValidation` property (new to VB6) that you can employ in this type of validation.

The second exam sub-objective for this chapter, enabling or disabling controls based on input in fields, typically involves a more global type of cross-validation between data entered in two or more controls. We discuss these techniques in the section entitled "Enabling Controls Based on Input."

OUTLINE

`Keystroke` Events at Field and Form Level 212

The `KeyPress` Event 212
The `KeyDown` and `KeyDown` Events 214
`KeyPress` Versus `KeyDown` and `KeyDown` 216
Enabling Two-Tier Validation With the Form's `KeyPreview` Property 216

Field-Level Validation Techniques 217

The `Validate` Event and `CausesValidation` Property 218
The `Change` Event and `Click` Events 220
An Obsolete Technique: Validation With `GotFocus` and `LostFocus` Events 221

Enabling Controls Based on Input 222

Miscellaneous Properties for Validation 224

`MaxLength` 224
Data-Bound Properties 224

Chapter Summary 227

Study Strategies

- Memorizing the meaning and use of the parameters of the three `Keystroke` events, `KeyPress`, `KeyUp`, and `KeyDown`. See the first half of this chapter as well as Exercise 5.1 and 5.2.
- Getting used to the `Shift` parameter of the `KeyUp` and `KeyDown` events if you're not familiar with programmatic manipulation of bit masks. You should study and manipulate the examples in Exercise 5.1.
- Understanding the difference between the `KeyPress` event on one hand and the `KeyUp` and `KeyDown` events on the other hand. See the section entitled "`KeyPress` Versus `KeyUp` and `KeyDown`."
- Understanding the meaning and use of the form's `KeyPreview` property. See the section entitled "Enabling Two-Tier Validation With the Form's `KeyPreview` Property" and Exercise 5.1.
- Understanding the use of the `CausesValidation` property and the `Validate` event and the relation between these two members. If you've done data validation with earlier versions of VB, be aware that these are new features in VB6. See the section titled "The `Validate` Event and `CausesValidation` Property" for more discussion of how Microsoft may treat this on the exam.
- Understanding techniques for enabling or disabling controls based on user input. See the section in this chapter entitled "Enabling Controls Based on Input" and Exercise 5.3.

Keystroke Events at Field and Form Level

The first line of defense against user error is to catch problems as the user presses each key. The following sections discuss how to use three `Keystroke` events to intercept, interpret, and modify each keystroke.

You can validate keyboard input on two different levels in VB:

- You write validation code in the `Keystroke` events of each separate control. The validation code in a control's `Keystroke` event runs when that control has focus and the user presses keys on the keyboard.
- The form also has the same `Keystroke` events that can intercept all keyboard input while that form is the application's active form. When form-level key handling is active, the form's `Keystroke` events receive the keystroke before the currently-active control's `Keystroke` events receive it. By default, form-level `Keystroke` handling is not enabled. We discuss how to implement form-level `Keystroke` handling in the section "Enabling Two-Tier Validation With the Form's `Keypreview` Property."

The `KeyPress` Event

The `KeyPress` event happens after the `KeyDown` event but before the `KeyUp` event. It detects the ASCII value of the character generated by the pressed key.

The `KeyPress` event procedure's single parameter is `KeyAscii`. `KeyAscii` is an integer representing the ASCII value of the character generated by the user's key press.

For instance, if the user keys an uppercase "A," then the `KeyPress` event fires and the `KeyAscii` parameter will have a value of 65 (since 65 is the ASCII code for uppercase "A").

If you write code in the `KeyPress` event that changes the value of `KeyAscii`, then the system will see the newly assigned character as the character that the user has just typed.

If you change the value of `KeyAscii` to 0, then the system will not see a keystroke, and you have in effect discarded the keystroke.

The preceding discussion implies the following general technique for handling user keyboard input in the `KeyPress` event procedure:

1. Use the `Chr` function to convert `KeyAscii` to a character value.
2. Manipulate or evaluate the character.
3. Use the `Asc` function to convert the changed character back to its corresponding integer value, or determine the desired ASCII value in some other way.
4. Assign the new ASCII value to the `KeyAscii` parameter.

In Listing 5.1, all characters keyed in by the user are converted to lowercase in the following steps:

1. Convert `KeyAscii` to its character equivalent with the `Chr` function.
2. Convert the newly derived character to lowercase.
3. Convert the lowercase character back to an ASCII value with the `Asc` function and reassign the lowercase ASCII value back to `KeyAscii`.

LISTING 5.1

CHANGING THE CASE OF A CHARACTER IN THE KEYPRESS EVENT PROCEDURE

```
Private Sub txtPassword_KeyPress(KeyAscii As Integer)
    Dim KeyChar As String
    KeyChar = Chr$(KeyAscii) 'convert to character
    KeyChar = LCase(KeyChar) 'change to lowercase
    KeyAscii = Asc(KeyChar)  'reassign changed ASCII
End Sub
```

Listing 5.2 uses a similar technique to allow the user to input only digits. The code checks to see whether the user has keyed in a numeric character. If not, the code discards the character by changing the value of `KeyAscii` to `0`.

Notice that the outer branch condition in the listing (`KeyAscii > 31`) allows lower-numbered ASCII characters to pass through. This is necessary because some of the control keys, such as Backspace, generate low ASCII codes (Backspace generates ASCII 8). If the event procedure discarded these characters, then the user would not be able to use Backspace or some other keys.

LISTING 5.2

DISCARDING SELECTED KEYSTROKES IN THE KEYPRESS EVENT PROCEDURE

```
Private Sub txtPassword_KeyPress(KeyAscii As Integer)
    Dim KeyChar As String
    If KeyAscii > 31 Then 'ignore low-ASCII characters like
➥BACKSPACE
        KeyChar = Chr(KeyAscii)
        If Not IsNumeric(KeyChar) Then
            KeyAscii = 0
        End If
    End If
End Sub
```

The `KeyPress` event only fires if the key that is pressed generates an ASCII character. There are many keys on the keyboard that do not generate ASCII characters including all of the function keys and most of the cursor movement keys. To detect those key presses, you must use the `KeyUp` and `KeyDown` events.

The `KeyUp` and `KeyDown` Events

The `KeyDown` and `KeyUp` events happen when the user respectively presses and releases a key on the keyboard. Their event procedures take the following two parameters:

- *`KeyCode`* contains an integer code for the physical key that the user pressed. You can check for a particular key by comparing `KeyCode` with one of the special VB internal constants for physical key codes. Each constant name begins with the string "vbKey" followed by an obvious name for the key (the letter of the key if it's an alphabetic key or some other obvious name for other keys). Examples of `vbKey` constants would be `vbKeyA`, `vbKeyW`, `vbKeyF1`, `vbKeyPgUp`, and so forth.
- *`Shift`* indicates if any of the three shift keys (Alt, Ctrl, or Shift) is pressed at the moment. This parameter works in the same way as the `Shift` parameter for the `MouseDown` and `MouseUp` event procedures. That is, `Shift` is an integer representing a bit mask. You can extract information concerning the state of each of the three control keys by ANDing the `Shift` parameter with one of the three VB constants for the control keys.

In Listing 5.3, the `KeyDown` event procedure is used to detect when the user keys Shift+F10. Note the use of internal VB constants to detect the keystroke and the state of the Shift key.

LISTING 5.3

DETECTING A SHIFTED KEYSTROKE IN THE KEYDOWN EVENT PROCEDURE

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    Dim blnIsShift As Boolean
    blnIsShift = Shift And vbShiftMask
    If blnIsShift And (KeyCode = vbKeyF10) Then
        'take some action for Shift+F10
    End If
End Sub
```

NOTE

How to Discover the Names of VB Keystroke Constants When in design mode, you can see the names of the internal VB keystroke constants by invoking the Object Browser with the F2 key, choosing `All Libraries` or `VBRun` from the `Libraries/Projects` list, and then choosing `KeyCodeConstants` from the `Classes` list, as shown in Figure 5.1.

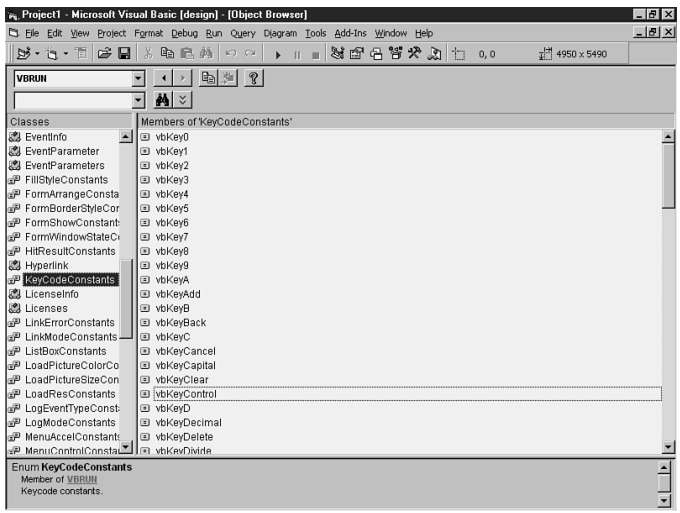


FIGURE 5.1
Finding the names of VB `KeyCode` constants in the Object Browser.

In the event procedure code, you can modify the values of the `KeyCode` and `Shift` parameters to change the keystroke information that the system sees, just as you can modify the `KeyAscii` parameter in the `KeyPress` event procedure.

KeyPress Versus KeyUp and KeyDown

KeyDown and KeyUp don't detect exactly the same information as KeyPress.

KeyPress detects a *character* in its KeyAscii parameter, but KeyDown and KeyUp detect a *physical keystroke* in their KeyCode parameter. While the difference may seem subtle, it's a difference with practical consequences:

- KeyUp and KeyDown can detect keystrokes not recognized by KeyPress, such as function, editing, and navigation keys.
- KeyPress can distinguish between different ASCII characters generated by the same physical keystroke. For instance, KeyPress' KeyAscii parameter gives two different values for "A" and "a." On the other hand, KeyUp and KeyDown's KeyCode parameter gives only one value for these two characters since the physical keystroke is the same (Of course with a little extra logic, you could use the Shift parameter to figure out whether the Shift key were being held). In addition, different country-specific keyboard mappings will generate difference ASCII codes for the same physical key.

NOTE

The Caps Lock Key The Caps Lock key generates its own KeyDown and KeyUp events.

To summarize,

- You should use KeyDown or KeyUp when you need to detect keystrokes that don't necessarily have an ASCII representation, such as the function keys or the arrow keys, or whenever you are interested in the physical keystroke as such.
- When you're interested in the actual character that was generated by the keystroke, you should use KeyPress.

Enabling Two-Tier Validation With the Form's KeyPreview Property

VB forms have KeyPress, KeyUp, and KeyDown events with behavior exactly like that of the same events for individual controls.

By default, a form's three Keystroke events are not enabled. Even if you wrote code in the event procedures of Form_KeyPress,

`Form_KeyUp`, or `Form_KeyDown`, nothing would happen. In order to enable these form events, you must set the form's `KeyPreview` property to `True` (its default value is `False`).

Once you've set the form's `KeyPreview` property to `True` and placed code in its `Keystroke` events, you have enabled *two-tier* keyboard validation. In other words, keyboard information will pass through two successive sets of validation routines: the form's Keystroke routines and then the keystroke routines for the control that currently has the focus.

When `KeyPreview` is `True`, the form's `Keystroke` events will run first. If the form's `Keystroke` event procedures modify the keystroke information, then the control's `Keystroke` events will receive that modified information and not the original keystroke information.

So, for example, if a form's `KeyPress` event procedure changes all characters to uppercase, then no control on the form will ever see a lowercase character in its `KeyPress` event.

Use common sense to decide whether to put keystroke validation code at the form or the control level. If you need some validation to take place for all keystrokes on the entire form (e.g., all characters in all fields need to be uppercase), then that validation should happen in the form-level `Keystroke` event procedures. Control-specific validation should happen in control `Keystroke` event procedures.

Field-level Validation Techniques

Field-level validation (as opposed to form-level validation) has to do with the verification of data entered into an individual field. Typically (but not always), field-level validation is performed without regard to the contents of other fields on the form. The appropriate times to perform field-level validation are:

- As the user enters individual keystrokes into the field (`Keystroke` events as described in previous sections).
- When the user attempts to leave the field—since we presume that the user considers the field entry to be complete (`Validate` event).
- When the field changes for any reason (`Change` event).

The `Validate` Event and `CausesValidation` Property

The `Validate` event and the accompanying `CausesValidate` property are new to VB6 and give programmers a much-needed replacement for older, more cumbersome techniques of field validation. (See the section in this chapter entitled "An Obsolete Technique: Validation With `GotFocus` and `LostFocus` Events".)

EXAM TIP

Don't Let Knowledge of Obsolete Techniques Trip You Up You should be aware that the older validation technique using `GotFocus` and `LostFocus` is no longer recommended in VB6. However, the exam might assume knowledge of the older technique.

The `Validate` Event

In general, the best time to validate a field's contents is when the user attempts to leave the field. The `Validate` event fires whenever the user attempts to set focus to another field on the same form or when the form unloads from memory while the current field has focus. A programmer can evaluate the state of the data at that point and react to any error in the data either by:

- Programmatically correcting the data error.
- Setting the `Validate` event procedure's `Cancel` parameter to `True` in order to prevent focus from leaving the control, thus forcing the user to fix whatever problem was encountered.

You should take the following steps to implement field-level validation with the `Validate` event:

1. Determine which controls (such as `CancelButtons`) should not trigger the `Validate` event when the user tries to set focus to them and set the `CausesValidation` property of these controls to `False` (default is `True`).
2. Write validation code (or call your own validation routines) in the `Validate` event procedure of all controls where you need to have validation in place.
3. If your validation code decides that the control's data is not valid, you can either:
   - Fix the problem right there in the validation code.
   - Force the focus to remain in the current control (presumably so that the user can rectify the problem) by setting the value of the `Validate` event's `Cancel` parameter to `True`.

Listing 5.4 illustrates the use of the `Validate` event procedure to perform validation and to decide whether or not to keep focus on the current control.

LISTING 5.4

USING THE VALIDATE EVENT PROCEDURE

```
Private Sub txtAge_Validate(Cancel As Boolean)
    If Not IsNumeric(txtAge.Text) Then
        Cancel = True
    ElseIf txtAge.Text < 21 Then
        Beep  'give the user some minimal feedback
        MsgBox "Enter an age greater than 21"
        Cancel = True
    'Following is not needed. Placed here for clarity
    Else
        Cancel = False
    End If
End Sub
```

The `CausesValidation` Property

The `CausesValidation` property deserves further explanation as its use can be a bit confusing on first acquaintance.

Imagine that the `Validate` event of a control ran whenever the user attempted to set focus away from the current control to another control on the form.

At first glance this would not seem to be a problem. After all you want to make sure that the control's contents are always validated, don't you?

But consider for a moment the case where the user is currently on a control such as a `TextBox` with incorrect data (a misspelled password, for instance) and has clicked on a Help button in a toolbar or on a Cancel button to abort data entry on this form.

In such a case, do you really want validation to occur? If validation does occur on the `TextBox` control whenever a user clicks the Cancel button, then it's possible that focus will remain on the original control (if there is a validation problem). In that case the Cancel button's `Click` event procedure code will never run, and the user will be permanently trapped with focus on the offending control (until of course the user becomes so frustrated that he or she reboots the machine or uses Task Manager to stop your application).

This is where the `CausesValidation` property comes to the rescue. By default, a control's `CausesValidation` property is `True`, meaning that other controls' `Validate` event will fire when users attempt to set focus to the current control.

You should set `CausesValidation` to `False` on controls where the user should always be able to set focus, regardless of whether there is a problem with data validation on some other control. For example a Cancel button should probably have its `CausesValidation` property set to `False`, because once the user decides to cancel a data entry session, the validity of individual data fields does not matter.

The `Change` Event and `Click` Events

`Change` events (of `TextBoxes`, for instance) or `Click` events (e.g. for `OptionButtons`, `CheckBoxes`, `ComboBoxes`, and `ListBoxes`) happen on many standard VB controls whenever control contents changes for any reason, including:

- Changes caused by user input;
- Changes caused by program code or system actions;
- Changes that happen in the underlying data of a bound control.

These events are most useful when you need to constantly monitor the overall state of data validity on the form, perhaps because you wish to decide whether particular controls (such as an OK button) should be enabled. In order to implement this type of behavior, you should write a general validation routine that checks the validity of the entire form and then call it from the appropriate `Click` or `Change` event procedures.

For more detailed discussion of these techniques, see the section entitled "Enabling controls based on input" and Exercise 5.3.

The `Change` event is less useful for keystroke validation, due to the fact that it fires so often (every time even a single character changes) but has no built-in way to cancel changes. You should leave keystroke validation to the `KeyPress`, `KeyUp`, and `KeyDown` event procedures, as described in previous sections of this chapter.

An Obsolete Technique: Validation With `GotFocus` and `LostFocus` Events

The technique discussed in this section is no longer recommended with VB6. However, this technique can make its appearance on the certification exam in the form of "trick questions" to trip up the unwary programmer with experience in VB5 and earlier versions.

Here we outline the technique without going into details of implementation.

In order to "pull back" the focus to a control that has a validation problem, VB6 uses the `Validate` event and the `CausesValidate` property as discussed in previous sections.

However, in versions of VB before VB6 the `Validate` event and `CausesValidate` property did not exist for standard controls such as the `Textbox`.

In earlier versions of VB, a technique for retaining focus on a control was:

1. Perform a validation check in the `LostFocus` event procedure.
2. If the validation fails, call the control's `SetFocus` method to keep focus on the control.

Although the technique sounds simple enough, there were a number of complicating factors that the programmer had to take into account:

Because the system was between two controls, calling the `SetFocus` method of the control being validated could cause the `LostFocus` event of a second control to fire. What if the second control in turn had similar validation code in its `LostFocus` event procedure? If there were a validation failure in the second control as well, then the second control would also try to "grab" focus back from the first control. The first control in turn would fire its `LostFocus` event a second time, and so on, in an infinite loop. To handle this problem of the dueling `LostFocus` event procedures, it was necessary to put a global flag variable in the application that would be set during a `LostFocus` event procedure's validation. In that way, other controls' validation code could check this flag and would not run if another control's validation were already pending.

EXAM TIP

Trick Questions on Validation With `GotFocus` and `LostFocus` Events
Watch out for questions based on VB5 that assume this is still a valid technique.

A second problem was caused by the fact that there were some controls (such as Cancel `CommandButtons`, for example) that should always be allowed to receive focus from another control on the form even if there were a validation problem. Thus, it would be necessary to put more logic in validation routines to ignore the validation logic if the proposed target control were always allowed to receive focus.

If the above seems less than robust or overly complex to you, you are not alone in your opinion. In the release of VB6, Microsoft heeded the desires of many VB programmers and provided the `Validate` event and the `CausesValidation` property (see the previous sections in this chapter) to put a stop to control validation madness.

The `GotFocus`/`LostFocus` technique just discussed in this section is therefore no longer necessary or recommended.

Enabling Controls Based on Input

A data entry screen often requires some of its controls to be selectively enabled, depending on the state of the rest of the data on the screen. Some examples of such selectively enabled controls might be:

- A drop-down list of credit card types enabled only if the user selects the option button for "credit card" from the option button group for type of payment.
- An OK button. This button signals that the user is finished with data input and that the data can be processed. The button should only be enabled if the data on all other fields is complete and valid.

In the first example above, the control that we wish to enable depends on just one other control or group of controls to be enabled. In the case of the example, it would be sufficient to put a line of code enabling or disabling the `ListBox` in the `Click` event of each type of payment option button as in Listing 5.5.

LISTING 5.5

CODE TO ENABLE A LISTBOX BASED ON A USER'S OPTIONBUTTON CHOICE

```
Private Sub optCash_Click()
    lstCreditCard.Enabled = False
End Sub

Private Sub optCreditCard_Click()
    lstCreditCard.Enabled = True
End Sub

Private Sub optDebitCard_Click()
    lstCreditCard.Enabled = False
End Sub
```

If the control to be enabled is a `TextBox`, you have the choice of two properties for disabling user input: the `Enabled` property and the `Locked` property. The two properties can be compared as follows:

- The `Enabled` property is `True` by default. When `Enabled` is `False`, the user can't set focus to the control and the control or its contents appear grayed out to the user.
- The `Locked` property (`TextBoxes` only) is `False` by default. When `Locked` is `True`, the user can still set focus to the control but can't make changes. The contents don't appear grayed out and the user can scroll through the contents if they are larger than the area of the `TextBox`. The `Locked` property is useful when you can't predict how big the contents of the `TextBox` will be, and you need to prevent user input but still want the user to be able to view all the contents.

In the example of the OK button, however, whether or not the button should be enabled depends on the validity of all the other fields on the screen. In some cases, it might depend on one or more relations between those fields. For instance, if "credit card" is chosen as the form of payment, then "credit card type" must not be left undefined.

In order to allow your application to decide whether or not to enable an OK button, you could follow these steps:

1. Write a Function procedure that performs all necessary data validation checks for the screen and returns a `True` or `False` result.

2. Call this Function from all event procedures where data changes (such as, `Change` event procedures for `TextBoxes` and `Click` event procedures for `OptionButtons`, `CheckBoxes`, and `List` and `Combo` boxes), and set the OK button's `Enabled` property in accordance with the return value of the Function.

Exercise 5.3 illustrates how to implement this type of validation by selectively enabling controls in validation code.

Miscellaneous Properties for Validation

The following properties will save you the work of writing validation code since they will automatically enforce certain data-entry constraints on the user.

MaxLength

The `TextBox` control's `MaxLength` property allows you to specify the maximum number of characters that the user can enter into a `TextBox`. If the user tries to enter more than the maximum allowed, then the input is ignored.

If you set `MaxLength`'s value to 0 (its default value), then there is no maximum length and the user can enter as many characters as desired (up to the 32K limit on the `Text` property's size).

Data-Bound Properties

These properties are only useful when the controls to which they belong are bound to data:

- `DataChanged` is a property that allows the programmer to distinguish between changes made to a bound control's contents by the underlying data engine (caused by data navigation or updates from other users) versus changes made in the application by users or program code.
- `DataFormat` allows the programmer to specify a mask for formatting data when it is retrieved from and stored to the underlying data structure.

CASE STUDY: A SIMPLE DATA ENTRY FORM

NEEDS

Your users need a data entry form that will allow them to input basic sales and marketing information about a customer such as name, age, and form of payment.

The data needs to be validated so that it is logically consistent and complete.

REQUIREMENTS

Implement a way for parts of the application outside the form to tell if all data on the form was successfully entered (typically, a `Public Boolean` variable on the form).

Identify the discrete pieces of information (fields) that the user needs to enter in this form.

Identify the fields that will require free-form keyboard input from the user (`TextBoxes`) and the fields that require data constrained to a few choices or a range of choices (`ComboBoxes`, `ListBoxes`, `OptionButtons`, and `CheckBoxes`).

Identify the form-wide constraints on keystroke input:

- Should all alphabetic characters be uppercase?
- Should certain function-shift key combinations have a consistent meaning for the entire form?

Identify field-level constraints on keystroke input:

- Should certain fields (such as age) accept only numeric input?
- Should certain function-shift key combinations have meaning that is specific to a certain field or fields?

Identify field validation rules and write routines that will handle this validation:

- Numeric ranges
- Constraints on character information such as maximum and minimum length
- Constraints that depend on the contents of other fields.

Call the field validation routines from the appropriate `Validate`, `Click`, or `Change` event procedures.

Identify `CommandButtons` that will allow the user to exit the form and either accept or reject the data. (Typically, you'll want an OK and a Cancel button.) Determine how you'd like these buttons to behave.

Identify data states that should trigger the enabling or disabling of various controls.

DESIGN SPECIFICATIONS

You might come up with a solution that looks like that in Figure 5.4 in Exercise 5.3. The form and its controls implement data input with validation constraints as follows:

- **Form**. Force all alphabetic characters to uppercase. Implement a form-level help key combination (not the F1 key). The form will also implement a `Public Boolean` variable (custom form property) that indicates whether the form has been closed with all valid data.
- **Age**. Use a `TextBox` control that limits input to numeric characters only. It should still allow the Backspace key to have effect. There should be a validation routine for age that takes into account maximum and minimum ages that the user can input.

continues

CASE STUDY: A SIMPLE DATA ENTRY FORM

continued

You should call this routine from the `Validate` event procedure of the `TextBox`. If the routine indicates that the data is not correct, then the `Validate` event procedure should set its `Cancel` parameter to `True`.

- **Name**. Use a `TextBox` control that requires a string of a certain minimum length. There should be a validation routine that takes this into account. You should call this routine from the `Validate` event procedure of the `TextBox`. If the routine indicates that the data's not correct, then the `Validate` event procedure should set its `Cancel` parameter to `True`.
- **Type of Payment**. Because there is a limited number of types of payment, you should consider using a group of option buttons to indicate type of payment.
- **Credit Card**. Because there is a limited number of Credit Card types that your company accepts, you should consider using a `ListBox` or `ComboBox` to indicate Credit Card type. Don't use `OptionButtons` because the types of Credit Cards accepted by your company is susceptible to change in the future, and it's easier to add items to a `ListBox` or `ComboBox`. This `ListBox` or `ComboBox` should only be enabled when Type of Payment is Credit Card. You should put code in the `Click` event procedure of each Type of Payment `OptionButton` to determine whether or not this control will be enabled.
- **Cancel Button**. This button should always be available to the user. Therefore you should set its `CausesValidation` property to `False`. The code in the `Click` event procedure will set the form's `Public Boolean` to `False` and unload or hide the form.
- **OK Button**. This button should only be available if all data on the form is valid. Therefore you should write a general validation routine that checks all controls on the form, enabling the OK button if all controls are valid, but otherwise disabling it. You should then call this general validation routine from all events that signal a change to data. On this form, that would include the `Change` event of all `TextBoxes` and the `Click` event of the `OptionButtons` and the `ListBox` or `ComboBox`. The code in the `Click` event procedure will set the form's `Public Boolean` to `True` and unload or hide the form.

CHAPTER SUMMARY

This chapter covered the following topics:

- Keystroke validation using the `KeyUp`, `KeyDown`, and `KeyPress` events.
- Enabling keystroke validation at the form level with the `KeyPreview` property.
- Using the `Validate` event and the `CausesValidation` property to enable validation of field contents.
- Enabling and disabling controls on a form based on user input.

KEY TERM

- Input Validation

APPLY YOUR KNOWLEDGE

Exercises

5.1 `Keystroke` Events at Field and Form Level

Estimated Time: 30 minutes

This exercise gives you some exposure to the `KeyPress` and `KeyUp`/`KeyDown` events. You also see how the form's `KeyPreview` property can be used to manage two-tier keystroke validation. Once you've finished this exercise, you should keep the results of your work for Exercise 5.2, which builds on the results of this exercise.

1. Begin a new VB project with a single form. Place controls on the form as shown in Figure 5.2. Use the Properties Window to change the control properties listed in Table 5.1.

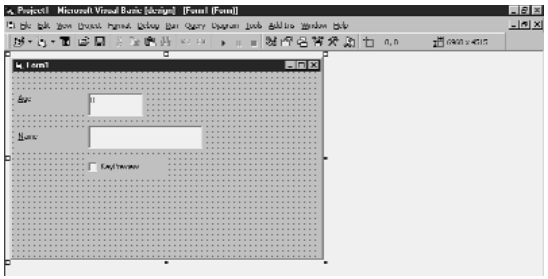


FIGURE 5.2
The form for Exercise 5.1.

TABLE 5.1

CONTROL PROPERTIES TO SET FOR EXERCISE 5.1

Control	*Property*	*New Value*
`Label`	`Name`	lblAge
	`Caption`	&Age
`Label`	`Name`	lblName
	`Caption`	&Name
`TextBox`	`Name`	txtAge
	`Text`	0
`TextBox`	`Name`	txtName
	`Text`	`<Blank>`
`CheckBox`	`Name`	chkKeyPreview
	`Caption`	KeyPreview

2. Put code to allow only numeric input for age in the `KeyPress` event of `txtKeyPress`, as follows:

```
Private Sub txtAge_KeyPress(KeyAscii As
➥Integer)
    Dim KeyChar As String
    KeyChar = Chr(KeyAscii)

    If KeyAscii > 31 Then 'ignore low-ASCII
➥chars like BACKSPACE
        'throw out non-numeric characters
        If Not IsNumeric(KeyChar) Then
            KeyAscii = 0
        End If
    End If
End Sub
```

3. Run the application and test the input validation for the Age `TextBox`. In particular, verify that only numeric keystrokes are accepted. Verify that the Backspace key works as a user would normally expect (to delete the character to the left). Try disabling the logic that allows the lower ASCII characters to pass (the outer `If…End If` pair in the code given above). Note that the Backspace key no longer works. Then re-enable the logic.

4. Also verify that the Name `TextBox` (which has no validation logic attached to its `Keystroke` events) allows both upper and lowercase character input.

5. Code the `Form_Load` event procedure and the `CheckBox` control's `Click` event procedure for management of the form's `KeyPreview` property, as follows:

APPLY YOUR KNOWLEDGE

```
Private Sub Form_Load()
    chkKeyPreview.Value = IIf(Me.KeyPreview,
➥vbChecked, vbUnchecked)
End Sub

Private Sub chkKeyPreview_Click()
    Me.KeyPreview = IIf(chkKeyPreview.Value
➥= vbChecked, True, False)
End Sub
```

6. Put the following code in the form's `KeyPress` event to force all characters to uppercase (note the final comment which shows an alternate, more efficient way to code the routine):

```
Private Sub Form_KeyPress(KeyAscii As
➥Integer)
    'Convert KeyAscii to a character
    Dim KeyChar As String
    KeyChar = Chr$(KeyAscii)

    'Convert the resulting character to
➥uppercase
    KeyChar = UCase(KeyChar)

    'Reassign the new character's Ascii code
    'back to Keyascii
    KeyAscii = Asc(KeyChar)

    'Of course, we could do it all in one line,
    'as follows (but example is less clear):
    'KeyAscii = Asc(UCase(Chr$(KeyAscii)))
End Sub
```

7. Run the application again. Type upper and lowercase characters into the `Name TextBox` with the `KeyPreview CheckBox` checked, and then with it unchecked.

8. Next provide the following `KeyPress` event procedure code for the form and `txtAge`:

```
Private Sub Form_KeyDown(KeyCode As Integer,
➥Shift As Integer)
    Dim blnIsShift As Boolean
    blnIsShift = Shift And vbShiftMask
    If blnIsShift And (KeyCode = vbKeyF9)
➥Then
        Me.Cls
        Me.Print "Form-level Shift+F9"
    End If
End Sub

Private Sub txtAge_KeyDown(KeyCode As
➥Integer, Shift As Integer)
    Dim blnIsShift As Boolean
    blnIsShift = Shift And vbShiftMask
    If blnIsShift And (KeyCode = vbKeyF9)
➥Then
        Me.Print "Control-level Shift+F9"
    ElseIf KeyCode = vbKeyF3 Then
        Me.Print "Control-level F3"
    End If
End Sub
```

9. Note the effects of pressing F9, Shift+F9, F3, and Shift+F3 with and without the form's `KeyPreview` property enabled.

Keep this project around for use in Exercise 5.2.

5.2 Field-Level Validation Techniques

This exercise builds on the previous exercise to add field content validation with code in the `Validate` event procedure and the use of separate validation routines.

Estimated Time: 35 minutes

1. Using the project created in the previous exercise, add the following code to implement general validation functions for Age and Name. Notice the use of optional parameters in the functions and form-wide constants to provide default values when those parameters are not supplied. We also add a `Public` variable that will keep track of whether or not this is closed with all data validated:

```
Option Explicit

Public DataIsOK As Boolean

Private Const DEFAULT_MAX_AGE = 150
Private Const DEFAULT_MIN_AGE = 21
Private Const DEFAULT_MIN_NAME_LENGTH = 3

Private Function IsValidAge _
  (vAge As Variant, _
   Optional MAXAGE As Variant, _
   Optional MINAGE As Variant) _
  As Boolean
```

APPLY YOUR KNOWLEDGE

```
    If IsMissing(MAXAGE) Then
        MAXAGE = DEFAULT_MAX_AGE
    End If

    If IsMissing(MINAGE) Then
        MINAGE = DEFAULT_MIN_AGE
    End If
    If IsNumeric(vAge) Then
        vAge = Val(vAge)
        IsValidAge = (vAge >= MINAGE) And
➥(vAge <= MAXAGE)
    Else
        IsValidAge = False
    End If
End Function

Function IsValidName(sName As String,
➥Optional MinLength As Variant) As Boolean
    If IsMissing(MinLength) Then
        MinLength = DEFAULT_MIN_NAME_LENGTH
    End If
    IsValidName = (Len(Trim$(sName)) >=
➥MinLength)
End Function
```

2. Enter the following code to call the validation functions from the `Validate` event procedures of `txtName` and `txtAge`:

```
Private Sub TxtAge_Validate(Cancel As
➥Boolean)
    Const MAXAGE = DEFAULT_MAX_AGE
    Const MINAGE = DEFAULT_MIN_AGE
    Cancel = Not IsValidAge((txtAge.Text),
➥MAXAGE, MINAGE)
    If Cancel Then
        Beep  'give the user some minimal
➥feedback
        MsgBox "Enter an age between " &
➥MINAGE & " and " & MAXAGE
    End If
End Sub

Private Sub txtName_Validate(Cancel As
➥Boolean)
    Const MINCHARS = DEFAULT_MIN_NAME_LENGTH
    Cancel = Not (IsValidName(txtName.Text,
➥MINCHARS))
    If Cancel Then
        Beep
        MsgBox "Enter a name that is at least "
             & MINCHARS & " characters long."
    End If
End Sub
```

3. Run the application to make sure that the validation routines cause the appropriate behavior. Enter names that are too short and ages that are outside the allowed range. Notice that you cannot set focus outside a `TextBox` if you don't enter valid data.

4. Add a `CommandButton` to the form as shown in Figure 5.3. Rename the `CommandButton` to `cmdCancel` and give it the `Caption` "&Cancel." For the moment, do not modify its `CausesValidation` property. Verify that `CausesValidation` remains at its default value of `True`. Enter the following code in the `CommandButton's Click` event procedure:

```
Private Sub cmdCancel_Click()
    DataIsOK = False
    Unload Me
End Sub
```

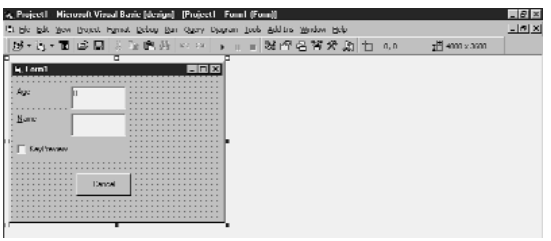


FIGURE 5.3
The form for Exercise 5.2.

5. Run the application. Notice that you can't click the `CommandButton` until the `TextBox` that currently has the focus contains valid data. Of course this is not the behavior that you want for the user interface. The user should always be able to cancel input without having to enter valid data anywhere. In the next step, you'll set the `CommandButton`'s `CausesValidation` property to fix this problem.

APPLY YOUR KNOWLEDGE

6. Stop the application and change the `CommandButton`'s `CausesValidation` property to `False`. You might also want to change the `CommandButton`'s `Cancel` property to `True` (this will allow the Esc key to execute the `CommandButton`'s `Click` event procedure code).
7. Test the application and note that the user is now able to click the Cancel button even without entering valid data in the current control. Also, verify that the Esc key causes the form to unload.

Again, save this project for use in the next exercise.

5.3 Selectively Enabling Controls

In this exercise, you'll build on the application from the previous exercises to add a group of option buttons to indicate Form of Payment. Then you'll add a drop-down list (for Credit Card Type) and a `CommandButton` (an OK button) that will be enabled only at certain times. The Credit Card Type list should only be enabled when Form of Payment is Credit Card. The OK button will be enabled only when all data on the screen is complete and valid.

You'll add the code that allows the application to determine when these controls should be enabled.

Estimated Time: 20 minutes

1. Add a `ListBox` and a second `CommandButton` as well as a `Frame` with three `OptionButtons` to the project from the previous exercises as illustrated in Figure 5.4. Table 5.2 details the property values that you should set for these new controls.

   Note that you can make multiple entries into a `ListBox` by pressing Ctrl-Enter between each entry. If you press Enter by itself, you will exit the `ListBox`.

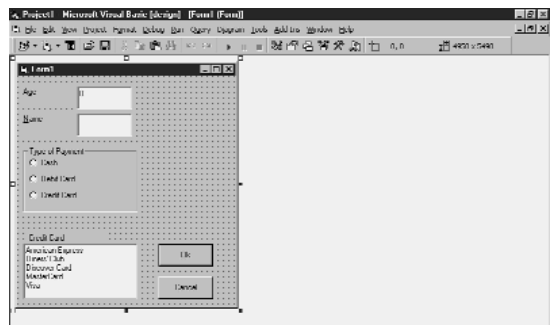


FIGURE 5.4
The form for Exercise 5.3.

TABLE 5.2

CONTROL PROPERTIES TO SET FOR EXERCISE 5.3

Control	*Property*	*New Value*
`Frame`	`Name`	fraTypeOfPayment
	`Caption`	Type of Payment
`OptionButton`	`Name`	optCash
	`Caption`	Cash
`OptionButton`	`Name`	optDebitCard
	`Caption`	Debit Card
`OptionButton`	`Name`	optCreditCard
	`Caption`	Credit Card
`Label`	`Name`	lblCreditCard
	`Caption`	Credit Card

continues

APPLY YOUR KNOWLEDGE

TABLE 5.2 *continued*

CONTROL PROPERTIES TO SET FOR EXERCISE 5.3

Control	*Property*	*New Value*
ListBox	Name	lstCreditCard
	List	American Express
		Diners' Card
		Visa
		MasterCard
		Discover
	Sorted	True
CommandButton	Name	cmdOK
	Caption	&Ok
	Default	True

2. Place code in the OK button's `Click` event procedure that will hide the form, setting its `Public` flag variable to indicate that the data on the form is valid:

```
Private Sub cmdOK_Click()
    DataIsOK = True
    Me.Hide
End Sub
```

3. Add a function named `ValidPaymentForm` to check the relative status of the newly added `OptionButtons` and `ListBox` to see if together they constitute valid payment information. (Basically, either Cash or Debit Card must be chosen as the Form of Payment or, if Credit Card is the Form of Payment, then a Credit Card Type must also be chosen from the `ListBox`.) The Function should read as follows:

```
Private Function ValidPaymentForm() As
Boolean
    If optCash.Value Or optDebitCard.Value
➥Then
        ValidPaymentForm = True
    ElseIf optCreditCard.Value And
➥lstCreditCard.ListIndex > -1 Then
        ValidPaymentForm = True
    Else
        ValidPaymentForm = False
    End If
End Function
```

4. Add a sub named `EnableOKButton`. This routine checks all the possible field validations on the screen (Name, Age, and Payment). If all information is valid, then the OK button is enabled. Otherwise it is disabled:

```
Private Sub EnableOKButton()
    Dim blnOKFlag As Boolean
    blnOKFlag = True
    If Not IsValidName((txtName.Text),
➥DEFAULT_MIN_NAME_LENGTH) Then
        blnOKFlag = False
    ElseIf Not IsValidAge((txtAge.Text),
➥DEFAULT_MAX_AGE, DEFAULT_MIN_AGE) Then
        blnOKFlag = False
    ElseIf Not ValidPaymentForm() Then
        blnOKFlag = False
    End If
    cmdOK.Enabled = blnOKFlag
End Sub
```

5. Call the `EnableOKButton Sub` in the `Click` event procedure of the `ListBox` and in the `Change` event procedures of both `TextBoxes`:

```
Private Sub lstCreditCard_Click()
    EnableOKButton
End Sub

Private Sub txtAge_Change()
        EnableOKButton
End Sub

Private Sub txtName_Change()
        EnableOKButton
End Sub
```

6. Also call the `EnableOKButton Sub` in the `Click` event procedure of each `OptionButton`. Add code to each of these event procedures that will determine whether or not the Credit Card list is enabled based on the Form of Payment chosen by the user:

APPLY YOUR KNOWLEDGE

```
Private Sub optCash_Click()
    lstCreditCard.Enabled = False
    EnableOKButton
End Sub

Private Sub optCreditCard_Click()
    lstCreditCard.Enabled = True
    EnableOKButton
End Sub

Private Sub optDebitCard_Click()
    lstCreditCard.Enabled = False
    EnableOKButton
End Sub
```

7. Run the application to verify the new behavior. The OK button should stay disabled until all data on the form is valid. If some data later becomes invalid, the OK button should once again turn gray.
8. In the `Form_Load` event, write a line to set the Credit Card `ListBox` control's `Enabled` property to `False`. Verify that the Credit Card `Listbox` is only enabled when the Credit Card option is clicked.
9. Verify that the OK button remains gray when the Credit Card option is chosen without any item chosen in the `ListBox`.

Review Questions

1. Why do you no longer need to use `GotFocus` and `LostFocus` events for field validation in VB6, as you did in previous versions of VB?
2. When might you disable a `TextBox` named `txtName` with `txtName.Locked = True` instead of using `txtName.Enabled = False`?
3. What's the relation between the `CausesValidation` property and the `Validate` event?
4. What's the default value of a form's `KeyPreview` property, and what's the significance of this property?
5. What's the difference between the `KeyPress` event on the one hand and the `KeyUp` and `KeyDown` events on the other?

Exam Questions

1. A sufficient condition to fire a control's `Validate` event is:

   A. The user must make a change to the control's contents and the control's `CausesValidation` property must be `True`.

   B. The user must make a change to a control's contents and set focus to another control on the same form whose `CausesValidation` property is `True`.

   C. The user must set focus to another control on the same form and the `CausesValidation` property of the control losing focus must be `True`.

   D. The user must set focus to a control whose `CausesValidation` property is `True`.

2. To process keystrokes at the form-wide level, you must

   A. Set the form's `KeyPreview` property to `True`.

   B. Program the form's `KeyUp`, `KeyDown`, or `KeyPress` events.

   C. Set the form's `KeyPreview` property to `True` and program at least one of the form's `KeyDown`, `KeyUp`, or `KeyPress` events.

   D. Set the form's `KeyPreview` property to `True` and program at least one of the `KeyDown`, `KeyUp`, or `KeyPress` events of a control on the form.

APPLY YOUR KNOWLEDGE

3. The timing of the `KeyPress` event is

   A. before `KeyDown` but after `KeyUp`

   B. before `KeyUp` but after `KeyDown`

   C. after both `KeyDown` and `KeyUp`

   D. before both `KeyDown` and `KeyUp`

4. The `KeyPress` event receives a parameter that is

   A. of type `Byte` and gives the ASCII value of the character corresponding to the key just pressed

   B. of type `Long` and gives the code for the physical key pressed on the keyboard

   C. of type `Integer` and gives the ASCII value of the character corresponding to the key just pressed

   D. of type `String` and gives the character corresponding to the key just pressed

5. You can programmatically cancel a user's keystroke entry by

   A. setting the parameter of the `KeyPress` event to `0`

   B. setting the parameter of the `KeyPress` event to `Chr(0)`

   C. setting the parameter of the `KeyPress` event to `Asc(0)`

   D. setting the parameter of the `KeyPress` event to `Chr$(0)`

6. A line of code in the `KeyUp` or `KeyDown` event procedure that checks to see if Ctrl was one of the shift keys being held down when F3 was pressed would read:

   A. `If (Shift = vbCtrlMask) And (KeyCode = ➥vbKeyF3) Then`

   B. `If KeyCode = vbKeyControl + vbKeyF3 Then`

   C. `If (KeyCode = vbKeyF3) And (Shift And ➥vbCtrlMask) Then`

   D. `If (Shift And vbCtrlMask) And (KeyCode ➥And vbKeyF3) Then`

7. The most appropriate strategy for implementing field-level validation in VB would be:

   A. programming the controls' `Keystroke`, `Change`, `GotFocus`, and `LostFocus` events

   B. programming the controls' `Keystroke`, `Change`, and `Validate` events

   C. setting the form's `KeyPreview` property to `True` and programming the controls' `Keystroke` events

   D. setting the form's `KeyPreview` property to `False` and programming the controls' `Keystroke` events

8. How would you make sure that the user enters a whole number in a `TextBox`?

   A. Discard inappropriate keystrokes in the `KeyUp` or `KeyDown` event procedure.

   B. Use the `IsNumeric` function in the `Validate` event procedure.

   C. Use the `IsNumeric` function in the `Change` event procedure.

   D. Discard inappropriate keystrokes in the `KeyPress` event procedure.

9. You are writing an application for distribution to many different countries that are likely to have different keyboard mappings. You have a situation where you need to detect whether the user has keyed a specific ASCII character. The best place to detect the character would be

APPLY YOUR KNOWLEDGE

A. in the `Change` event procedure

B. in the `KeyUp` or `KeyDown` event procedure

C. in the `Validate` event procedure

D. in the `KeyPress` event procedure

Answers to Review Questions

1. `GotFocus` and `LostFocus` events are no longer necessary for field validation in VB6, because VB6 has introduced the `Validate` event for controls. See "Field-Level Validation Techniques."
2. You can prevent changes to a `TextBox` by setting its `Locked` property to `True` when there might be more data in the `TextBox` than the user could see and you wanted to allow the user to set focus to the `TextBox` to scroll through the data. See "Enabling Controls Based on Input."
3. A control's `Validate` event will fire when the user attempts to set focus to another control whose `CausesValidation` property is `True`. See "The `Validate Event` and `CausesValidation` Property."
4. The form's `KeyPreview` property by default is `False`. Setting it to `True` enables the form's `KeyUp`, `KeyDown`, and `KeyPress` events. See "Enabling Two-Tier Validation With the Form's `KeyPreview` Property."
5. The `KeyPress` event detects a character while the `KeyUp` and `KeyDown` events detect physical keystrokes. See "`KeyPress` Versus `KeyUp` and `KeyDown`."

Answers to Exam Questions

1. **D**. A control's `Validate` event fires when the user sets focus to another control on the same form whose `CausesValidation` property is `True`. B would also cause the `Validate` event to fire, but the user does not have to make a change in order for the event to fire. A and C are incorrect, because the current control's `CausesValidation` property has nothing to do with whether its `Validate` event fires (firing of the `Validate` event depends on the control that is receiving focus, not the control that loses focus). See "The `Validate` Event and `CausesValidation` Property."
2. **C**. To process keystrokes at the form-wide level, you must set the form's `KeyPreview` property to `True` and program at least one of the `KeyDown`, `KeyUp`, or `KeyPress` events. While A and B are both necessary, neither one is sufficient to implement form-level keystroke handling. D is wrong because the `Keystroke` events of controls can never process keystrokes at the form level. See "`Keystroke` Events at Field and Form Level."
3. **B**. The timing of the `KeyPress` event is before `KeyUp` but after `KeyDown`. See "The `KeyPress` Event."
4. **C**. The `KeyPress` event receives a parameter that is of type `Integer` and gives the ASCII value of the character corresponding to the key just pressed. See "The `KeyPress` Event."
5. **A**. You can programmatically cancel a user's keystroke by setting the `KeyPress` event's parameter (known as `KeyAscii`) to `0`. B. and D. are incorrect, because the `Chr` function returns a `String` or variant `String`, and `KeyAscii` must be set to an integer value. C. is incorrect, because the `Asc` function requires a `String` argument. See "The `KeyPress` Event."

APPLY YOUR KNOWLEDGE

6. **C**. A line of code in the `KeyUp` or `KeyDown` event procedure that checks to see if Ctrl was one of the shift keys being pressed when F3 was pressed would read:

```
If (KeyCode = vbKeyF3) And (Shift And
➥vbCtrlMask)  Then
```

To find out whether a particular key press caused the `KeyUp` or `KeyDown` event to fire, you must check the value of the `KeyCode` parameter, comparing it with the appropriate VB internal constant (in this case, `vbKeyF3`). To check the state of the shift keys (Ctrl, Alt, and Shift) in the `KeyUp` or `KeyDown` events, you must use the `Shift` parameter. The `Shift` parameter is a bit mask containing information about the state of all three shift keys. To extract information about a single shift key, you can `AND` the corresponding VB internal constant for the key with the `Shift` parameter, as the correct answer does with (`Shift AND vbCtrlMask`. Answer A is incorrect because you cannot compare the `Shift` parameter with an internal constant to get meaningful information. B is incorrect because the `KeyCode` parameter does not store information about the state of the shift keys. D is incorrect because it extracts information incorrectly from the `KeyCode` parameter. See "The `KeyUp` and `KeyDown` Events."

7. **B**. The most appropriate strategy for implementing field-level validation in VB would be programming the controls' `Keystroke`, `Change`, and `Validate` events. Answer A reflects techniques that have become obsolete in VB6 with the introduction of the new `Validate` event and `CausesValidation` property. Answers C and D are incorrect because the form's `KeyPreview` property has nothing to do with field-level validation. See "Field-Level Validation Techniques."

8. **D**. The best way to validate that a user enters a whole number in a `TextBox` is to discard inappropriate keystrokes in the `KeyPress` event procedure. The user's erroneous keystrokes are simply ignored, thereby providing the most transparent validation. A is less appropriate because it's harder to check for a range of keystrokes in the `KeyUp` and `KeyDown` event procedures. B (using the `Validate` event) would work, but it would allow the user to enter many erroneous keystrokes before validation. Note that the question asks about a WHOLE number. The `Validate` event might work better for numbers that could take a decimal point. C is not appropriate at all, because you have no way of rejecting individual keystrokes in the `Change` event. See "The `KeyPress` Event," "The `KeyUp` and `KeyDown` Events," and "Field-Level Validation Techniques."

9. **D**. The best way to detect whether the user has keyed a particular ASCII character is in the `KeyPress` event. The `KeyPress` event's single parameter, `KeyAscii`, is an integer value giving the ASCII code of the character generated by the pending keystroke. B is not the best choice because the `KeyUp` and `KeyDown` events' `KeyCode` parameter refers to the physical keystroke, not to the character generated by the keystroke. The `Change` and `Validate` events are inappropriate for the reasons discussed in the answer to question 8. See "The `KeyPress` Event," "The `KeyUp` and `KeyDown` Events," and "Field-Level Validation Techniques."

OBJECTIVE

This chapter helps you prepare for the exam by covering the following objective and its subobjective:

Write code that processes data entered on a form (70-175 and 70-176).

- **Given a scenario, add code to the appropriate form event. Events include `Initialize`, `Terminate`, `Load`, `Unload`, `QueryUnload`, `Activate`, and `DeActivate`.**

▶ The seven form events named in the exam subobjective for this chapter are all very familiar to VB programmers. Programmers can monitor, react to, and control the various phases of a form's life by placing code in these events' corresponding event procedures.

CHAPTER 6

Writing Code that Processes Data Entered on a Form

OUTLINE

Relative Timing of Form Events **239**

`Initialize`, `Load`, and `Activate` Events **240**

The `Initialize` Event 240

The `Load` Event and the `Activate` Event 241

`Deactivate`, `Unload`, `QueryUnload`, and `Terminate` Events **243**

The `DeActivate` Event 243

The `QueryUnload` Event 243

The `Unload` Event 245

The `Terminate` Event 246

`Activate`/`DeActivate` Versus `GotFocus`/`LostFocus` Events 246

`Show`/`Hide` Methods Versus `Load`/`Unload` Statements 247

Using the `Unload` and `QueryUnload` Events in an MDI Application **248**

Form Methods and Their Effect on Form Events 249

Implicitly Loading a Form 249

`Show` and `Hide` 249

Manipulating a Form from Another Form's `Load` Event Procedure 250

Chapter Summary **254**

STUDY STRATEGIES

- Read the VB help screen for each of the events listed in the objective.
- Write an application to experiment with the timing and usage of each of the events. At a minimum, you should put a Debug.Print statement in each event procedure. Exercise 6.1 gives a good basis for such a program.
- Make sure to also experiment with the parameters of the `Unload` and `QueryUnload` events (see Exercise 6.2 for ideas).
- Memorize (by reading the section "Relative Timing of Form Events" and by experimenting on your own) the order and timing of these events.

INTRODUCTION

This chapter deals with the following:

- The important events of forms
- When form events fire
- How to effectively program with form events

A VB form supports many events to help you efficiently manage the phases of a form's lifetime within the application.

It's useful to know about programming form events because they often mark important moments in an application's runtime session, such as:

- When a form first becomes available in memory
- When a user changes attention between forms
- When a form leaves memory

These events can become quite important for tying together parts of your application, for startup and cleanup operations, and for providing more global validation of user changes to controls on a form.

NOTE

Should the Word "Form" Be Capitalized? In previous versions of VB, Microsoft's documentation always capitalized the word "form." After all, forms are objects, just like, say, `TextBoxes` or `Recordsets`. However, in much (not all) of the newer documentation, as well as in the published Exam Objectives, Microsoft has begun to drop the capitalization of this word.

We may slip up and type "Form" here and there in this book, but our conscious policy is to use the form "form."

RELATIVE TIMING OF FORM EVENTS

You might wish to refer to this section as you read the following sections on individual form events.

The following events will occur in the order listed when a form first loads into memory:

1. `Initialize`
2. `Load`

The following events will occur in the order listed when a form becomes the active form in the application (depending on the circumstances, this might happen right after the form loads into memory):

1. `Activate`
2. `GotFocus` (triggered only if there are no enabled or visible controls on the form)

The following events happen to an active form when another form in the same application becomes active:

1. `LostFocus` (triggered only if there are no enabled or visible controls on the form)
2. `DeActivate`

The following events will occur in the order listed when a form is unloaded from memory:

1. `QueryUnload`
2. `Unload`
3. `Terminate` (fires only if you set the form to `Nothing` in or after the `Unload` event)

Initialize, Load, and Activate Events

These events fire toward the beginning of the life of a form. However, the exact circumstances and timing of these events can differ.

Generally, of the three "beginning" events, the `Initialize` event fires least often; the `Activate` event fires most often, while the `Load` event fires less often than the `Activate` event and more often than the `Initialize` event.

The `Activate` event fires most often (at least on a multi-form application) because the user can navigate back and forth between forms many times after the forms are loaded in memory and visible.

If the form is loaded, unloaded, and re-loaded into memory several times in the application, then the form's `Load` event will fire at each load. The `Initialize` event, however, might not necessarily fire each time the form re-loads, because the form instance might not have been set to `Nothing` in the previous unload.

NOTE

When Is a Form Instance Destroyed?
For more information on how a form instance is completely destroyed, see the section in this chapter on the `Terminate` event.

The `Initialize` Event

The *`Initialize`* event fires when an instance of the form is created in your application.

`Initialize` fires just before `Load` the very first time the form is loaded into memory during the application's current session. It may or may not fire before subsequent `Loads` of the form, depending on what else has happened:

- If the form was previously loaded and then unloaded but was not un-instantiated by having its instance set to `Nothing`, the `Initialize` event *will not* fire the next time the form is loaded.

- If the form was previously unloaded and then un-instantiated with

  ```
  Set formname = Nothing '(this also fires Terminate event)
  ```

  the `Initialize` event *will* fire the next time the form loads.

Forms have an `Initialize` event because Microsoft wants us to see forms as programmable, customizable object classes, and custom programmer-created VB classes always sport built-in `Initialize` and `Terminate` events.

You should therefore program a form's `Initialize` event procedure similar to the way you'd program a custom VB class' `Initialize` event procedure. That is, you should initialize custom form properties here. This might include setting the value of `Public` form variables (`Public` variables in a form are automatically properties of the form) or of `Private` variables (`Private` variables can store intermediate values of properties that are implemented with `Property` procedures).

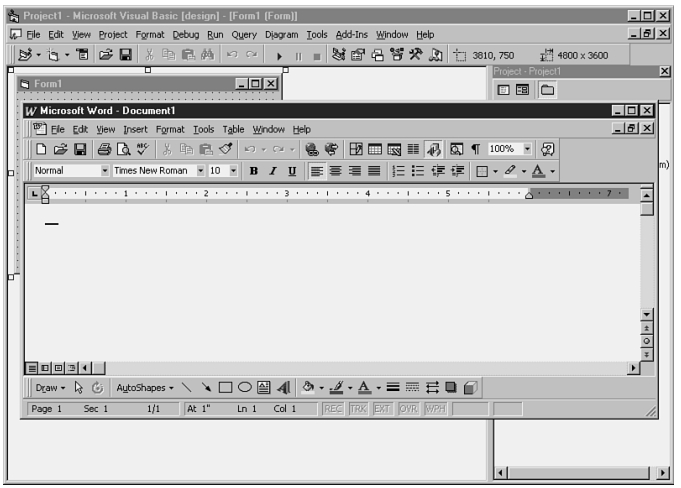
The `Load` Event and the `Activate` Event

The *`Load`* event fires when the form loads into memory. This event's procedure is the customary place for programmers to insert code that sets form-level `Private` variables (not associated with custom properties) and performs other startup processes.

A form receives an *`Activate`* event when it becomes the active form.

A form is the active form in an application when the focus in the application is on the form itself or (more likely) on a control within the form. This could mean that either the form itself has the focus or that a control on the form has focus. The user can tell which form is active because the title bar appears highlighted and the form usually appears on top of the user's desktop, as in Figure 6.1.

FIGURE 6.1
An Active form on the user's desktop.



NOTE

Forms as an Application's Startup Object As mentioned in Chapter 4, "Creating Data Input Forms and Dialog Boxes," a form might be the application's startup object. In such a case, you can count on the startup form's `Initialize`, `Load`, and `Activate` events to run when the application starts.

Focus can come to the form (and fire the `Activate` event) either by user action or through program code. For instance, the user can activate the form from another form in the application by clicking on the form with the mouse. Your program can cause a form to become the application's active form by calling that form's `Show` method or by calling the `SetFocus` method of one of the form's controls.

The `Activate` event could therefore fire many more times in an application than either the `Load` or `Initialize` events (because it could fire as often as the user returns to the form with the mouse or as often as your application makes it the active form).

Sometimes you will need to decide between putting code in the form's `Activate` event procedure and putting it in the `Load` event procedure. The following points can serve as guidelines for deciding between `Activate` and `Load` event procedures:

- The `Load` event procedure fires before the form is established visually and before the data connections of any data controls that it contains have been established. You will get runtime errors if you place code in `form_Load` that tries to use data connections belonging to the form's own Data Controls. Code that attempts graphics output to the form in `form_Load` will have no effect unless you set the form's `AutoRedraw` property to `True`.

- The `Activate` event can fire multiple times once the form is loaded, so you should be cautious when placing code into the `Activate` event procedure if you do not want that code to run more than once per form session. If such code must go in the `Activate` event procedure (say, because it relies on an established data connection), then you could put a `Boolean Static` variable in the `Activate` event procedure to keep track of whether the `Activate` event has already run.

DeActivate, Unload, QueryUnload, and Terminate Events

These events fire toward the end of a form's life. Their event procedures are therefore good places to put "cleanup" code. Once again, the exact circumstances and timing of these events can differ.

NOTE

Be Careful With the End Statement. You can quickly terminate your application by putting a one-word statement, "End", in your code. However you should be aware that doing so will abruptly stop any further processing by the application.

In particular this means that loaded forms' `QueryUnload`, `Unload`, and `Terminate` events will not run. If you've put code in these event procedures to perform cleanup on the environment or save pending data, this code will not run when you call End.

Pressing the VCR stop button on the VB menu in order to stop your design-time application is equivalent to calling the End statement.

The `DeActivate` Event

The `DeActivate` event fires when a form or a control on the form loses focus to another object in the application outside the form. The `DeActivate` event does *not* fire when the user navigates to a completely different application, but only when the user navigates elsewhere in the current application. The `DeActivate` event does not fire when the application closes or when the current form unloads.

The `QueryUnload` Event

The `QueryUnload` event fires just before a form unloads from memory. The `QueryUnload` event fires just before the `Unload` event. Its main purpose is to let you detect why the form is being unloaded and to programmatically halt unloading if necessary.

The `QueryUnload` event procedure takes two parameters:

- **`Cancel`** This is a `True`/`False` value which is `False` by default. When `Cancel` is `False`, it means that the unloading will continue. You can set it to `True` to stop the form from unloading.

- **`UnloadMode`** This parameter can take several values, corresponding to how the `QueryUnload` event was triggered. You can compare `UnloadMode`'s value with one of the following VB internal constants:

  - vbformControlMenu. The form's `QueryUnload` event was triggered because the user is closing the form.
  - vbformCode. The form's `QueryUnload` event was triggered by code that programmatically closes the form.
  - vbAppWindows. The `QueryUnload` event was triggered because the Windows session is ending.
  - vbAppTaskManager. The `QueryUnload` event was triggered because the Windows Task Manager is closing your application.
  - vbformMDIform. The form is an MDI Child, and the MDI Parent is closing.

A common use of the `QueryUnload` event is to prompt the user to save changes (see Figure 6.2).

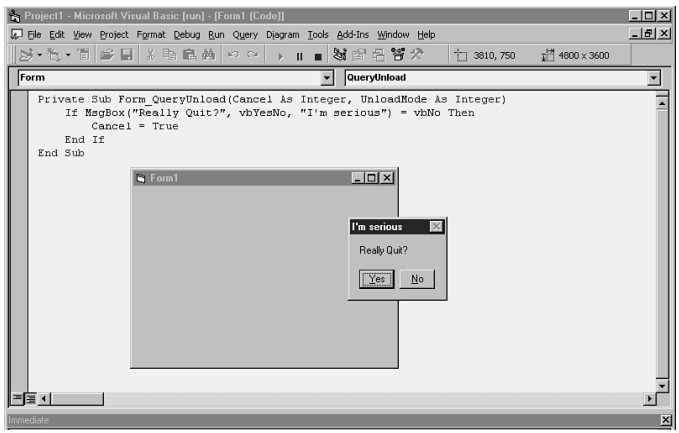


FIGURE 6.2
A typical use of the `QueryUnload` event is to give users a chance to change their minds.

You give the user the option to cancel the unload. If the user chooses to cancel, you can set the `Cancel` parameter to `True`, as in Listing 6.1.

LISTING 6.1

SETTING THE CANCEL PARAMETER IN THE QUERYUNLOAD EVENT PROCEDURE

```
Private Sub frmData_QueryUnload(Cancel as Integer, _
 UnloadMode As Integer)
    Dim intUserChoice As Integer
    intUserChoice = MsgBox("Save Changes?" , _
     vbYesNoCancel)
    If intUserChoice = vbYes Then
        Call SaveData
    ElseIf intUserChoice = vbCancel Then
        Cancel = True
    End If
End Sub
```

The `Unload` Event

The `Unload` event procedure is where programmers usually put cleanup code. The `Unload` event fires after the `QueryUnload` event. The `Unload` event procedure takes a single parameter, the `Cancel` parameter. `Unload`'s `Cancel` parameter works the same as `QueryUnload`'s `Cancel` parameter.

It is possible to stop the form from unloading in the `Unload` event procedure by setting `Cancel` to `True`. However, since the `Unload` event doesn't receive the `QueryUnload` event's `UnloadMode` parameter, your `Unload` event procedure has less information about why the form is being unloaded than the `QueryUnload` event procedure has.

For forms which are not MDI Child forms, `Unload` always happens immediately after `QueryUnload` (unless, of course, the unloading was just cancelled in the `QueryUnload` by setting the `Cancel` parameter to `True`).

NOTE

Timing of `QueryUnload` and `Unload` in MDI Applications. In an MDI application, MDI Child forms have a slightly different timing for `QueryUnload` and `Unload` events. See the section in this chapter titled "Using the `Unload` and `QueryUnload` Events in an MDI Application" for more information.

The `Terminate` Event

The `Terminate` event happens when the instance of the form has been completely unloaded and all the form's variables have been set to `Nothing`. The `Terminate` event happens only when you explicitly set the form to `Nothing` in your code during or after the `Unload` event procedure. For example, you could use the statement:

```
Set Form1 = Nothing
```

after calling the `Unload` statement for `Form1`.

You might use the `Terminate` event to perform final cleanup operations for your form.

NOTE

Choosing Between the `Activate` and `Load` Events. When a form first loads into memory and immediately displays (either because it's the application's startup form or because, say, you've used the `Show` method to load the form), the `Activate` event happens after the `Load` event. The `Activate` event procedure is a better place than the `Load` event procedure to put startup code that affects the form's appearance (such as calls to the graphics methods) or code that manipulates Data-Bound controls. This is because the `Load` event happens a bit too early for some of these operations to have their proper effect. In fact, putting some types of code in the `Load` event procedure can cause a runtime error if the code tries to manipulate runtime properties of controls that aren't fully initialized.

`Activate` / `DeActivate` Versus `GotFocus` / `LostFocus` Events

As noted earlier, a form is the active form in an application when the focus in the application is within that form. This could mean that either the form itself has the focus or that a control on the form has focus. The user can tell which form is active because the title bar appears highlighted, and the form usually appears on top of the user's desktop.

A form receives an `Activate` event when it becomes the active form and a `Deactivate` event when it loses its active status to another form in the same application. The `Activate` and `Deactivate` event procedures are ideal places to put code that needs to react to the user's navigation between forms.

Forms also support `GotFocus` and `LostFocus` events, as do controls. You might think that `GotFocus` and `LostFocus` event procedures would be good places to put code that reacts to a user entering and leaving a form in a multi-form application.

However, a form's `GotFocus` and `LostFocus` events do not usually fire: A form can only receive focus if it contains no visible enabled controls. Because most forms have at least one visible enabled control, `GotFocus` and `LostFocus` don't normally fire on forms when the user moves between forms in an application.

NOTE

`Navigation` Events Happen Only With Respect to the Current Qpplication. `Activate`, `DeActivate`, `GotFocus`, and `LostFocus` occur only with respect to movement between forms within the current application. When the user changes to or returns from another application, none of these events occur.

Show / Hide Methods Versus Load / Unload Statements

The `Load` and `Unload` statements cause a form to load into memory or to unload from memory respectively. Both statements take the name of a form or a form variable as their parameter. For example, the statement

```
Load frmMain
```

would cause frmMain to load into memory, and the statement

```
Unload frmMain
```

would take it out of memory.

Although the `Load` statement brings a form into memory, it doesn't make the form visible. You must call the form's `Show` method or set its `Visible` property to `True` in order to make it visible to the user.

You might ask, since calling the `Show` method or setting the `Visible` property will load the form anyway, why bother ever using the `Load` statement?

You might also ask, since calling the `Unload` statement would make the form invisible anyway, why bother to ever use the `Hide` method?

VB provides programmers with both `Load` and `Unload` statements and `Show` and `Hide` methods because there are two different strategies for form management in an application. Which strategy you choose depends on how your application needs to balance speed of operation with efficient use of memory. The two strategies are:

- **Fast.** Load all the forms you'll need in your application when your application begins to run. While the application is running, use only the `Show` and `Hide` methods of forms. Use only the `Unload` statement when your application is ending.

- **Memory Efficient.** Load a form (either with the `Show` method or a combination of the `Load` statement and `Show` method) only when you need to use it. Immediately unload a form as soon as you don't need it in the application.

NOTE

Another Use of the `Hide` Method
Even in the Memory Efficient model described here, there is a good reason to use `Hide` when the form needs to disappear instead of calling `Unload` immediately:

Consider the situation where you've loaded a form modally from elsewhere in your code. When the user dismisses the form, you might want to check the state of some of the form's controls by following these steps:

From inside the modal form, you should close it with the `Hide` method.

In your calling code, check the information you need from the form.

Only *then* should you unload the form.

When you look at the Fast strategy, it might appear that there would be a big delay at the beginning of the program as your application loaded all the forms it was going to use. While this is objectively true, programmers usually cover up for the fact by supplying a "splash" screen to show the user flashy graphics as the application loads the forms. This is such a common technique in Windows programming that users have come to accept and even expect a delay of several seconds when a program begins to run. Once the forms are loaded, there will be no further delays as the application runs (because no forms will need to be loaded).

In reality, you'll find yourself using a combination of both the Fast and Memory Efficient strategies. Not every VB application, for example, will be able to load and maintain all of its forms in memory throughout the entire session of the application.

Using the Unload and QueryUnload Events in an MDI Application

As we note in "`Initialize`, `Load`, and `Activate` Events", a form's `QueryUnload` event happens before its `Unload` event. Both `QueryUnload` and `Unload` event procedures receive a `Cancel` parameter that you can programmatically set to `True` to stop the form from unloading.

When you attempt to unload the main MDI form in an MDI application, VB unloads all the open Child forms first. This means that the various `Unload` and `QueryUnload` events have a special timing relationship in an MDI application. When there is an attempt to unload the main MDI form, the order of the `Unload` and `QueryUnload` events is as follows:

1. The MDI form's `QueryUnload` event
2. The `QueryUnload` event of each open Child form
3. The `Unload` event of each Child form
4. The `Unload` event of the MDI form

If `Cancel` is set to `True` within any of these event procedures, the entire unloading process stops. If unloading is halted during any of the `QueryUnload` event procedures, then none of the `Unload` events is triggered, and no form is unloaded. If unloading is halted during any of the `Unload` event procedures, then only forms whose `Unload` events happened before the one where the `Cancel` occurred will unload.

Form Methods and Their Effect on Form Events

The following sections discuss how referring to a form's members in code can load a form into memory, and how the form's `Show` and `Hide` methods relate to the loading and unloading of a form.

Implicitly Loading a Form

A form which is not yet in memory will load into memory automatically whenever you refer to any of its *built-in* methods or properties in code, or to any of the methods or properties of a control on the form. Thus, statements such as

```
Form1.Caption = "Hello"
Form1.txtCity.Text = "Chicago"
Form1.Show
```

would cause `Form1` to load if it had not already been loaded.

Causing a form to load in this way is known as *implicit loading*.

WARNING

Referring to Custom Members Won't Cause Implicit Loading
If you implement custom methods on a form through custom procedures or custom properties through `Public` variables or Property `Let`/`Get`/`Set` procedures, references to these items will *not* cause implicit loading of the form, and the form's `Load` event will not fire.

The `Initialize` event, however, will still fire in these cases.

`Show` and `Hide`

These methods toggle a form's visibility. Calling a form's `Show` method sets the form's `Visible` property to `True`, and the form then becomes the application's active form. After a call to a form's `Hide` method, the form's `Visible` property will be `False`, and the form will no longer be the application's active form.

The `Show` method does more than simply toggle a form's `Visible` property: It also can accept a parameter that indicates the form's modal state.

- A modeless form is the default way of managing a form in an application. When a form displays modelessly, the rest of the VB application continues any execution it needs to finish, and the user is able to freely navigate between this modeless form and any other forms in the application. Since modeless is the default state, modeless is implied when there is no reference to the parameter.

  ```
  Form1.Show
  MsgBox "Form1 visible" 'this line runs immediately
  ```

  If you wish to provide more clarity in your code, you can use the vbModeless parameter:

  ```
  Form1.Show vbModeless
  ```

- A modal form demands more attention from the user. If the form is displayed modally, the user cannot navigate to other forms in the application. The form remains active until the form is closed. In addition, the calling procedure that originally shows the modal form with the call to its `Show` method will not resume until the form has been unloaded or hidden by either user action or code. You can display a form modally by calling the `Show` method with the vbModal constant as a parameter:

  ```
  Form1.Show vbModal
  'Next line won't run
  'till after Form1 is dismissed
  MsgBox "Finished Form1"
  ```

See "`Show`/`Hide` Methods Versus `Load`/`Unload` Statements."

Manipulating a Form from Another Form's Load Event Procedure

It might seem that using a current form's `Load` event procedure to call another form's methods or loading another form implicitly or explicitly would be asking for trouble.

However, it's perfectly possible to do so, and in fact you can accomplish some tasks in a simple and elegant fashion by loading and activating a second form from a first form's `Load` event procedure.

For instance, consider a form that is the main data input form for an application. Perhaps this form needs a preliminary dialog box, such as a login screen, just before it appears. One very simple way to achieve this effect is by performing the following steps:

1. In the `Load` event of the main form, put a modal call to the `Show` method of the login dialog box form with a statement such as

   ```
   Secondform.Show vbModal
   ```

2. Let the login dialog box expose a custom flag property through a `Public` variable or through Property procedures. This property will be a Boolean value that indicates whether or not the login has been successful.
3. Just before the login hides itself, it sets the flag property appropriately.
4. The `Load` event procedure of the main form has not finished running in all the time that the login dialog box was running because the login dialog box was running modally, and therefore the main form's `Load` event procedure was paused.
5. The code in the main form's `Load` event that follows the modal call to the login dialog box can check the dialog box form's `Boolean` flag property to decide what action to take. After retrieving this information, the calling code in the main form should manage memory efficiently by setting the login dialog box form to `Nothing`.
6. If the main form decides not to unload itself, then it will continue loading and will become the application's active form.

Perhaps the key point to bear in mind is step 4 above: Actions you take to manipulate other forms from within a given form's `Load` event procedure have their effect regardless of where you call them from. If you call the secondary form modally (as described in step 1 above), then the current `form_Load` pauses until the secondary form is dismissed.

Conversely, if you were to call the `Show` method of a main form and you were to call up a second form in a modeless state from within the first form's `Load` event, using code such as

```
Secondform.Show vbModeless
```

or simply

```
Secondform.Show
```

then the second form would load implicitly (if it were not already loaded) and would display.

However, the main form's `Load` event procedure would continue to run (because the second form was modeless), and so the first form would eventually load and display as well. The net result would be that you would end up with both forms visible, and the first form would end up as the active form in the application.

CASE STUDY: A SMALL DATA INPUT SYSTEM WITH LOGIN SECURITY

NEEDS

Your company needs a small data-entry system that gets several pieces of discrete information from a user. Access to this system needs some simple password protection. Access to the data would not be considered a serious security breach, but management would like to keep curious users from inadvertently changing something they don't understand.

REQUIREMENTS

Your applications designer has specified a small data maintenance system with the following features:

- The user can only enter this system by first negotiating a login screen.
- The login screen will take any reasonable-length input for UserID.
- The login screen will take an arbitrary password such as "PASSWORD."
- If users try to continue without giving an acceptable login and password, they'll be prompted to either try again or to abandon the system.
- Users can cancel the login attempt and exit the system at any time.
- The data entry system itself consists of two data input forms (a main form and a secondary form).
- After a successful login, the main input form appears by itself.
- The secondary form can be called up from the main form.
- Users can freely move between the two forms.
- Whenever users leave the secondary form, they will receive a save prompt if there have been any changes made to the form.
- Users can specify a "remember" option on the secondary form. When this option is flagged, users' last input will be stored before the secondary form closes and re-displayed when the secondary form is re-displayed.

CASE STUDY: A SMALL DATA INPUT SYSTEM WITH LOGIN SECURITY

DESIGN SPECIFICATIONS

The points given in the scenario suggest the following implementation strategies:

- Login screen could be implemented as a modal screen displayed from the `Load` event procedure of the main input form (spec #1 & 7).
- A separate routine could run validation checks for the login (spec #2 & 3).
- Login screen could have a custom end-status property indicating how it terminated. Main input form's `Load` event could read this flag after the login terminates to determine whether or not to unload itself or continue loading (spec #5 & 7).
- The login screen's `QueryUnload` event procedure would determine the state of its end-status property. Among other things, the `QueryUnload` event procedure would determine how the form is being unloaded and, if appropriate, call login validation routines. Code here could prompt users for a retry when appropriate (spec #4).
- Main data input form would have a `CommandButton` to modelessly call the `Show` method of the secondary form (spec #8 and 9).
- The secondary data input screen could have a `Private` "dirty flag" variable to determine whether a save was necessary (spec #10).
- The secondary data input screen could have code in its `Deactivate` event procedure to check the "dirty flag" and determine whether to prompt the user for a save (spec #9 & 10) before relinquishing focus from the form's controls.
- User input on the secondary form can be mirrored in one or more custom form properties. The secondary "remember" option can be implemented with a `CheckBox` control. If it's not checked, code in the `Unload` event procedure will set the secondary form to `Nothing`, thus destroying the contents of the "mirror" properties. If the remember option is checked, the form will not be set to `Nothing`. In the `form_Load` event, any values in the "mirror" properties will be loaded into the appropriate input controls (spec #11).

The design details given in this section are implemented in Exercise 6.2.

CHAPTER SUMMARY

KEY TERMS

- Active form
- Implicit loading
- Modal
- Modeless
- Startup object

This chapter discussed the important form events: `Initialize`, `Terminate`, `Load`, `QueryUnload`, `Unload`, `Activate`, and `DeActivate`, including the following specific points:

- Relative timing of form events
- Use of the `Initialize`, `Load`, and `Activate` events
- Use of the `DeActivate`, `Unload`, `QueryUnload`, and `Terminate` events
- Comparison of the `Activate`/`DeActivate` and `GotFocus`/`LostFocus` events
- `Show`/`Hide` methods; `Load`/`Unload` statements
- Using the `Unload` and `QueryUnload` events in an MDI application
- Implicitly loading a form
- Manipulating a form from another form's `Load` event procedure

APPLY YOUR KNOWLEDGE

Exercises

6.1 Testing the Major Form Events

The purpose of this exercise is to experiment with the major form events so that you can see for yourself the relative timing of the events and the situations that cause each event to fire.

You will use three forms to test the major form events. One form (`Form1`) will control the form to be tested (`Form2`), and a third form (`frmDisplay`) will display diagnostic messages so that you can examine the timing of various events.

Estimated Time: 45 minutes

1. Start a new project in Visual Basic and create a form like the one shown in Figure 6.3. The form will keep the default name of `Form1` and will contain `CommandButton` controls named cmdLoadform2, cmdformShow, cmdUnloadform2, cmdform2Hide, cmdform2Nothing, cmdClearDisplay, and cmdEndAbruptly, all corresponding to the similarly labeled `CommandButtons` shown in the figure. Set the `Enabled` property of cmdform2Nothing to `False`. Set the form's `AutoRedraw` property to `True`. See Table 6.1 for a list of all the property settings for the `CommandButtons` and the form.

TABLE 6.1

PROPERTIES TO ASSIGN TO OBJECTS ON FORM1

Object	*Property*	*Value*
Form1	AutoRedraw	True
CommandButton	Name	cmdLoadForm2
	Caption	Load Form2
CommandButton	Name	cmdForm2Show
	Caption	Form2.Show
CommandButton	Name	cmdUnloadForm2
	Caption	Unload Form2
CommandButton	Name	cmdForm2Hide
	Caption	Form2.Hide
CommandButton	Name	cmdForm2Nothing
	Caption	Set Form2=Nothing
	Enabled	False
CommandButton	Name	cmdClearDisplay
	Caption	Clear Display
CommandButton	Name	cmdEndAbruptly
	Caption	End Abruptly

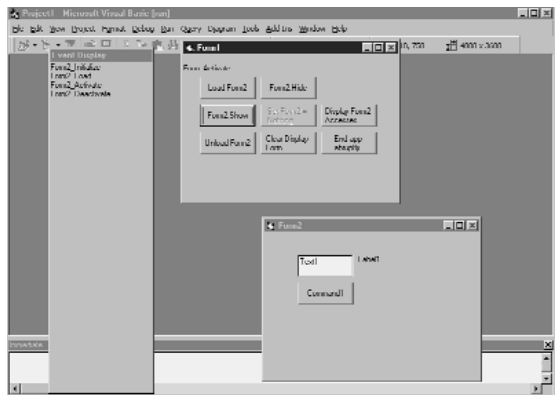


FIGURE 6.3
The forms for Exercise 6.1.

APPLY YOUR KNOWLEDGE

2. Add a second form and name it `frmDisplay`. Give it the appropriate caption and shape as shown in Figure 6.3. Set its `AutoRedraw` property to `True` (in this way, its output will still be visible even if another form covers it while it produces graphics output), its `BorderStyle` property to 1 - Fixed Single, and its `ControlBox` property to `False`.

3. Add a third form and let its Name default to `form2`. Put a single line of code into the following event procedures of the form: `Activate`, `DeActivate`, `GotFocus`, `Initialize`, `Load`, `LostFocus`, `QueryUnload`, `Terminate`, and `Unload`. The line of code in each event procedure should read `frmDisplay.Print` "***`EventProcName`***"

   when EventProcName is the name of the event procedure where you are placing the code (e.g., "`form_Unload`," "`form_Activate`," and so on.)

4. In `form2`'s `Unload` event procedure, add another line of code to enable Form1.cmdform2Nothing:

```
Private Sub Form_Unload(Cancel As Integer)
    frmDisplay.Print "Form2_Unload"
    Form1.cmdForm2Nothing.Enabled = True
End Sub
```

5. In `form2`'s `Terminate` event procedure, add a line of code to disable this same `CommandButton`:

```
Private Sub Form_Terminate()
    frmDisplay.Print "Form2_Terminate"
    Form1.cmdForm2Nothing.Enabled = False
End Sub
```

6. In the `Click` event procedures of the `CommandButtons` of `Form1`, add a line of code to each event procedure as shown in Table 6.2.

TABLE 6.2

LINES OF CODE TO WRITE IN FORM1'S COMMANDBUTTONS' CLICK EVENT PROCEDURES

`CommandButton` ***procedure***	***Line of code in*** `Click` ***event procedure***
cmdClearDisplayform	frmDisplay.Cls
cmdEndAbruptly	End
cmdform2Hide	form2.Hide
cmdform2Nothing	Set form2 = Nothing
cmdform2Show	form2.Show
cmdLoadform2	Load form2
cmdUnloadform2	Unload form2

7. Save and run the project. Experiment with various command sequences and note the timing of the various form events as shown on the display form. If the display form gets too crowded with information, click the `CommandButton` on `Form1` to clear the display.

8. Notice the output from `Form1`'s `Load` event procedure never appears on its surface while the output from its `Activate` event procedure does appear.

9. Notice that `form2`'s `GotFocus` and `LostFocus` events never fire. Disable all the controls on `form2` and then observe the effect of this on the `GotFocus` and `LostFocus` events.

6.2 A Multi-Form Data Input System With Login Security

This exercise implements the solution specified in the Case Study section of this chapter. You will create three forms: a main input form, a login dialog box, and a secondary input form.

APPLY YOUR KNOWLEDGE

Estimated Time: 60 minutes

1. Start a new project in Visual Basic and create a form like the one shown in Figure 6.4. Name it frmMain and make sure it's the startup object for this project. Add the controls with captions as shown in the figure and name them lblData, txtData, cmdSubDataForm, cmdSave, and cmdExit.

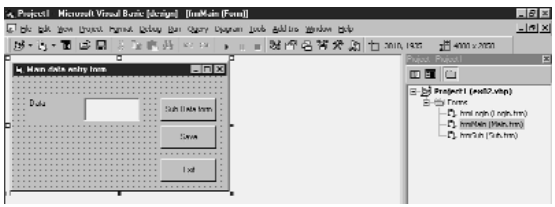


FIGURE 6.4
The main form for Exercise 6.2.

2. In frmMain's General Declarations, add a declaration for a `Private Boolean` variable named blnDirtyFlag. This variable will track whether or not there are unsaved changes pending on this form. Create a `Private Sub` procedure on the form named SaveChanges. In this exercise, we will not actually implement saving changes, so the only code you will write here will be a line to set blnDirtyFlag to `False` (indicating changes have been saved). In the `Change` event procedure for txtData, set blnDirtyFlag to `True` (to indicate that there's now an unsaved change).
3. In frmMain's `Unload` event procedure, check to see whether blnDirtyFlag has been set to `True`. If it is `True`, then use `MsgBox` to ask users if they want to save changes. If users reply "Yes", then invoke the `SaveChanges` procedure written in step 2.
4. Add a form to the application and name it frmLogin. Put controls on its surface like those shown in Figure 6.5. Name the controls lblUserID, txtUserID, lblPassword, txtPassword, cmdOK, and cmdCancel. See Table 6.3 for the initial settings of these controls. In the `Click` event procedure of cmdCancel, hide the current form, setting the contents of the two `TextBoxes` to blank strings. In the `Click` event procedure of cmdOK, unload the current form.

TABLE 6.3

PROPERTIES TO ASSIGN TO OBJECTS ON FRMLOGIN

Object	*Property*	*Value*
Form	Name	frmLogin
Label	Name	lblUserID
	Caption	&User ID
	TabIndex	0
TextBox	Name	txtUserID
	Text	<BLANK>
	TabIndex	1
Label	Name	lblPassword
	Caption	&Password
	TabIndex	2
TextBox	Name	txtPassword
	Text	<BLANK>
	TabIndex	3
CommandButton	Name	cmdOK
	Caption	&Ok
	TabIndex	4
CommandButton	Name	cmdCancel
	Caption	&Cancel
	TabIndex	5

APPLY YOUR KNOWLEDGE

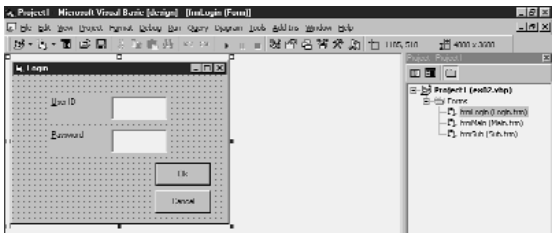


FIGURE 6.5▲
The login form for Exercise 6.2.

5. In frmLogin's `Initialize` event procedure, set the value of txtUserID to "MCP User" and txtPassword to a blank string.

6. Declare two `Private` String variables in frmLogin named sUserID and sPassword. They will hold the user's current entries for UserID and Password. In the `Change` event procedures for txtUserID and txtPassword, update the variables sUserID and sPassword with the contents of each respective `TextBox`.

7. Create a `Public` Function in frmLogin and call it LoginIsValid. LoginIsValid will become a custom method of the form since it's `Public`. Let its return type be `Boolean`. It will return `True` if the string in sUserID is not blank and the string in password is "PASSWORD" (case should not matter). Otherwise, it will return `False`.

8. Put validation code in frmLogin's `QueryUnload` event procedure. If the form is not being unloaded through VB code, then blank out txtUserID and txtPassword (this will effectively render the login invalid). Otherwise (if the form is being unloaded through code), check the return value of the LoginIsValid function. If its return value is `False`, use `MsgBox` to ask users if they wish to retry the login. If they answer that they want to retry, then cancel the unloading of the form.

9. Back in frmMain's `Load` event procedure, put code to display frmLogin modally. After frmLogin runs, get the return value of its `LoginIsValid` method. If frmLogin.LoginIsValid returns `False`, then unload the current form. Otherwise, do nothing and let frmMain continue to load.

10. Add a third form to the project and name it `frmSub`. Give it controls like those shown in in Figure 6.6. Name them lblSubData, txtSubData, and chkRemember. Add a `Private Boolean` variable name blnDirtyFlag and a `Public String` variable (i.e., a custom Property) named `SubData`. In the `Change` event of txtSubData, update the variable `SubData` with the contents of the `TextBox` and set blnDirtyFlag to `True`.

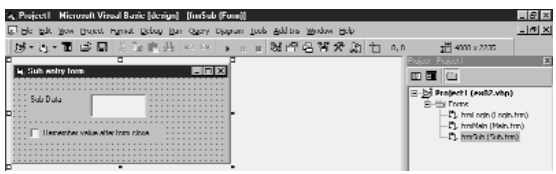


FIGURE 6.6▲
The sub form for Exercise 6.2.

11. Add a `Private Sub` Procedure to `frmSub` and call it SaveChanges. Really, this will not take any save actions but, in this exercise, will simply set blnDirtyFlag to `False`.

12. In `frmSub`'s `DeActivate` event procedure, check the value of blnDirtyFlag. If blnDirtyFlag is `True`, then ask users if they wish to save changes. If they do wish to save, then call the SaveChanges procedure.

13. In `frmSub`'s `Unload` event procedure, check the status of the `Value` property of the chkRemember check box. If it's not checked, then set `frmSub` to `Nothing` (thus effectively destroying the contents

APPLY YOUR KNOWLEDGE

of the variable `SubData`). Otherwise (if it's checked), do nothing, which will leave the contents of `SubData` intact. In `frmSub`'s `Load` procedure, set the contents of txtSubData to the contents of the `SubData` variable.

14. Back in frmMain, put code in the `Click` event procedure of the `CommandButton` cmdSubDataForm to display `frmSub` as a modeless form. (In most applications, you'd display it as a modal form, but, in this case, we want to illustrate the use of `Activate`/`DeActivate` events.)
15. Put code in frmMain's `cmdExit_Click` procedure that will unload frmMain and end the application.

Review Questions

1. What effect will the End statement have for the `Unload` event of forms that are loaded in memory?
2. What VB IDE icon is equivalent to invoking the End statement?
3. What are some considerations for deciding whether to put initialization code in the `Initialize` event procedure versus the `Load` event procedure?
4. Describe the relation between the `Show` method and the `Load` event.
5. What are some considerations for deciding whether to put initialization code in the `Activate` event procedure versus the `Load` event procedure?
6. Compare the circumstances that cause a form's `Activate` and `DeActivate` events to fire versus its `GotFocus` and `LostFocus` events.
7. When will a form's `DeActivate` event *not* fire even though the user clicks on an object outside the form?
8. What two sequential actions will cause a form's `Terminate` event to fire?
9. What parameter do the `QueryUnload` and `Unload` events have in common, and what is the purpose of this parameter?
10. What parameter does the `QueryUnload` event have in addition to the `Unload` event, and what is its purpose?
11. In what type of application do the `QueryUnload` and `Unload` events fire at different times?

Exam Questions

1. If `Form1` is your project's startup form and you put the line

   ```
   form2.Show
   ```

   in `Form1`'s `Load` event procedure,

   A. You'll receive a runtime error.

   B. You'll receive a compiler error.

   C. `Form1` will end up as the active form after all initial code has run.

   D. `Form2` will end up as the active form after all initial code has run.

2. One difference between `QueryUnload` and `Unload` events is that

   A. `QueryUnload` happens first, and `Unload` receives the `UnloadMode` parameter.

   B. `QueryUnload` happens first, and `QueryUnload` receives the `UnloadMode` parameter.

   C. `Unload` happens first, and `Unload` receives the `UnloadMode` parameter.

   D. `Unload` happens first, and `QueryUnload` receives the `UnloadMode` parameter.

APPLY YOUR KNOWLEDGE

3. When the user changes focus to another application, the currently active form in your application

   A. Receives a `DeActivate` event.

   B. Receives a `LostFocus` event.

   C. Receives both `DeActivate` and `LostFocus` events.

   D. Receives neither `DeActivate` nor `LostFocus` events.

4. To make an unloaded form visible to the user,

   A. You need to call only the `Load` statement.

   B. You need to call only the `Show` method.

   C. You must call both `Load` and `Show`.

   D. You must set the `Visible` property.

5. A form's `Terminate` event

   A. Fires whenever the `Unload` event fires, just before the `Unload` event.

   B. Fires whenever the `Unload` event fires, just after the `Unload` event.

   C. Fires when you set the form to `Nothing` after the `Unload` event begins.

   D. Fires whenever you set the form to `Nothing`, regardless of the `Unload` event.

6. If a running VB application has various instances of Child forms and encounters the statement

   ```
   Cancel = False
   ```

   in the MDI Parent form's `Unload` event,

   A. No Child forms will be unloaded.

   B. All Child forms will be unloaded but not the MDI Parent form.

   C. The Child forms will unload as well as the MDI Parent form.

   D. A runtime error will occur.

7. If a running MDI application has various instances of Child forms and encounters the statement

   ```
   Cancel = True
   ```

   in the `Unload` event of one of the Child forms, then

   A. All Child forms whose `Unload` events occurred before this one are unloaded, and the unloading process halts.

   B. All Child forms in the application will unload, except for the Child with `Cancel = True` in its `Unload` event.

   C. The MDI Parent form and all Child forms with `Cancel = True` in their `Unload` events will stay loaded. All other Child forms will unload.

   D. A runtime error will occur.

8. A `DeActivate` event fires (pick all that apply)

   A. Whenever a form unloads.

   B. When the currently active form unloads.

   C. When the user navigates between forms in the same application.

   D. Whenever the application terminates.

9. A `Terminate` event signifies that (pick two)

   A. The application is closing the form just before ending.

   B. The form has just become invisible and now the `Unload` event is about to fire.

APPLY YOUR KNOWLEDGE

C. The instance of the form has been destroyed.

D. All form variables' values have lost their values.

10. If the form `frmMy` has not been loaded in the application, then the following lines of code run in the order given:

    1) `Load frmMy`

    2) `frmMy.Show vbModal`

    3) `Unload frmMy`

    4) `Set frmMy = Nothing`

    5) `frmMy.Show`

    6) `Unload frmMy`

    7) `frmMy.Show`

11. `frmMy`'s `Initialize` event will fire (pick all that apply)

    A. After line 1

    B. After line 2

    C. After line 5

    D. After line 7

11. A form named `frmInfo` has not yet been loaded in memory during the current session of the application. It has a `Public String` variable, Status. If the following code runs:

    ```
    frmInfo.Status = "UNOPENED"
    ```

    which of the following events will fire?

    A. The `Initialize` event, followed by the `Load` event

    B. The `Load` event only

    C. The `Initialize` event only

    D. No event

Answers to Review Questions

1. Invoking the End statement will immediately terminate the application without running any further events. Therefore the `Unload` events of loaded forms will not run. See "`DeActivate`, `Unload`, `QueryUnload`, and `Terminate` Events."

2. The "Stop button" in the VB IDE has the same effect as calling the End statement from code. Therefore, if you press this button to end a design-time instance of your program, you will not fire the `Unload` events of forms or of other ending events such as `QueryUnload` and `Terminate`. See "`DeActivate`, `Unload`, `QueryUnload`, and `Terminate` Events."

3. You should put code in the `Initialize` event procedure to assign the beginning values of the form's `Public` variables or of `Private` variables that represent the stored values of form custom properties (properties implemented with Property procedures). This will make the form's behavior consistent with other VB classes because the `Initialize` event behaves like the `Initialize` event of any other VB class. See "The `Initialize` Event."

4. The `Show` method will cause an *implicit load* of a form, if the form was not already in memory, and will fire the `Load` event. If the form was already in memory, the `Show` method merely makes it visible but does not fire the `Load` event. Depending on the argument you pass to the `Show` method, the form will display modally as a dialog box (`vbModal`) or modelessly (`vbModaless`-the default). Note: Referring to a method or property of a form can cause an implicit load. See "Implicitly Loading a Form."

APPLY YOUR KNOWLEDGE

5. You should put code in the `Activate` event procedure if: a) you want it to run every time the user makes the form the active form in the application, or b) if you need to perform initialization tasks that require the form to already be loaded in memory (such as drawing graphics or using a data control's connection). Otherwise, it's okay to put initialization code in the `form_Load` event procedure. See "The `Load` Event and the `Activate` Event."

6. The `Activate` and `DeActivate` events fire whenever focus changes in the current application between the current form and another form. The `GotFocus` and `LostFocus` events fire only if there is no other object on the form capable of receiving focus. See "`Activate`/`DeActivate` Versus `GotFocus`/`LostFocus` Events."

7. A form's `DeActivate` event will *not* fire when the user navigates to another application with the mouse. See "The `DeActivate` Event."

8. You can cause a form's `Terminate` event to fire by unloading the form and then setting the form equal to `Nothing` in code. If you have any other Form object variables that refer to the form, they must be set to `Nothing` as well before `Terminate` will fire. See "The `Terminate` Event."

9. The `QueryUnload` and `Unload` events both have a `Cancel` parameter that the programmer can set to `True` to halt the unloading action. See "The `QueryUnload` Event," "The `Unload` Event," and "Using the `Unload` and `QueryUnload` Events in an MDI Application."

10. The `QueryUnload` event also has an `Action` parameter whose purpose is to advise that the unloading is taking place. See "The `QueryUnload` Event."

11. The `QueryUnload` and `Unload` events fire at different times in an MDI application when the MDI Parent form `Unload` events happen in the following order: 1) The MDI Parent's `QueryUnload`; 2) all the loaded children's `QueryUnload` events; 3) all the loaded children's `Unload`; 4) the `Unload` event of the MDI Parent. See "Using the `Unload` and `QueryUnload` Events in an MDI Application."

Answers to Exam Questions

1. **C**. `Form1` will end up as the active form after all initial code has run. It's okay to call another form's `Show` method from a `Load` event procedure, so A and B are incorrect. `Form1` ends up as the active form when everything is finished because the following actions happen: 1) `Form1`'s `Load` event procedure begins; 2) `form2`'s `Show` method is called from within `Form1`'s `Load` event procedure; 3) `form2`'s `Load` event fires (it receives an implicit load due to the `Show` method), and `form2` briefly becomes the active form (so its `Activate` event also fires); 4) then `Form1`'s `Load` event finishes, `Form1` becomes the active form, and its `Activate` event fires. See "Manipulating a Form From Another Form's `Load` Event Procedure" for more information.

2. **B**. A form's `QueryUnload` event always precedes its `Unload` event. The `QueryUnload` event receives the `Cancel` and `UnloadMode` parameter while the `Unload` event receives only the `Cancel` parameter. Note that, in the case of MDI Child forms, a form's `Unload` event might not directly follow its `QueryUnload` event: when the main MDI form unloads, all Child form `QueryUnload` events happen together followed by all Child form `Unload` events. See "The `QueryUnload` Event," "The `Unload` Event," and "Using the `Unload` and `QueryUnload` Events in an MDI Application."

APPLY YOUR KNOWLEDGE

3. **D**. Receives neither `DeActivate` nor `LostFocus` events. Although `DeActivate` and `LostFocus` fire when the form loses its status as the active form, a form is the active form only with respect to the application where it is running. So when the user moves to another application, the active form of the current application does not change, and these two events do not fire. See "The `DeActivate` Event" and "`Activate`/`DeActivate` Versus `GotFocus`/`LostFocus` Events" for more information.

4. **B**. You need to call only the Show method. The Load statement by itself will only put the form in memory but will not make it the active form nor will it cause it to be visible. Toggling the Visible property will manipulate the form's visibility, but it is not the only way nor even the preferred way to do this (Show and Hide methods are preferred). The Show method will make the form visible, loading it into memory with an implicit load if it had not been loaded before. See "`Show`/`Hide` Methods Versus `Load`/`Unload` Statements" and "Implicitly Loading a Form."

5. **C**. The `Terminate` event fires when you set the form to `Nothing` after the `Unload` event begins. The `Terminate` event cannot fire before the form unloads and does not fire until the form is destroyed by setting it to `Nothing`. The `Unload` event by itself does not destroy a form completely since it leaves the form's properties in memory. Therefore, the `Unload` by itself cannot trigger the firing of the `Terminate` event. See "The `Terminate` Event."

6. **C**. The Child forms will unload as well as the MDI Parent form. This is sort of a trick question based on the fact that the `Cancel` parameter's default property is `False` which, of course, means that unloading will proceed normally. Therefore, setting the `Cancel` parameter to `False` has no effect. If the MDI Parent's `Unload` event runs without getting canceled (as it will in this case), this means all children have already unloaded. See "The `Unload` Event" and "`Using` the `Unload` and `QueryUnload` Events in an MDI Application."

7. **A**. All Child forms whose `Unload` events occurred before this one are unloaded, and the unloading process halts. Setting `Cancel` to `True` in an `Unload` event halts all unloading actions, including the current one, but it's too late to halt the unloads that have already occurred. See "Using the `Unload` and `QueryUnload` Events in an MDI Application."

8. **C**. A `DeActivate` event fires only when the currently active form or an object on the currently active form loses focus to another form in the current application. `DeActivate` will not occur when a form unloads. Neither will `DeActivate` occur when an application terminates. See "The `DeActivate` Event."

9. **C, D**. The `Terminate` event signifies that an instance of the form has been destroyed and all the form's variables have lost their values. A is incorrect because `Terminate` has nothing to do with the lifetime of the application, and B is incorrect because the `Terminate` event cannot occur before the `Unload` event fires. See "The `Terminate` Event."

10. **A, C**. The `Initialize` event fires after line 1 (`Load frmMy`—this is the first loading of the form, and so the instance is created here) and after line 5 (`frmMy.Show`—this line causes the form to initialize, because it had been destroyed in the previous line by setting the form to `Nothing` after it was unloaded).

APPLY YOUR KNOWLEDGE

Initialize doesn't fire after line 2 (`frmMy.Show vbModal`) because the form has already been initialized by the previous `Load` statement. It doesn't fire after line 7 (`frmMy.Show`) because, even though the form was just unloaded before this, its variables were not destroyed by setting it to `Nothing`. Therefore the form remained initialized. See "The `Initialize` Event."

11. **C**. The `Initialize` event only. Although making any reference to a form's properties or methods will normally cause an implicit Load, there is an exception to this rule for calls to a form's *custom* members (i.e. properties implemented as `Public` variables, methods implemented as Public procedures, or properties implemented with Property procedures). In such cases, no implicit load occurs. The `Initialize` event will still run in such cases, however, provided the form is not already instantiated. See "The `Initialize` Event" and "Implicitly Loading a Form."

OBJECTIVE

This chapter helps you prepare for the exam by covering the following objective and its subobjectives:

Implement online user assistance in a distributed application (70-175 and 70-176).

- **Set appropriate properties to enable user assistance. Help contents include `HelpFile`, `HelpContextID`, and `WhatsThisHelp`.**
- **Create HTML Help for an application.**
- **Implement messages from a server component to a user interface.**

- The basic idea behind the exam objective covered in this chapter is the display of Help information to a user.
- Assuming that a standard HTML Help or `WinHelp` file already exists for your application, VB makes it easy for you to link up the Help file's topics (which have unique internal identifying numbers that you must know) to various types of context-sensitive help in VB. You can set special VB properties to change the Help file and to cause Help to appear in various ways when the user presses the F1 key or performs other actions.
- This chapter also briefly discusses how to create an HTML Help file that you can use in your VB applications. HTML Help is Microsoft's new standard for Help file formats.
- The third subobjective listed, "Implement messages from a server component to a user interface," is more appropriate for a discussion of COM components. We therefore discuss that subobjective in Chapter 12, "Creating a COM Component that Implements Business Rules or Logic" in the section, "Sending Messages to the User from a COM Component."

CHAPTER 7

Implementing Online User Assistance in a Distributed Application

OUTLINE

Two Types of Help Files **267**

HTML Help Files 267

`WinHelp` Files 267

Referencing Help Through the `HelpFile` Property of an Application **268**

Setting Help Files at Design Time 269

Setting Help Files at Runtime 270

Context-Sensitive Help for Forms and Controls **271**

Context-Sensitive Help With the `HelpContextID` Property 271

Adding ToolTips to an Application 272

Providing `WhatsThisHelp` in an Application 273

Creating HTML Help **276**

HTML Help Source File Structures 276

Creating and Compiling an HTML Help File Project With HTML Help Workshop 277

Chapter Summary **293**

STUDY STRATEGIES

- Using an existing Help file whose contents you know (such as the sample HTML and WinHelp files supplied with this book) to experiment with VB's `App.Helpfile` property and with the other Help properties discussed in this chapter.
- Following Exercise 7.1 to get some hands-on experience with VB's Help properties.
- Downloading and becoming familiar with the HTML Help Workshop (see Appendix F, "Suggested Readings and Resources"). Be warned that this product has a semi-standard, somewhat half-baked user interface, and cryptic, poorly designed, and incomplete help. Most of the development of an HTML Help system can be done with a text editor, but a few steps are more easily accomplished with the HTML Help workshop.
- Becoming familiar with the sections in this chapter under "Creating HTML Help" and Exercises 7.2 and 7.3. This should give you the information and experience to be able to understand HTML Help creation sufficiently to overcome the shortcomings of the HTML Help Workshop interface.

Two Types of Help Files

We live in exciting times, especially those of us who work with Help files on Windows operating systems. Microsoft is in the midst of a transition between its older Help file format, WinHelp, and its Help file format of the future, HTML Help. Although the certification exam focuses on the HTML Help format, we briefly discuss both formats here.

HTML Help Files

HTML Help is a newer format that Microsoft now uses as its standard Help file format. Figure 7.1 shows an example of an HTML Help file displayed on a user's screen. HTML Help files have a `chm` extension. In order to display HTML Help files on a user's system, the user's system must be set up properly with the correct Registry entries and support files.

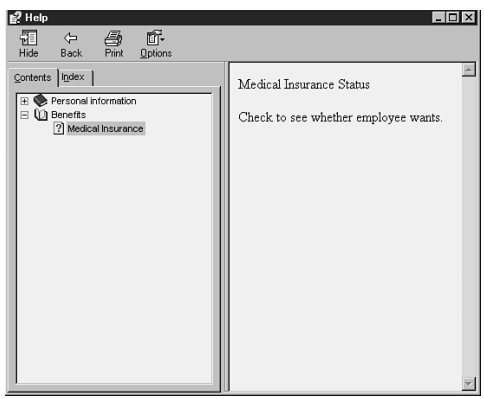


FIGURE 7.1
An HTML Help screen as seen by an end user.

We discuss the creation of HTML Help files in this chapter under the section entitled "Creating HTML Help" as the exam objective.

`WinHelp` Files

`WinHelp` is Microsoft's older Help file format. Figure 7.2 shows an example of an HTML Help file as displayed on a user's screen.

`WinHelp` files have an `hlp` extension. In order to display `WinHelp` files on a user's system, the `WinHelp` engine, `Winhlp32.exe`, must reside on the user's system.

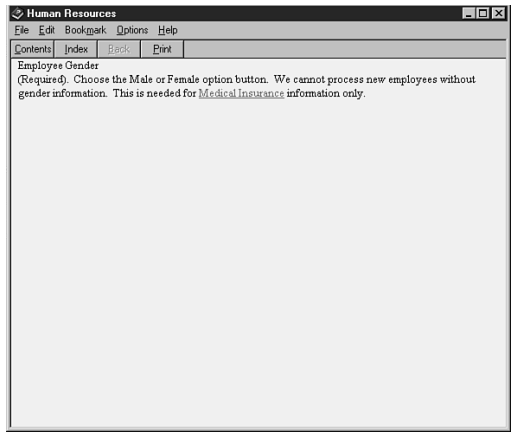


FIGURE 7.2
A `WinHelp` screen as seen by an end user.

In order to create `WinHelp` files, a developer must know how to use Microsoft's Help Compiler program, hc.exe, to create compiled .hlp files. Before compiling the .hlp file, the developer must create one or more special .rtf-format files containing specific tags and formatting, as well as a text file with extension .hpj, that serves as a header file for the Help project.

By all appearances, Microsoft intends that `WinHelp` files fade from the scene.

Referencing Help Through the HelpFile Property of an Application

The simplest way to implement Help in an application is to tell the application the name and location of a Help file. The user can then get help by pressing the F1 key.

Visual Basic lets the developer specify an HTML Help or a `WinHelp` file as the project's Help file either at design time or at runtime.

For most applications, it is sufficient to identify the Help file during development and to distribute the Help file with the application. Occasionally the application's Help file cannot be included at development time because the name of the file is unknown, the file doesn't exist, or because different users will have different files. In these cases, the Help file can be specified at runtime. This section describes both ways of identifying the Help file to an application.

Setting Help Files at Design Time

The easiest way to include Help with Visual Basic is to identify the Help file at design time. This is done through the Properties window for that project, as shown in Figure 7.3. To reference Properties for a project, choose Properties from the Project menu in the development environment.

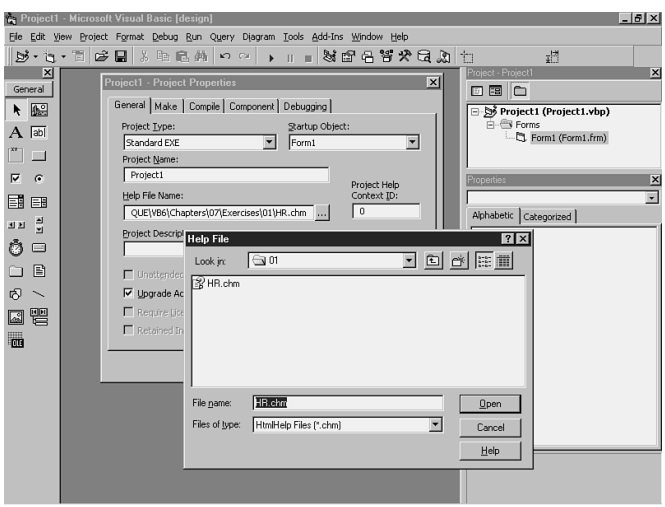


FIGURE 7.3
Setting an application's Help file with the Project Properties dialog box.

On the General tab of the Project Properties dialog box, the name and path of the Help file will go under Help File Name. If the *.hlp or *.chm file will be in the same directory as the application, it is not necessary to include the full path to the file. The application will use the same search rules to find the file as other Windows applications. Visual Basic will let you specify only one Help file for the application. To use more than one Help file, you will have to change Help files at runtime.

Setting Help Files at Runtime

The second way to include Help in an application, specifying the filename at runtime, is more flexible than setting the project properties for the file at design time. With this method, different users have the ability to reference different Help files, or a single user can use different Help files depending on how the application is being used.

As an example, imagine that you are creating an accounting system that will be used by both a data entry group and the financial analysis staff of a company. The data entry group will use the windows of the application to enter and change data in the system. The financial analysis group may require read-only access to the data. You may want to provide customized Help files to each group, describing their own needs within the application. If you limit yourself to setting the Help file of the application just at design time, you could only provide a single Help file. By setting the Help file at runtime, you are not constrained to a single file.

You can use the `HelpFile` property of the `App` object to set a reference to a Help file at runtime. The `App` object is a global object available to the application. It provides information about the application as well as the means to change some of the characteristics of the application. The `HelpFile` property initially identifies the Help file that was entered into the Project Properties dialog box at design time (refer back to Figure 7.3). `App.Helpfile` is a read/write property at runtime, so you can reset the Help file after the application has started.

You can set a Help file at runtime with code as follows:

```
App.HelpFile = filename
```

`Filename` is the Help file's name, including, if necessary, a path to that file. To set a reference to your own Help file, for example, you might write the following code:

```
App.HelpFile = 'C:\MyPath\MyHelp.chm'
```

Usually, if an application needs to set the Help file path at runtime, it will get the data it needs from previously stored information in the System Registry. Occasionally, the user may need to specify the file name, and sometimes the path to that file. The `HelpFile` property is a text string that describes the path and file. If the path is not defined, the application follows the standard Windows search methods just as it does when you identify the file at design time.

Context-Sensitive Help for Forms and Controls

Providing context-sensitive help in an application can save users time trying to find information instead of making them navigate through a Help Contents page or an Index.

Context-sensitive help gets the user to the required information quickly and directly by being "aware" of which control has focus when the user presses F1.

The following sections discuss three easy ways to add context-sensitive help to your application:

- `HelpContextID` Help. This type of help uses the traditional F1 key to show help information to a user.
- ToolTips. This type of help shows brief messages to the user when the mouse pauses over an object.
- `WhatsThisHelp`. This type of help uses the WhatsThis icon to let users query the meaning of an object.

NOTE

VB Uses `WinHelp` and HTML Help Files Identically. When you implement context-sensitive help as described in the following sections, you don't need to worry about whether the project's Help file is a WinHelp or an HTML Help file: both file types furnish Topic IDs for their help topics, and the VB programmer uses the two different file types in exactly the same way to provide context-sensitive help, as described below.

Context-Sensitive Help With the `HelpContextID` Property

Each standard visible object in Visual Basic, including the menus and forms, has a `HelpContextID` property. The `HelpContextID` property is a numeric value corresponding to a Context ID or Topic ID in a Help file. The default value for `HelpContextID` is 0, meaning no context help is provided. The `HelpContextID` for an object is usually specified at design time, but can be defined at runtime as well.

The `HelpContextID` property of an object maps to a Topic ID and its corresponding topic in a Help file. At runtime when that object is active and the F1 key is pressed, Windows will open the Help file for the application with a specific topic displayed. Visual Basic applications have a hierarchy that is followed to determine which topic to display when Help is invoked. When the F1 key is pressed, the application does the following:

NOTE

Context ID or Topic ID? The `HelpContextID` and `WhatsThisTopicID` properties discussed in this chapter must match Context ID numbers or Topic ID numbers in the corresponding Help file. HTML Help documentation uses the term "Topic ID" while WinHelp documentation uses the term "Context ID" to refer to these topic-mapping numbers. We shall use the term Topic ID in this chapter because that's the term used for HTML Help, the Microsoft standard.

1. It checks the `HelpContextID` of the active control on the active form. If that ID is non-zero, it opens the application Help file with the corresponding topic displayed.
2. If the `HelpContextID` is zero,Visual Basic checks the `HelpContextID` of the container control for the active object. Usually this is the active form itself. It can also be another container control such as `Frame` or `PictureBox`. If the ID of the container is non-zero, Help is opened with the container's corresponding Help topic displayed.If the `HelpContextID` for the container is also zero, then the application checks the container's container for a `HelpContextID`.
3. Visual Basic keeps checking the `HelpContextID`s of objects' containers, and then of containers' containers, and so on, until it gets to the form that is the highest-level container. If a non-zero `HelpContextID` is found along the way, that help topic is displayed.
4. If all `HelpContextID`s through the form level are zero, the Help file for the application is opened to the Contents page for a WinHelp file (refer back to Figure 7.2) or, for an HTML Help file (refer to Figure 7.1), the default topic or the contents page if no default topic exists.

The `HelpContextID`s of objects must correspond to the Topic IDs of the Help file specified in the Project Properties window (refer back to Figure 7.3) or to a Help file previously identified to the application through the `App.HelpFile` property. If a non-zero `HelpContextID` for an object exists but a Help file has not been specified for the application, the user will get no response when the F1 key is pressed. Help is displayed only if the Help file has been set in the application and a Topic ID has been defined for a control. If a Topic ID corresponding to the `HelpContextID` does not exist in the Help file, the Windows Help engine will display an error stating that `the help topic does not exist`.

Adding ToolTips to an Application

A fast and simple way of adding Help information to an application without using a Help file is by providing ToolTips to the users.

ToolTips, which are becoming more and more common in Windows applications, are little bits of information that appear when the user rests the mouse pointer over an object in a window.

Visual Basic 6 is a good example of an application that provides ToolTips. If you rest the mouse pointer over a button on the toolbar or a control in the toolbox, a message identifying the purpose of the object appears. You can also easily implement such ToolTip Help in your own project. Figure 7.4 shows an example of ToolTip Help in a VB application.

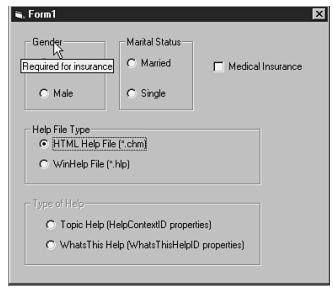


FIGURE 7.4
An example of ToolTips in Visual Basic 6.

Adding ToolTips to an application is easy in Visual Basic. Just set the `ToolTipText` property of each control to the text you want displayed when the mouse pointer is over that control.

If an application performs calculations and displays results in a `Label` control, for example, you might want to explain to the user how the result was determined. This can be done by putting the formula used in the calculation into the `ToolTipText` property of the `Label` control. When the user rests the pointer over the `Label`, the formula will appear.

You may also want to add ToolTips to command buttons to identify the purpose of each button to the user. Instead of placing large amounts of descriptive text in the `Caption` of a `CommandButton`, you can provide additional help to the user through ToolTips. By doing this the captions of the buttons are kept simple and, after the user is familiar with the functionality of a button, that user will no longer need to use the ToolTips.

If an application has ToolTips in a `ToolBar` or a `TabStrip` control, the `ShowTips` property of these controls must be set to `True` for the tips to appear at runtime. Controls that do not have a visible interface at runtime, such as the `Timer` or `CommonDialog` controls, do not have a `ToolTipText` property.

Providing `WhatsThisHelp` in an Application

When ToolTips don't provide enough information and you don't want to force the user to toggle between a Help file and your application, Visual Basic provides another means of displaying tips to the user.

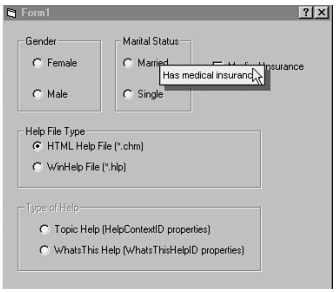


FIGURE 7.5▲
`WhatsThisHelp` as it appears in a Visual Basic application. Note the similarity to ToolTips.

`WhatsThisHelp` gives the means to pop-up information from the `App.HelpFile` object to the user in the same format used by ToolTips (see Figure 7.5).

Unlike ToolTips, `WhatsThisHelp` gets the pop-up information from a topic within a Help file so more extensive explanations of objects can be provided. Also, unlike context-sensitive Help discussed earlier in the chapter, the Help file does not open in a separate window. `WhatsThisHelp` pops up for the user and is visible only until the user clicks on the application window again.

`WhatsThisHelp` Property

`WhatsThisHelp` usually functions on the form level in Visual Basic. The user typically invokes `WhatsThisHelp` by clicking on a menu item on the form and then by selecting the object for which help is desired.

If the developer wants to implement `WhatsThisHelp` on a form, the `WhatsThisHelp` property of the form must be set to `True`. This is done by the developer at design time.

The `WhatsThisHelp` property has to be set to `True` for any of the `WhatsThisHelp` methods discussed in this section to work.

`WhatsThisMode` Method

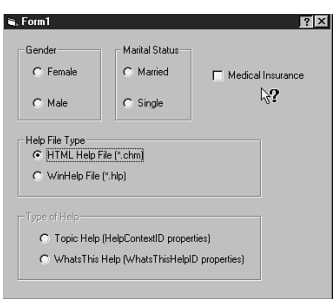


FIGURE 7.6▲
The `WhatsThisHelp` mouse pointer.

After you have decided to implement `WhatsThisHelp` on a form, you must provide a means for the user to invoke the pop-up information. The `WhatsThisMode` method of a form will start the process of `WhatsThisHelp` in an application.

If you provide a `WhatsThis` menu item, the code to do this would be in the `Click` event procedure of the `WhatsThis` menu item and would look something like this (assuming the `WhatsThisHelp` property of `Form1` is set to `True`):

```
Form1.WhatsThisMode
```

Invoking the `WhatsThisMode` method will automatically change the appearance of the mouse pointer to let the user know that normal actions have been suspended while in the Help mode (see Figure 7.6). The mouse pointer remains like this until the user clicks on an object on the form.

WhatsThisHelpID Property

After WhatsThisHelp has been invoked and the WhatsThis mouse pointer is showing (refer again to Figure 7.6), the user can select an object on the form to get help for that object. When this happens, Windows uses the WhatsThisHelpID of the object to determine what information will be displayed. The WhatsThisHelpID maps to a topic in a Help file in the same way that a HelpContextID does. The only difference is that with the WhatsThisHelpID, the Help information appears in a pop-up window and not in a separate dialog box. As soon as the user clicks again in the application, the pop-up window goes away.

WhatsThisButton Property

A second way of invoking WhatsThisHelp is from a button on the title bar of a form, as shown in Figure 7.7. If you set the WhatsThisButton property to True, you will not need to invoke the WhatsThisMode method as described earlier. Instead Windows controls the invocation of Help mode.

The following conditions must be true for the WhatsThisButton to appear on a title bar:

1. The WhatsThisHelp property of the form must be True, and one of the following must be set:
   - The ControlBox property of the form must be True.
   - The MinButton and MaxButton properties of the form must be False.
   - The Borderstyle of the form must be either Fixed Single or Sizeable.
2. The BorderStyle of the form must be Fixed Dialog if it is not Fixed Single or Sizeable.

If any of the three conditions above are not met, the WhatsThisButton will not appear for the form.

The WhatsThisHelp and WhatsThisButton properties of the form can only be written at design time. They're available at runtime as read-only properties.

NOTE

WhatsThisHelp Topics in an HTML Help Project As discussed later in this chapter under the section "HTML Help Source files for WhatsThisHelp," WhatsThisHelp topics do not look the same to a Help file developer as extended topics used with HelpContextIDs.

WhatsThisHelp topics from a WinHelp file, on the other hand, are the same in the WinHelp project as extended topics used with HelpContextIDs.

However, as noted earlier, the VB programmer using a Help file sees no difference here. To the VB programmer, it is all a matter of using the HelpContextID property or the WhatsThisHelpID property, regardless of the format of the Help file.

FIGURE 7.7
A WhatsThisButton on the title bar of a form.

`ShowWhatsThis` Method

The third way of displaying `WhatsThisHelp` in an application is by using the `ShowWhatsThis` method of a control. Code such as

```
Command1.ShowWhatsThis
```

will show help for the topic defined by the property `Command1.WhatsThisHelpID`. For `ShowWhatsThis`, the user does not have to click on the `WhatsThisButton` on the title bar or select a menu item to go into `WhatsThis` mode. You simply cause the `WhatsThisHelp` topic for the given object to appear by calling it up in your code.

`ShowWhatsThis` is usually invoked with a right-mouse pop-up menu on a control.

As with the other methods of showing `WhatsThisHelp`, the `WhatsThisHelp` property of the form containing the controls with `WhatsThisHelp` must be set to `True`.

Creating HTML Help

The Microsoft Exam Objectives for VB6 require you to know the fundamentals of HTML Help creation. In the following sections, we discuss the basic structure of the files that go into making up a compiled HTML Help file, as well as how to use Microsoft's HTML Help Workshop to manipulate and compile these files.

HTML Help Source File Structures

An HTML Help project is similar to a VB project in several ways:

- Its end product (the *.exe file in VB and the *.chm file in HTML Help) is generated by a compiler from source code files.
- The source code files are created by the programmer in text code format within a development environment.
- There can be various source code files of different types (*.frm, *.bas, *.cls, and *.ctl files would be examples in VB, while *.htm, *.hhk, and *.h files would be examples in HTML Help).
- The source code files for a single project are tied together by being listed together in a header file (*.vbp file in VB, *.hhp

file in HTML Help). The header file also lists other general information about the project, such as the name and destination of the compiled file.

We discuss the formats and roles of the various types of source code files in the following sections.

Creating and Compiling an HTML Help File Project With HTML Help Workshop

You can use Microsoft's HTML Help Workshop to create, maintain, and compile the various files needed for an HTML Help project.

To create a new HTML Help project with an hhp header file, you should take the following steps:

NOTE

Downloading the Microsoft HTML Help Workshop As of this writing, you can obtain HTML Help Workshop as a free download from Microsoft's Web site at

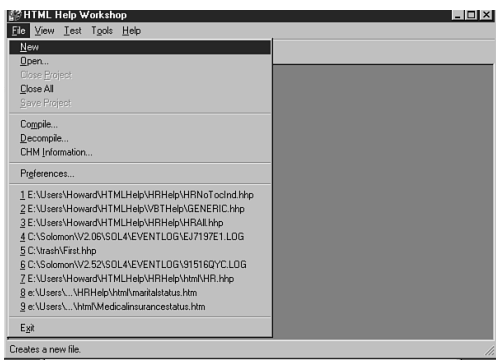
`www.microsoft.com/workshop/author/htmlhelp/default.asp`

The text on the Web page refers to the file as "HTML Help 1.1." The name of the file to download is `htmlhelp.exe`.

STEP BY STEP

7.1 Creating a New HTML Help Project With the HTML Help Workshop

1. Run the HTML Help Workshop application and click the File, New menu option (see Figure 7.8).



◀**FIGURE 7.8**
The HTML Help Workshop File menu dialog box.

2. On the resulting New dialog box screen, choose Project as the type of object that you wish to create (see Figure 7.9) and click the OK button to continue.

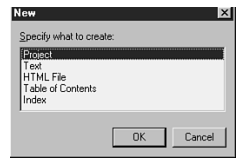


FIGURE 7.9▲
The HTML Help Workshop's New dialog box.

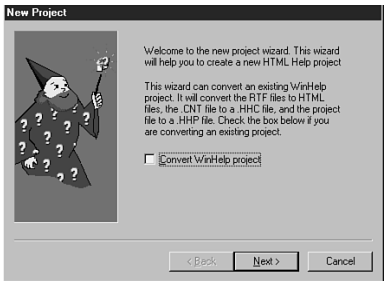


FIGURE 7.10▲
The First screen of the HTML Help Workshop's New Project Wizard.

3. You will see the first screen of the New Project Wizard (see Figure 7.10). Leave the `Convert WinHelp project` check box unchecked and proceed to the next screen.

4. On the New Project Wizard's Destination screen, click the Browse button to specify the name and location of the new HTML Project Header file (*.hhp) that you wish to create (see Figure 7.11). After you've selected a location for the new project header file, you can proceed to the next screen.

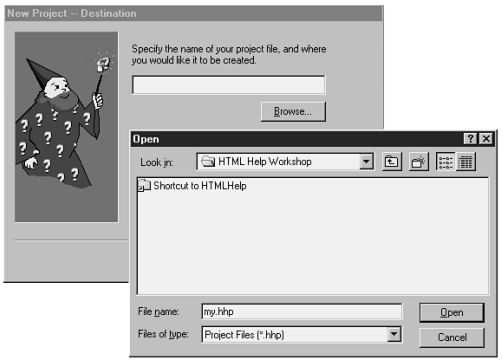


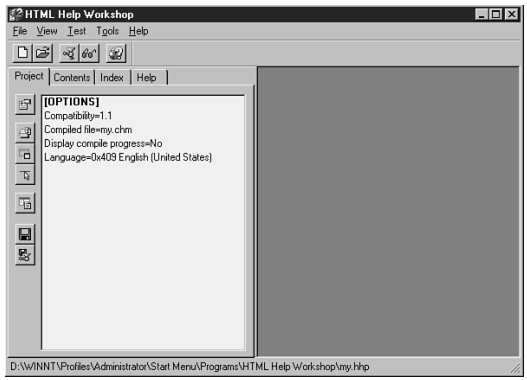
FIGURE 7.11▶
Specifying the name of a new HTML Help project file (*.hhp).

FIGURE 7.12▲
The HTML Help Workshop New Project Wizard's Existing Files dialog box.

5. For the time being, you can leave all the options on the New Project Wizard's Existing Files dialog box see (see Figure 7.12) unchecked and proceed to the next screen.

6. Click the Finish button on the New Project Wizard's final screen. You should then see general information for the new project in the left-hand pane of the HTML Help Workshop, as shown in Figure 7.13.

7. The new project has now been initialized with a Project Header file (*.hhp). You can save changes to the project by answering the prompt when you exit the HTML Help Workshop or by choosing the File, Save Project menu option.



◀**FIGURE 7.13**
The HTML Help Workshop just after starting a new project.

The HTML Project Header File

The HTML Project Header file (*.hhp) is a text file divided into sections. Each section is headed by its title that is enclosed in square brackets. Listing 7.1 shows an example of an HTML Project Header file, while Figure 7.14 shows what the same file looks like when viewed in the HTML Help Workshop.

LISTING 7.1

CONTENTS OF A SAMPLE HTML PROJECT HEADER FILE

```
[OPTIONS]
Compatibility=1.1
Compiled file=HR.chm
Display compile progress=No
Language=0x409 English (United States)

[FILES]
html\Gender.htm
html\maritalstatus.htm
html\Medicalinsurancestatus.htm

[ALIAS]
IDH_GENDER=html\Gender.htm
IDH_MARITALSTATUS=html\Gender.htm
IDH_MEDICALINSURANCE=html\Gender.htm

[MAP]
#include HR.h

[TEXT POPUPS]
CSHelp.h
CSHelp.txt
```

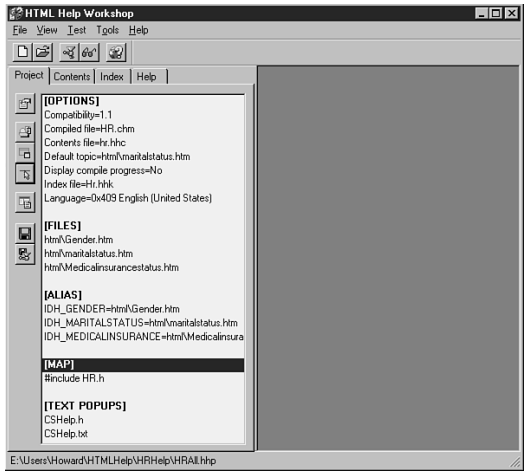


FIGURE 7.14▶
The HTML Help Workshop with a full-featured Help project loaded.

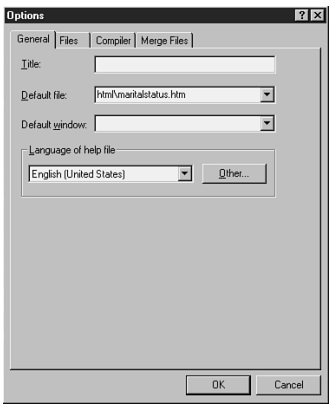


FIGURE 7.15▲
The HTML Help Workshop's Options tabbed dialog box.

The `[OPTIONS]` section gives general project information, including the name of the compiled HTML Help file, the name of the files for the Contents and Index, the default HTML topic file contents to display if the Help file is called without a ContextID, and the language in which to display prompts and captions. You can manipulate the project's `[OPTIONS]` from within the HTML Help Workshop by clicking the Change Project Options icon (the topmost Toolbar button on the vertical toolbar along the left of the HTML Workshop's main screen) and filling in the appropriate information in the resulting Options tabbed dialog box, as shown in Figure 7.15.

The other *.hhp file sections (explained more at length under the sections in this chapter entitled "HTML Help source files for Extensive Help" and "HTML Help source files for `WhatsThisHelp`") include lists of different types of files that contain the actual Help information to be shown by the system.

HTML Help Source Files for Extensive Help (`HelpContextID`)

In order to implement Help topics in separate screens with VB's `HelpContextID` property, the underlying help project must have separate topics defined and each topic must have its own unique identifying number for use by VB's `HelpContextID`.

Each Topic is defined in a separate HTML (*.htm) file, and the association between Topic files and `HelpContextID` numbers is accomplished by one or more mapping files (*.h) and the [`FILES`], [`ALIAS`], and [`MAP`] sections of the project file. A brief summary of the role of each of these elements follows:

- *Topic files (*.htm)* contain the actual content of help topics. They are in HTML (Web page) format.
- *The [`FILES`] section* in the *.hhp file specifies all the Topic files used in the help project.
- *Mapping files (*.h)* define the Topic ID numbers that will be used by VB to identify topics in the help file and assign a uniquely named constant to each Topic ID.
- *The [`MAP`] section* in the *.hhp file specifies all the mapping files used in the help project.
- *The [`ALIAS`] section* in the *.hhp file ties together the named constants defined in the mapping files (*.h) with individual Topic files (*.htm).

We discuss the use of each of these elements in the following sections.

If you want to enable users to navigate to other Help file topics while viewing individual Help topics, then you also must supply either an Index or a Contents file (developers often supply both).

Topic Files and the [`FILES`] Section

Each Topic file represents a single Help file screen topic that can be associated with a `HelpContextID` for use in VB, as discussed in the section entitled "Context-Sensitive Help With the `HelpContextID` Property." Figure 7.16 shows a Help topic from an HTML Help file displayed on a user's screen.

A Topic file is really just a Web page, or *.htm file, as illustrated in Listing 7.2. Just like any other HTML file, the Topic file can contain references to URLs on the Web.

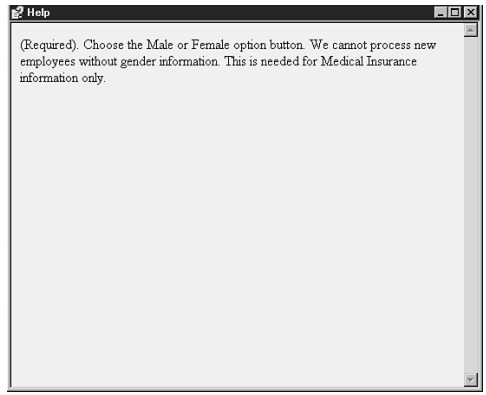


FIGURE 7.16
A Help topic from an HTML Help file displayed on a user's screen.

LISTING 7.2

SAMPLE HTML HELP TOPIC FILE CONTENTS

```
<HTML>
<HEAD>
<title>Gender of Employee</title>
</HEAD>
<BODY>
<p>
(Required). Choose the Male or Female option button.
We cannot process new employees without gender information.
This is needed for Medical Insurance information only.
</p>
</BODY>
</HTML>
```

You must include every Topic file you wish to use in your Help project in the [FILES] section of the *.hhp project header file. To add information to the [FILES] section, you can follow these steps:

STEP BY STEP

7.2 Adding Topic Files to the [FILES] Section of an HTML Help Project File

1. Make sure that the HTML Help Workshop application is running with the appropriate HTML Help project (*.hhp) file loaded.

2. Click the Add/Remove Topic files icon (the second Toolbar button from the top on the vertical toolbar along the left of the HTML Workshop's main screen). This will bring up the Topic Files dialog box, as shown in Figure 7.17.

3. Click the Add button on the Topic Files dialog box to bring up a Browse screen, as shown in Figure 7.18. Browse to an *.htm file that you wish to add to your project as a help topic. Repeat this step for each of the *.htm files that you wish to add to the project.

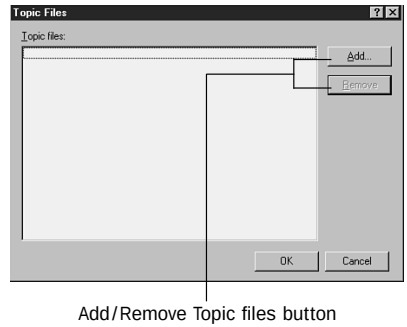
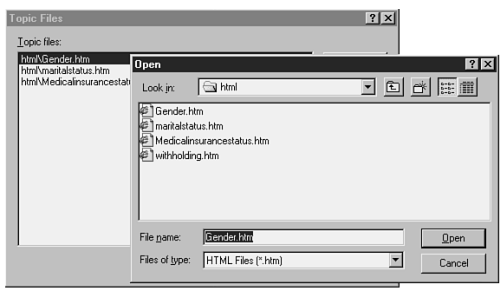


FIGURE 7.17▲
HTML Help Workshop's Topic Files dialog box.



◀FIGURE 7.18
Adding a Topic file to the Help project from the HTML Help Workshop New Project Wizard's Topic Files dialog box.

4. Once you're finished adding Topic files, the Topic Files dialog box should show a list of the files you've added, as shown in Figure 7.19. Click OK on the Topic Files dialog box.

   You should now see a [FILES] section in the project window (refer to Figure 7.14).

5. Be sure to save the project file.

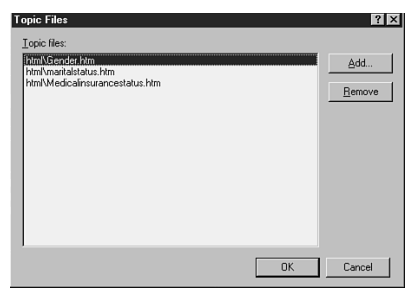


FIGURE 7.19▲
The HTML Help Workshop New Project Wizard's Topic Files dialog box showing topic files that have been added to the project.

Mapping Files, Topic IDs, and the [ALIAS] and [MAP] Sections

Although Topic files contain the actual "meat" of the Help information, their sustenance will be unavailable to VB programmers and thus to users unless you associate each Topic file with its own unique Topic ID so that VB programmers can set the `HelpContextID` property of objects to point to the topics.

You associate Topic files with unique Topic IDs by creating and including a Topic ID mapping file (*.h) in your project and then associating the Topic IDs with Topic files already included in the [FILES] section. To accomplish this, take the following steps:

STEP BY STEP

7.3 Associating Topic Files With Topic IDs

1. Use a text editor to create one or more mapping files (*.h). Each line of the mapping file will define a constant name for a Topic ID number with the format:

   ```
   #define ConstName TopicIDValue
   ```

   See Listing 7.3 for an example.

LISTING 7.3

SAMPLE TOPIC ID MAPPING FILE

```
#define IDH_GENDER 10
#define IDH_MARITALSTATUS 20
#define IDH_MEDICALINSURANCE 30
```

2. Specify the mapping file or files in the [MAP] section of the *.hhp file. Do this in the HTML Help Workshop by following these steps:

   - Click the HTMLHelp API Information icon (the fourth Toolbar button from the top on the vertical toolbar along the left of the HTML Workshop's main screen) to bring up the HTMLHelp API Information dialog box (see Figure 7.20).
   - On the Map tab of this dialog box, click the Header button to bring up the Include File dialog box. Click the Browse button on this dialog box to browse and select the header file that you created in the first step above (see Figure 7.21).
   - Click the various OK buttons until you've returned to the main project screen that will now have a [MAP] section (refer to Figure 7.14).

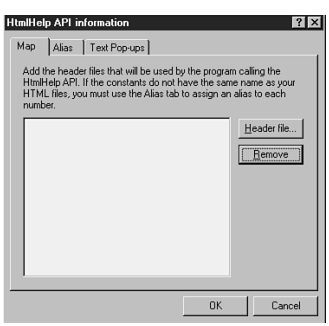
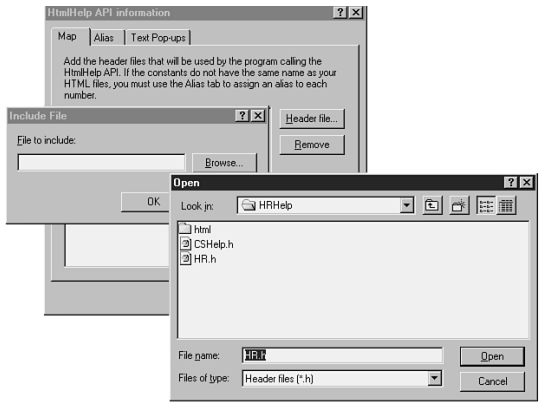


FIGURE 7.20
The HTML Help Workshop's HTMLHelp API Information tabbed dialog box.



◀**FIGURE 7.21**
Including a Mapping Header file for Topic IDs with the Include File dialog box from the Map tab of the HTMLHelp API Information dialog box.

3. In the [ALIAS] section of the HTML Help Project file (*.hhp), associate constant names from the mapping files with corresponding Topic files named in the [FILES] section. Do this in the HTML Help Workshop by following these steps:

- Click the HTMLHelp API Information icon (the fourth Toolbar button from the top on the vertical toolbar along the left of the HTML Workshop's main screen) to bring up the HTMLHelp API Information dialog box (refer to Figure 7.20).
- On the Alias tab of this dialog box, click the Add button to bring up the Alias dialog box (see Figure 7.22).
- In the first field of the Alias dialog box, enter the name of a constant from one of the mapping files that you created in the first step above (refer to Figure 7.22).
- In the second field of the Alias dialog box, use the drop-down list to select the topic file that you wish to associate with this constant's `ContextID`, as shown in Figure 7.22. Add a comment if you wish and click OK.
- Repeat the previous three steps for all the Topic IDs that you want to define. Then click the various OK buttons until you've returned to the main project screen that will now have an [ALIAS] section, as shown in Figure 7.14.

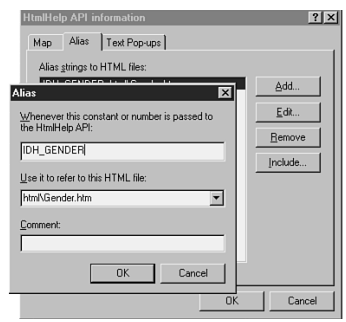


FIGURE 7.22▲
The Alias dialog box on the HTMLHelp API Information tabbed dialog box with completed information for a Topic file mapping.

Figure 7.14 shows the project file with a [`MAP`] section that points to the mapping file, and an [`ALIAS`] section associating Topic ID constant names with Topic files.

If an HTML Help file contained the information shown in Figure 7.14 and the mapping file contained the information shown in Listing 7.3, then the VB programmer using this file could use a `HelpContextID` property value of 1 to associate an object with the information in the `gender.htm` Topic file.

The Index File

The Index file (*.hhk) provides an Index of keywords the user of the compiled Help file can view in order to navigate the HTML Help file's topics (see Figure 7.23).

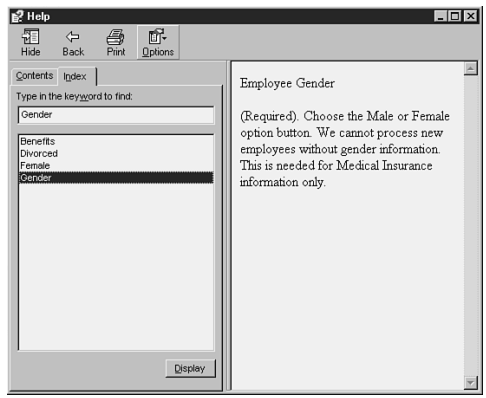


FIGURE 7.23
The Index of an HTML Help file as seen by the end user.

Strictly speaking, you don't need an Index file.

An Index file is really a specialized HTML file that uses a specific HTML format, that of the unnumbered list of OBJECTS (with standard HTML tags such as <UL>…</UL>, <LI>…</LI>, and <OBJECT>…/OBJECT>). Listing 7.4 provides a sample of an Index file.

NOTE

Index Files Are Not Covered in Detail
Because Index files aren't necessary to a context-sensitive help system as specified in the Microsoft VB exam objectives, we do not discuss their implementation in detail here.

LISTING 7.4

A Sample HTML Help Index File

```
<HTML>
<HEAD>
</HEAD><BODY>
<UL>
        <LI> <OBJECT type="text/sitemap">
                <param name="Name" value="Benefits">
                <param name="Name" value="Medical Insurance">
                <param name="Local" value="html\
➥Medicalinsurancestatus.htm">
                </OBJECT>
        <LI> <OBJECT type="text/sitemap">
                <param name="Name" value="Divorced">
                <param name="Name" value="Marital Status">
                <param name="Local" value="html\
➥maritalstatus.htm">
                </OBJECT>
        <LI> <OBJECT type="text/sitemap">
                <param name="Name" value="Female">
                <param name="Name" value="Gender of Employee">
                <param name="Local" value="html\Gender.htm">
                </OBJECT>
        <LI> <OBJECT type="text/sitemap">
                <param name="Name" value="Gender">
                <param name="Name" value="Gender of Employee">
                <param name="Local" value="html\Gender.htm">
                </OBJECT>
</UL>
</BODY></HTML>
```

Each OBJECT of type "text/sitemap" in the unnumbered HTML list represents a topic file already included in the project with one or more keywords pointing to it. For example, in the listing we see that the Help topic furnished by the file `MedicalInsuranceStatus.htm` has two keywords that point to it, "Benefits" and "Medical Insurance."

You can add a reference to an Index file of your project by using the Files tab on the Project Options dialog box (available from the Change Project Options icon on the main screen of the HTML Help Workshop).

You can automatically create and maintain an Index file using the Index tab on the main project screen.

The Contents File

The Contents file (*.hhc) provides a Table of Contents that the user of the compiled Help file can view in order to navigate the HTML Help file's topics (see Figure 7.24).

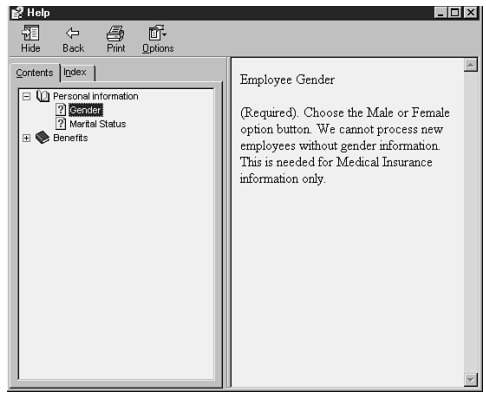


FIGURE 7.24
The Contents of an HTML Help file as seen by the end user.

Listing 7.5 provides a simple example of a Contents file.

LISTING 7.5

A Sample HTML Help Contents File

```
<HTML>
<HEAD>
</HEAD><BODY>
<UL>
        <LI> <OBJECT type="text/sitemap">
                <param name="Name" value="Personal
➥information">
                </OBJECT>
        <UL>
                <LI> <OBJECT type="text/sitemap">
                        <param name="Name" value="Gender">
                        <param name="Local"
➥value="html\Gender.htm">
                        </OBJECT>
                <LI> <OBJECT type="text/sitemap">
                        <param name="Name" value="Marital
➥Status">
                        <param name="Local"
➥value="html\maritalstatus.htm">
                        </OBJECT>
        </UL>
        <LI> <OBJECT type="text/sitemap">
```

```
                <param name="Name" value="Benefits">
                </OBJECT>
        <UL>
                <LI> <OBJECT type="text/sitemap">
                        <param name="Name" value="Medical
➥Insurance">
                        <param name="Local" value="html\
➥Medicalinsurancestatus.htm">
                        </OBJECT>
        </UL>
</UL>
</BODY></HTML>
```

Like an Index file, a Contents file is really a specialized HTML file that also uses the same HTML format as the Index file, that of the unnumbered list of OBJECTS (with standard HTML tags such as <UL>…</UL>, <LI>…</LI>, and <OBJECT>…/OBJECT>).

Also like an Index file, a Contents file is optional.

As the Listing illustrates, the lists in Contents files differ from those of Index files because Contents lists can be nested in various levels of depth. This nesting then implements subsections and sub-subsections in the Table of Contents.

Each OBJECT of type "text/sitemap" in the unnumbered HTML can represent a topic with a single Table of Contents entry pointing to it. For example in the listing, we see that the Help topic furnished by the file `MedicalInsuranceStatus.htm` is referred to by the Table of Contents entry "Medical Insurance."

You can add a reference to a Contents file to your project by using the Files tab on the Project Options dialog (available from the Change Project Options icon on the main screen of the HTML Help Workshop).

You can automatically create and maintain a Contents file using the Contents tab on the main project screen.

NOTE

Contents Files Are Not Covered in Detail Because Contents files aren't necessary to a context-sensitive help system as specified in the Microsoft VB exam objectives, we do not discuss their implementation in detail here.

HTML Help Source Files for `WhatsThisHelp`

If you wish to implement `WhatsThisHelp` from an HTML Help file, you must include the following:

- So-called "context-sensitive" or "Pop-up" Topic files. One of these files can contain many small "one liner" `WhatsThisHelp` topics as well as a unique constant name to identify each topic.

NOTE

Confusion Over Use of the Term "Context-Sensitive" Strictly speaking, all of the VB help that we discuss in this chapter is "context-sensitive" help. However, documentation and terminology for the HTML Help Workshop use this term for only "Pop-Up" topics that correspond to `WhatsThisHelp` topics in VB.

- Topic ID mapping files. This second type of file format is identical to the format of the *.h file described above for mapping `HelpContextID` topics. The purpose of this file is to map each of the unique constant names defined in the Pop-Up Topic file to a numeric Topic ID.

You include these two files in an HTML Help project by specifying them under the [`TEXT POPUPS`] section of the HTML Help project file (*.hhp), as described in the following sections.

Context-Sensitive Topic Files

A context-sensitive Topic file (*.txt) is a text file that associates a unique constant name with some brief text that you can use in a `WhatsThisHelp` system in VB.

The format for each `WhatsThisHelp` topic entry is:

```
.topic ConstName
Text of Topic
```

ConstName is a unique identifier that will later be used to link the topic to a Topic ID (see the following section) and *Text of Topic* is the text that the user will actually see when `WhatsThisHelp` pops up.

Listing 7.6 gives a sample of such a file.

LISTING 7.6

A SAMPLE HTML HELP TOPIC FILE FOR TEXT POPUP HELP (`WhatsThisHelp`)

```
.topic IDH_GENDER
Optional.

.topic IDH_MARITALSTATUS
Required for Tax status.

.topic IDH_MEDICALINSURANCE
Has medical insurance.
```

Context-Sensitive Topic ID Mapping Files

Just as you must map constant names to numeric values for `HelpContextIDs`, so must you map `WhatsThisHelp` topic constants to numeric values for `WhatsThisHelpIDs`. You use a Context-Sensitive

Mapping file (*.h) to accomplish this. The format of each line in such a file is the same as the format of the lines in a mapping file for `HelpContextID` constants:

```
#define ConstName ContextIDValue
```

Listing 7.7 gives an example of the contents of a Context-Sensitive Topic ID mapping file.

LISTING 7.7

A TEXT POPUP (WHATSTHISHELP) TOPIC ID MAPPING FILE

```
#define IDH_GENDER 10
#define IDH_MARITALSTATUS 20
#define IDH_MEDICALINSURANCE 30
```

Including WHATSTHISHELP Files in the HTML Help Project

After you've created a Context-Sensitive Topic and Context-Sensitive Topic ID mapping file as described in the previous two sections, you can include them in the HTML Help project by following these steps:

STEP BY STEP

7.4 Associating Topic Files With Topic IDs

1. Click the HTMLHelp API Information icon (the fourth Toolbar button from the top on the vertical toolbar along the left of the HTML Workshop's main screen) to bring up the HTMLHelp API Information dialog box (refer to Figure 7.20).
2. On the Text Pop-ups tab of this dialog box, click the Header File button to bring up the Include File dialog box. Then browse to and select the mapping file that you created for Pop-Up Topic IDs (Figure 7.25). After the file is selected, click OK to return to the Text Pop-ups tab.

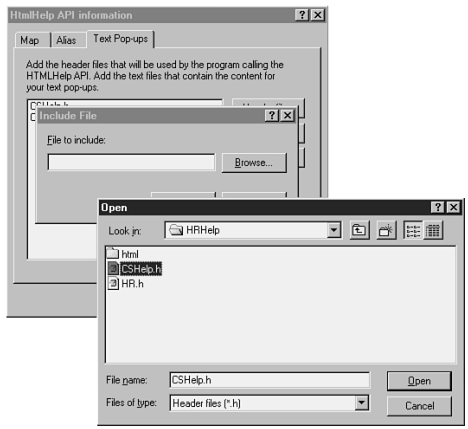


FIGURE 7.25▶
Selecting a Topic ID mapping file for `WhatsThisHelp` with the Include File dialog box from the Text Pop-ups tab of the HTMLHelp API Information dialog box.

3. On the Text Pop-ups tab, click the Text File button to bring up the Include File dialog box. Then browse to and select the Context-Sensitive Topic ID file (*.txt) that you created for Pop-up topics (Figure 7.26).

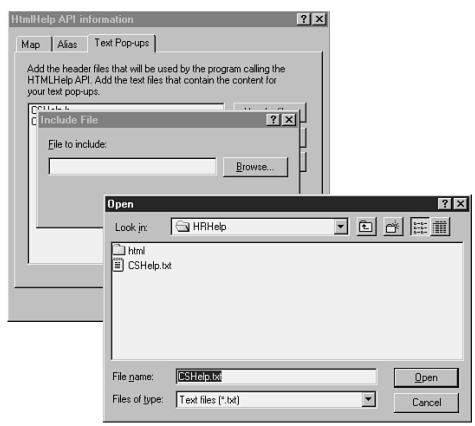


FIGURE 7.26▶
Selecting a Topic file for `WhatsThisHelp` with the Include File dialog box from the Text Pop-ups tab of the HTMLHelp API Information dialog box.

4. Click OK until you've returned to the main screen. You should now see a [`TEXT POPUPS`] section in your project that includes the two files you just selected (refer to Figure 7.14).

CHAPTER SUMMARY

This chapter covered the following key topics for the Microsoft Certification Exam:

- The two different help file formats: HTML Help (the new standard) and `WinHelp` (the older standard).
- VB's `App.HelpFile` property that you can set at design time or runtime to provide a Help file for your application.
- Use of the `HelpContextID` property to provide full-screen, context-sensitive help information from a Help file.
- How to implement `WhatsThisHelp` to provide pop-up help information from a Help file.
- How to use the `ToolTipText` property to provide pop-up help information that does not come from a Help file.
- How to create an HTML Help file.

KEY TERMS

- Context
- Context ID
- HTML Help
- Topic ID
- `WinHelp`

APPLY YOUR KNOWLEDGE

Exercises

7.1 Using Built-In VB Help Properties With an Existing Help File

Estimated Time: 15 minutes

Objective: This exercise guides you through the steps you need to follow to implement both `HelpContextID` Help and `WhatsThisHelp` from an existing HTML Help file or from an existing `WinHelp` file. The single form in the project provides a simple interface such as a human resources department might use to enter information about a new employee. In the steps of this exercise, you will connect objects on the form to topics in both `WinHelp` and HTML Help files. You will use two precompiled Help files that have been provided for this exercise on the CD.

1. Locate the `WinHelp` file, HR.hlp and the HTML Help file, HR.chm, on the CD accompanying this book. Make sure they are available on your system as you work through this exercise.
2. Start a new Standard EXE VB project with a single form. Place controls on the form's surface as shown in Figure 7.27. Adjust the objects' properties as given in Table 7.1.

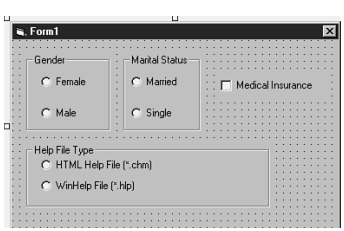


FIGURE 7.27
The form for Exercise 7.1.

TABLE 7.1

OBJECTS TO PUT ON THE FORM FOR EXERCISE 7.1

Control	*Property*	*Value*
Frame	Name	fraGender
	Caption	Gender
OptionButton	Name	optFemale
	Caption	Female
OptionButton	Name	optMale
	Caption	Male
Frame	Name	fraMaritalStatus
	Caption	Marital Status
OptionButton	Name	optSingle
	Caption	Single
OptionButton	Name	optMarried
	Caption	Married
Frame	Name	fraHelpFileType
	Caption	Help File Type
OptionButton	Name	optHTMLHelpFile
	Caption	HTML Help File (*.chm)
OptionButton	Name	optWinHelpFile
	Caption	WinHelp File (*.hlp)
Check Box	Name	chkMedicalInsurance
	Caption	Medical Insurance

3. Use the Project Properties menu dialog box to set the project's `HelpFile` to HR.chm.
4. Make both the `HelpContextID` and `WhatsThisHelpID` properties for the Gender frame 10 (this is the Gender topic number in the Help file's source code).

APPLY YOUR KNOWLEDGE

5. Make both the `HelpContextID` and `WhatsThisHelpID` properties for the Marital Status frame 20 (this is the Marital Status topic number in the Help file's source code).
6. Make both the `HelpContextID` and `WhatsThisHelpID` properties for the Medical Insurance Check Box 30 (this is the Medical Insurance topic number in the Help file's source code).
7. Run the application and test the Help topics in HR.chm by navigating to each of the three areas for which you set the `HelpContextID` above and pressing F1.
8. Stop the application and set the form's `WhatsThisHelp` and `WhatsThisHelpButton` properties to `True`. Re-run the application and note the difference in behavior.
9. Enable the application to switch between different Help files as it runs. You'll do this by putting code in the `Click` events of the Option buttons for the two different Help files:

```
Private Sub optHTMLHelpFile_Click()
  'Path will vary depending on where you
➥place the file
  App.HelpFile = "A:\HR.CHM"
End Sub

Private Sub optWinHelpFile_Click()
  'Path will vary depending on where you
➥place the file
  App.HelpFile = "A:\HR.HLP"
End Sub
```

10. Repeat steps 7 and 8 above, but for each step, switch Help files during the step by clicking the Help file option buttons. Observe the difference in behavior and appearance between help coming from the *.hlp and *.chm Help files.

7.2 Creating HTML Help for `HelpContextIDs`

Estimated Time: 90 minutes (not including time to download and install HTML Help Workshop)

Objective: This exercise shows you how to create an HTML Help file whose topics can be used with VB's `HelpContextID` property. In this exercise and the following exercise, you'll re-create the HR.chm Help file used in Exercise 7.1.

> **NOTE**
>
> **You Need the Microsoft HTML Help Workshop** In order to complete Exercises 7.2 and 7.3, you will need to download and install the Microsoft HTML Help Workshop on your workstation.
>
> As of this writing, you can obtain HTML Help Workshop as a free download from Microsoft's Web site at
>
> `www.microsoft.com/workshop/author/htmlhelp/default.asp`
>
> The text on the Web page refers to the file as "HTML Help 1.1." The name of the file to download is htmlhelp.exe.

1. The three topics that you will implement will be the three Help topics of Exercise 7.1: Gender, Marital Status, and Insurance Status.
2. Create an HTML file named `Gender.htm` for the Gender help topic. You may use a text editor or a Web authoring tool. The contents of `Gender.htm` should look like this:

APPLY YOUR KNOWLEDGE

```
<HTML>
<HEAD>
<title>Gender of Employee</title>
</HEAD>
<BODY>
<p>
(Required). Choose the Male or Female option
➥button.
We cannot process new employees without
➥gender information.
This is needed for Medical Insurance
➥information only.
</p>
</BODY>
</HTML>
```

3. Create an HTML file named `MaritalStatus.htm` for the Marital Status help topic. You may use a text editor or a Web authoring tool. The contents of `MaritalStatus.htm` should look like this:

```
<HTML>
<HEAD>
<title>Marital Status</title>
</HEAD>
<p>
(Required). Needed for deduction for
➥withholding tax computations.
</p>
</BODY>
</HTML>
```

4. Create an HTML file named `InsuranceStatus.htm` for the Insurance Status help topic. You may use a text editor or a Web authoring tool. The contents of `InsuranceStatus.htm` should look like this:

```
<HTML>
<HEAD>
<title>Medical Insurance</title>
</HEAD>
<BODY>
<p>Medical Insurance Status</p>
Check to see whether employee wants.
</BODY>
</HTML>
```

5. Use a text editor to create a header definition file named `HR.h` that will map a unique constant name corresponding to each Topic ID that you want for the three help topics. The contents of `HR.h` should look like this:

```
#define IDH_GENDER 10
#define IDH_MARITALSTATUS 20
#define IDH_MEDICALINSURANCE 30
```

6. Start the HTML Help Workshop application.
7. Choose the File/New menu option and choose "Project" as the item you want to create when you see the prompt (refer to Figure 7.9).
8. You'll enter the New Project Wizard (refer to Figure 7.10). Leave the first Check box (Convert `WinHelp` Project) unchecked and proceed to the next screen. On that screen, you'll be prompted to specify a name and a path for your Help project file. Use the Browse button to choose a path, preferably the same folder where you created the HTML files (or perhaps a parent folder of that HTML folder) in the above steps (refer to Figure 7.11).
9. Proceed to the next screen (refer to Figure 7.12) which prompts you for existing files to include in the project. For this exercise we'll omit Contents and Index files. However, you've created HTML files in the previous steps of this exercise and you'll want to include them. Check the box labeled "HTML files (.htm)" and proceed to the next screen.
10. The next screen of the New Project Wizard prompts you to specify the individual HTML files that you'd like to include. For each HTML file that you want to add, click the Add button and use the resulting file dialog box to choose an HTML file, as shown in Figure 7.28. When you've specified all the files to include in the project, click the Next button.

APPLY YOUR KNOWLEDGE

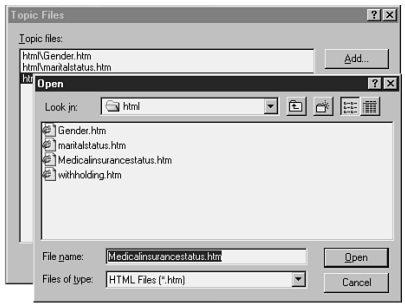


FIGURE 7.28▲
Using the New Project Wizard to add existing HTML Topic files to the HTML Help project.

11. Answer the various prompts to finish the Wizard and save the project file. The Wizard creates a project file whose left-hand windows should show part of the information ([OPTIONS] and [FILES] sections) as illustrated in Figure 7.14. Notice that the HTML files you specified in step 10 appear under the [FILES] section.

12. Now you must add to your project the Mapping file that you created in step 5 above. To do so, click the HTMLHelp API Information Toolbar button. It's the fourth button from the top among the vertical Toolbar buttons along the left side of the HTML Help Workshop's main screen, as indicated in Figure 7.20. The HTMLHelp API Information tabbed dialog box will appear as in Figure 7.20. Making sure that you've selected the first tab (labeled "Map"), click the Header File button, then Browse to and select the HR.h file that you created in step 5 above (refer to Figure 7.21). After you've selected this file and closed the HTMLHelp API Information dialog box, a [MAP] section should appear in your project window, as shown in Figure 7.14.

13. Now you must create Alias information in your project file to tie the Topic IDs you defined in steps 5 and 12 to specific HTML Topic files that you created in steps 2 through 4. To do so, click once again the HTMLHelp API Information button (see Figure 7.20), and this time choose the Alias tab on the resulting dialog box.

14. On the Alias tab, click the Add button for each HTML Topic file that you wish to associate with a Topic ID. Each time you click the button, the Alias dialog box will pop up, as shown in Figure 7.22. In the first field of the dialog, type the name of one of the mapping constants that you included in the file created in step 5. In the second field of the dialog box, either type or use the drop-down list to associate one of the project's HTML Topic files with the constant.

15. When you're done adding Alias information, the list of Alias strings should look like Figure 7.29. After you click OK on the HTMLHelp API Information dialog box, a new [ALIAS] section will appear in the project window, as shown in Figure 7.14.

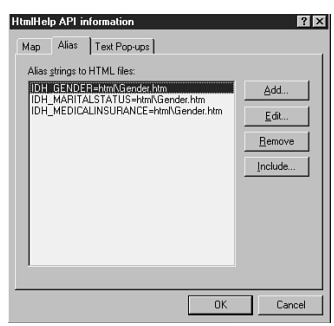


FIGURE 7.29▲
The newly created list of topic file-to-constant mappings on the Alias tab of the HTMLHelp API Information dialog box.

APPLY YOUR KNOWLEDGE

16. You may now compile your Help project by choosing File, Compile from the menu, or by clicking the Compile Toolbar button (the funnel, third from the left) on the upper Toolbar, as indicated in Figure 7.30. Once the project has compiled, you'll see summary information about the compilation in the right-hand pane of the HTML Help Workshop (see Figure 7.30). If prompted to save, answer "Yes."

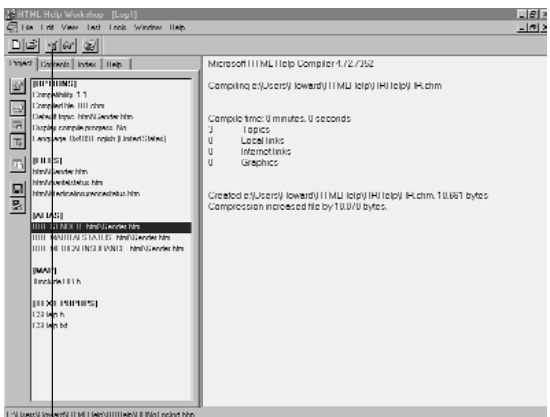


FIGURE 7.30
The HTML Help Workshop screen after compiling an HTML Help project.

17. You can now test your newly created HTML help file (extension `.chm`) by substituting it for the file that you used in the first Exercise. Note that `WhatsThisHelp` will not yet work. `HelpContextID` help will work at only this point.

7.3 Creating HTML Help for `WhatsThisHelp`

Estimated Time: 30 minutes

Objective: This exercise shows you how to modify an HTML Help file so that it can provide topics that can be used to provide `WhatsThisHelp` in a VB application. You will do so by adding to the HTML Help file project that you created in Exercise 7.2.

1. Use a text editor to create a context-sensitive Topic file called `CSHelp.txt` for the three topics that you implemented in the previous project: Gender, Marital Status, and Medical Insurance Status. The contents of the file should look like this:

```
.topic IDH_GENDER
Optional.

.topic IDH_MARITALSTATUS
Required for Tax status.

.topic IDH_MEDICALINSURANCE
Has medical insurance.
```

2. Use a text editor to create a context-sensitive mapping file called `CSHelp.h` to map the topic names from the Topic file with numeric constants. The contents of this file should look like this:

```
#define IDH_GENDER 10
#define IDH_MARITALSTATUS 20
#define IDH_MEDICALINSURANCE 30
```

3. Make sure that you have the HTML Help Workshop running with the same project open that you created in Exercise 7.2. Click the HTMLHelp API Information button, as described in Exercise 7.2 (refer to Figure 7.20). In the resulting dialog box, choose the Text Pop-ups tab.
4. Click the Header File button, browse to and select the `CSHelp.h` file that you created in step 2 above (refer to Figure 7.25).
5. Click the Text File button, browse to and select the `CSHelp.txt` file that you created in step 1 above (refer to Figure 7.26).

APPLY YOUR KNOWLEDGE

6. Once you select these files and click OK on the HTMLHelp API Information dialog box, your project window should acquire a new section entitled [`TEXT POPUPS`], as shown in Figure 7.14.
7. You may now recompile your help file as described in Exercise 7.2.
8. Test your newly compiled version by substituting it for the help file of the VB project from Exercise 7.1. Verify that you can now use `WhatsThisHelp` with the newly compiled help file.

Review Questions

1. You have an online Help file that you would like to distribute with your application. How do you set a reference to this Help file at design time so that the application displays help when the user presses the F1 key?
2. An application you have created will be used by people in several departments. Each department will have a different Help file. The name of the Help file will be stored in the System Registry and read by your application at runtime. After your application has read the filename from the Registry, how do you set a reference in code so that online help becomes available when the user presses the F1 key?
3. Name the current standard format for Microsoft help files and give the extension used by help files created under that standard. What standard does it replace, and what extension do the old standard's files use?
4. You are developing an application with Visual Basic 6. In other applications, such as Microsoft Office, you have seen little explanations pop up when you leave the mouse pointer over a button on the toolbar. You want to add this functionality to your application for your users when they leave the mouse pointer over a button in the toolbar. How can you implement these little pop-up tips in your Visual Basic applications?
5. If you are going to add context-sensitive help to your Visual Basic application, what important information do you need from the person who is creating the Help file?
6. What section in an HTML Help Header file specifies the names and locations of Topic files?

Exam Questions

1. Which property of the `App` object identifies the Help file and the path to that file that will be displayed when the user presses the F1 key?

   A. `Help`

   B. `HelpContents`

   C. `HelpFile`

   D. `HelpTopic`

2. The format of Topic files in an HTML Help project is

   A. Web page (*.htm)

   B. Text format (*.txt)

   C. Help Project format (*.hpp)

   D. Rich Text Format (*.rtf)

3. In what context(s) would the line

   ```
   #define Poetry 1
   ```

   be appropriate? (Pick all that apply.)

APPLY YOUR KNOWLEDGE

A. In the Topic mapping file for HTML Help topic files

B. In the [`MAP`] section of an HTML Project Header file

C. In the [`ALIAS`] section of an HTML Project Header file

D. In the topic mapping file for `WhatsThisHelp` topics

4. If you are using `WhatsThisHelp` on a form in your application, which property of that form must be set to `True`, regardless of the method used to display `WhatsThis` information?

   A. `ShowWhatsThis`

   B. `WhatsThisHelpID`

   C. `WhatsThisMode`

   D. `WhatsThisHelp`

5. Where does Visual Basic get the information that is displayed as ToolTips?

   A. From the Help file specified in the Project Properties window

   B. From the Help file specified by `App.HelpFile`

   C. From the Help file specified by the `HelpFile` property of a `CommonDialog` control

   D. None of these

6. What are the two ways that can be used to connect a Help file to an application so that when the user presses the F1 key, the Help file is displayed?

   A. Setting the `HelpFile` property of a `CommonDialog` control

   B. Setting the `HelpFile` property of the `App` object

   C. Identifying the Help file on the Project Properties dialog box

   D. Using the `OpenFile` statement in the `KeyPress` event of an MDI form

7. You have a window, `Form1`, with two controls, a frame, `Frame1`, and a text box, `Text1`. `Text1` is drawn on `Frame1`. If the cursor is in `Text1` when the user presses the F1 key, where will Visual Basic look first for a `HelpContextID`?

   A. `Text1.HelpContextID`

   B. `Frame1.HelpContextID`

   C. `Form1.HelpContextID`

   D. None of the these

8. The purpose of the [`ALIAS`] section in an HTML Help project file is

   A. To map Topic ID names to Topic ID constant numbers.

   B. To map Pop-Up Topic ID names to Pop-Up Topic ID constant numbers.

   C. To map Topic ID names to Topic filenames.

   D. To map Topic filenames to Topic ID constant names.

9. If you are using `WhatsThisHelp` in your application, how does Visual Basic determine the help topic that will be displayed for an object?

   A. Code must be placed in the `Click` event of the object the user selects when in `WhatsThis` mode.

   B. Visual Basic uses the `HelpContextID` of the selected object.

   C. Visual Basic uses the `WhatsThisHelpID` of the selected object.

   D. None of these.

APPLY YOUR KNOWLEDGE

Answers to Review Questions

1. A reference to a Help file for an application can be set through the Project Properties window. From the Project menu, choose Properties. Either type the Help file name in the Help File Name field, or browse for and select the file. See "Referencing Help Through the `HelpFile` Property of an Application."

2. If the name and/or location of a Help file is not known at design time, a reference to the file can be set at runtime by using the `HelpFile` property of the `App` object. After the filename has been set, pressing F1 in the application will display the online help. See "Referencing Help Through the `HelpFile` Property of an Application."

3. The current standard format for Microsoft help files is HTML Help, and the extension for an HTML Help file is `chm`. The HTML Help format replaces the `WinHelp` format as the standard, and the extension for `WinHelp` files is `hlp`. See "Two Types of Help Files."

4. Pop-up tips for controls in Visual Basic applications can be implemented by just putting the desired text in the `ToolTipText` property of controls. For ToolTips to work on `ToolBars` and `TabStrips`, you must also set the `ShowTips` property of these controls to `True`. See "Adding ToolTips to an Application."

5. To add context-sensitive help to an application, you need to know the mapping between Topic IDs that will be used in the project and topics in the Help file. The mapping is usually created by the person who writes the Help file. Cooperation between the help author and developer is important to ensure that the correct Topic IDs are associated with the proper objects in the application.

6. The [`FILES`] section of an HTML Help project file contains the list of Topic files. See "Topic Files and the [`FILES`] Section."

Answers to Exam Questions

1. **C.** The `App.HelpFile` contains the name, and optionally, the path to a Help file associated with the application. The `HelpFile` property can be set at design time through the Project Properties window, or at runtime by setting the `App.HelpFile` property. For more information, see the section titled "Referencing Help Through the `HelpFile` Property of an Application."

2. **A.** The format for Topic files in an HTML Help project is Web page (*.htm). For more information, see the section titled "Topic Files and the [`FILES`] Section."

3. **A, D.** The format `#define TopicIDName TopicIDValue` is appropriate to Context ID Mapping files for HTML Help Topic files as well as to Context ID Mapping files for Pop-Up (`WhatsThisHelp`) topics. The [`MAP`] section of an HTML Help file contains a list of Topic file names and locations, while the [`ALIAS`] section contains a mapping between Topic file names and Topic ID constant names. For more information, see the sections titled "Mapping Files, Topic IDs, and the [`ALIAS`] and [`MAP`] Sections."

4. **D.** `WhatsThisHelp` must be set to `True` whether you are using `ShowWhatsThis`, `WhatsThisMode`, or a `WhatsThisButton` on your form. For more information, see the section titled "Providing `WhatsThisHelp` in an Application."

APPLY YOUR KNOWLEDGE

5. **D.** ToolTips don't come from a Help file. They come from the `ToolTipText` property of the control for which the tip is intended. For more information, see the section titled "Adding ToolTips to an Application."

6. **B, C.** Help will automatically be invoked with the F1 key when the `App.HelpFile` property is set and when the Help file is identified on the Project Properties dialog box. For more information, see the section titled "Referencing Help Through the `HelpFile` Property of an Application."

7. **A.** Visual Basic will check the active control for a `HelpContextID` first. If one is not found there, the container of the active control will be checked next. For more information, see the section titled "Context-Sensitive Help with the `HelpContextID` Property."

8. **C.** The purpose of the [`ALIAS`] section in an HTML Help project file is to map Topic ID names to Topic file names. For more information, see the section titled "Mapping Files, Topic IDs, and the [`ALIAS`] and [`MAP`] Sections."

9. **C.** The `WhatsThisHelpID` identifies the help topic that will be used. For more information, see the section titled "Providing `WhatsThisHelp` in an Application."

OBJECTIVES

This chapter helps you prepare for the exam by covering the following objectives:

Use data binding to display and manipulate data from a data source (70-175 and 70-176).

- *Data binding* is the act of connecting a control in the programming environment directly to a field in a row in a `Recordset` so that users can directly view and edit the data by changing the control contents.

Access and manipulate a data source by using ADO and the ADO Data Control (70-175 and 70-176).

- Microsoft's VB exam objectives for data access concentrate on programming with ADO (ActiveX Data Objects, defined at greater length throughout this chapter), including more automated ways to program with ADO (the ADO Data Control and the Data Environment Designer).

Use the ADO `Errors` collection to handle database errors (70-175).

- The ADO `Errors` collection is, as its name implies, a special structure furnished by ADO to give you information about the most recent error raised by ADO in your application.

CHAPTER 8

Creating Data Services: Part I

OUTLINE

Overview of OLE DB and ADO **305**

ADO and the ADO Object Model 306

Programming With Automated Data-Binding Tools **308**

Managing ADO Objects With the Data Environment Designer 308

Accessing Data With ADO and the ADO Data Control **317**

Using the ADO Data Control 318

Using the ADO `Errors` Collection **357**

Chapter Summary **358**

STUDY STRATEGIES

- Get familiar with connecting to data through ADO using different providers, such as Access, ODBC, and SQL Server providers. Familiarity with providers is not an official exam objective, but you can't work with anything else in ADO without being able to connect to at least one provider. See the discussions about how to open connections in the sections titled "Adding Connection and Command Objects with the Data Environment Designer" and "Setting Up the ADO Data Control."
- As a basis for the rest of your studies on ADO data-access topics, you need to get to know the ADO object model. Make sure that you review the general descriptions of the ADO objects in the section titled "ADO and the ADO Object Model," the sections under "Programming with ADO," and the code in all the exercises, but most especially in Exercise 8.1.
- Get familiar with the Data Environment Designer through practical experience, as given in Exercise 8.2 and as discussed in the section titled "Managing ADO Objects with the Data Environment Designer." Make sure that you are familiar with the relationships among the Data Environment Designer, `Connection` objects, `Command` objects, and the `Recordsets` that are automatically created through the Data Environment Designer's `Command` objects. You should also know how to automatically place controls such as `TextBoxes` and the `DataGrid` on a form by dragging a `Command` object from a Data Environment Designer. Know how to programmatically manipulate the recordset provided through a `Command` object on a Data Environment Designer.
- Get familiar with the ADO Data Control through experience and examples, as provided in Exercise 8.3 and discussed in the sections under "Using the ADO Data Control."

INTRODUCTION

Microsoft's VB exam objectives for data access concentrate on programming with ADO (ActiveX Data Objects), the newest standard for data access from Microsoft programming environments.

This chapter and the following chapter discuss the basics of the ADO object model, how to bind controls to ADO data, and how to choose and manipulate ADO data connections.

OVERVIEW OF OLE DB AND ADO

This chapter and the following chapter focus on *ADO*, which stands for *ActiveX Data Objects*. ADO is a general object model that is Microsoft's latest programming interface for access to data provided by *OLE DB*.

OLE DB is, in turn, an open standard for providing data access. Various OLE DB *data providers* (or just *providers*) now exist. OLE DB providers include those for Microsoft Jet databases, SQL Server, and ODBC.

A programmer using ADO can connect to a data source through one of the existing OLE DB providers. The programmer can then manipulate this data by using the ADO object model.

OLE DB in turn is an implementation of Microsoft's Universal Data Access model (UDA). The UDA is a general COM-based standard for access to any type of data source, no matter how exotic.

Besides typical data sources such as databases and spreadsheets, UDA aims to include anything that can be considered as a "data source" in the broad sense of the term. Examples of some of the more unexpected types of data sources might include file directory structures, text files, or COM ports.

Veteran (and even less-than-veteran) VB programmers will recall with varying degrees of fondness and other emotions such data access models as Jet, DAO (Data Access Objects), and RDO (Remote Data Objects). Although these data-access models are still available in VB6, Microsoft wants programmers to do all new data development with the ADO model.

The certification exam's data-access questions will focus on ADO. This chapter, therefore, covers various topics related to ADO that you will find on the VB certification exams.

ADO and the ADO Object Model

As discussed earlier, ADO is a data-access object model that provides a programming interface for the OLE DB standard, which in turn is an implementation of Microsoft's Universal Data Access model.

For those who have experience with previous data-access object models in VB, the main points of comparison between ADO and the earlier data access models are as follows:

- ADO's object hierarchy is flatter than the object hierarchies of previous models (DAO and RDO). ADO provides fewer object classes than earlier models, and most objects can be instantiated directly by the programmer instead of having to be instantiated through other objects.

  The DAO `Recordset` object, for example, had to be instantiated through a `Database` object. The ADO `Recordset` object may be instantiated independently of any other object, and then attached to a `Connection` or `Command` object—or you can open the `Recordset` by calling methods of the `Connection` and `Command` objects.

  Making objects less dependent on each other has interesting and useful consequences besides just conceptual clarity: As a consequence of the example just given, you can create an ADO `Recordset` object that never has to be connected to an existing data source—and you can therefore use such a *disconnected* `Recordset` to track and maintain virtual data created entirely within your application or to maintain data from a database offline.

- ADO provides the programmer more opportunities to fine-tune the *data cursor* of a `Recordset`. A data cursor (or just a "cursor") represents a set of resources initiated by a data connection that provides a connection to a specific row in a set

of data. The data cursor can change position to point to a different row of data. ADO enables you to directly specify where the cursor is implemented (client or server side) and several different types of cursors.

- ADO provides better overall performance than earlier object models.
- ADO is more resource-efficient than earlier object models.
- ADO provides more universal data access, due to the universal nature of its underlying standard, OLE DB.

Following are the objects exposed in the ADO object model:

- **`Connection` object**. Specifies information about the physical connection with a data source.
- **`Command` object**. Stores information about actions performed on the data, such as data modification and retrieval. You can use a `Command` object to execute actions on the data or to return data from the server in a `Recordset` object.
- **`Recordset` object**. Provides a rich selection of properties, events, and methods to expose data in a field-row format, and thus allows you to programmatically traverse, examine, and manipulate specific fields in specific rows of data.
- **`Parameters` collection of the `Command` object (made up of `Parameter` objects)**. Contains information about parameter values that are passed by a `Command` object.
- **`Fields` collection of the `Recordset` object (made up of `Field` objects)**. Contains information about field structure and content of the data in a `Recordset` object.
- **`Properties` collection (made up of `Property` objects)**. Contains information about provider-specific properties of `Command` and `Parameter` objects.
- **`Errors` collection of the `Connection` object (made up of `Error` objects)**. Contains information about the most recent error that occurred when attempting an ADO operation.

NOTE

DAO Performs Better Than ADO When Accessing Jet Databases
When accessing Jet databases (that is, MS Access), DAO will perform better than ADO, because DAO was optimized for Jet.

The rest of this chapter discusses how to use ADO at various levels, beginning at a very automated level and ending with lower-level manipulation in code of the object model and of SQL data providers.

PROGRAMMING WITH AUTOMATED DATA-BINDING TOOLS

- Use data binding to display and manipulate data from a data source.

Microsoft provides a number of tools to make ADO programming accessible to people with different levels of programming experience. The following sections explore two of these tools, the Data Environment Designer and the ADO Data Control.

Managing ADO Objects With the Data Environment Designer

A Data Environment Designer is a visual interface for managing a VB application's ADO `Connection` and `Command` objects.

Take the following steps to add a Data Environment Designer to your project:

STEP BY STEP

8.1 Adding a Data Environment Designer to a Project

1. Choose the Project menu in VB. Make sure that Add Data Environment is one of the Project menu's options. If Add Data Environment does not appear on the Project menu, add it by performing the following steps:
2. Choose Project, Components and select the Designers tab on the Components dialog box (see Figure 8.1).

3. In the Components dialog box, check the Data Environment box, as shown in Figure 8.1.
4. Click the OK button on the Components dialog box.
5. Choose Project, Add Data Environment from the VB menu.
6. Navigate to the Properties window of the Data Environment Designer (see Figure 8.2). Change the `Name` property of the `Data Environment` object as desired (`Name` is recommended to begin with "de").

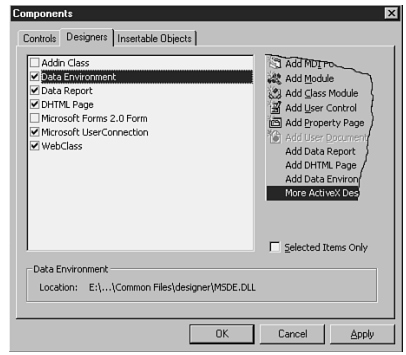
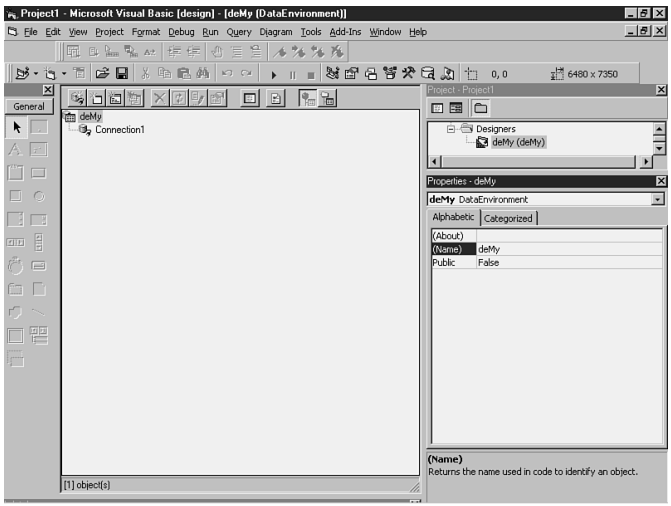


FIGURE 8.1▲
Adding the Data Environment Designer component to your project.



◀FIGURE 8.2
Changing the `Name` property of a Data Environment.

You are now ready to add ADO objects to the Data Environment.

Adding `Connection` and `Command` Objects With the Data Environment Designer

You can add `Connection` and `Command` objects to a Data Environment Designer. You can then manipulate these objects from VB's Visual Design Environment, or in VB code.

STEP BY STEP

8.2 Adding `Connection` and `Command` Objects

1. Make sure that your VB project contains a Data Environment Designer, as discussed in the preceding section.
2. Right-click on the Data Environment Designer's surface, and choose Add Connection, as shown in Figure 8.3.

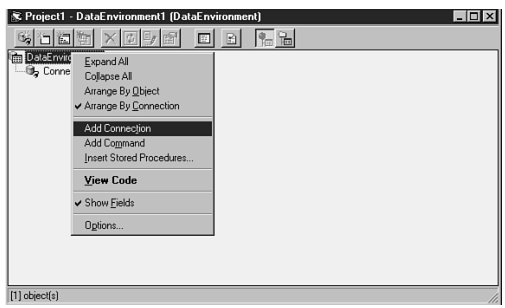


FIGURE 8.3▶
Adding a `Connection` object to a Data Environment Designer.

3. Right-click on the `Connection` object, and then choose Properties from the drop-down menu.
4. On the Provider tab (see Figure 8.4), choose an OLE DB Data Provider. (Some choices are Microsoft Jet 3.51 OLE DB (for MS Access databases), Microsoft OLE DB Provider for ODBC Drivers, or Microsoft OLE DB Provider for SQL Server.)
5. On the Connection tab (see Figure 8.5), set up the specific data connection with the following steps. (Note that the contents of the tab will differ depending on the type of data provider selected on the Connection tab.)

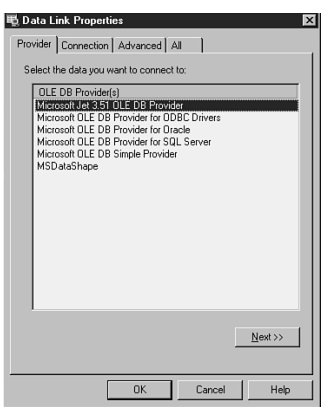


FIGURE 8.4▲
Choosing the provider for a `Connection` object in the Data Environment Designer.

6. Set up the source of the data, whose nature will vary depending on the type of connection. For a Jet data source (Microsoft Access), you will specify the MDB file's name and path. For an ODBC data source, you can specify a data source name (based on an existing DSN) or a connection string that creates a new DSN.
7. Fill in logon information about the username and password.
8. Click OK to accept the ODBC Data Source options you have built.

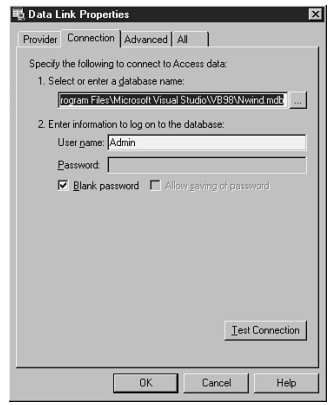


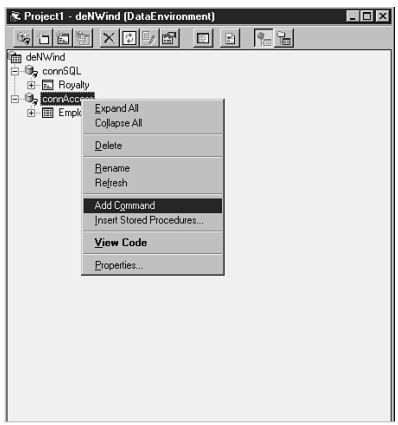
FIGURE 8.5▲
Setting up connection information for a `Connection` object in the Data Environment Designer.

After you have data connections established, you can add `Command` objects to the data connections.

STEP BY STEP

8.3 Adding `Command` Objects to the Data Connections

1. Right-click a `Connection` object and choose Add Command from the drop-down menu, as in Figure 8.6.



◀FIGURE 8.6
Adding a `Command` object to a connection in the Data Environment Designer.

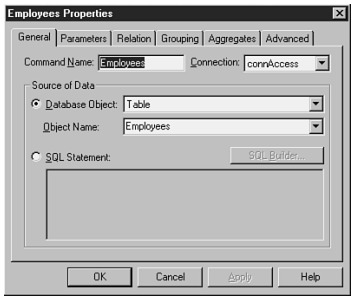


FIGURE 8.7▲
Setting general information about a `Command` object in the Data Environment Designer.

2. On the General tab, give a name to the new `Command` object, as in Figure 8.7.
3. Change the connection information if you want to, or leave it the same to accept the default Connection information from the `Connection` object.
4. Choose the Source of Data Database Object = Table, Stored Procedure, View, or Synonym) (see Figure 8.8).

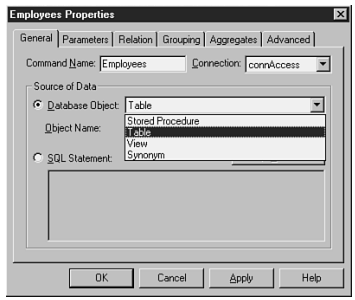


FIGURE 8.8▶
Setting information about the data source for a `Command` object in a Data Environment Designer.

5. On the Advanced tab (Figure 8.9), you can set cursor type, cursor location, locking strategy, and cache size. See later sections in this chapter on each of these subjects.

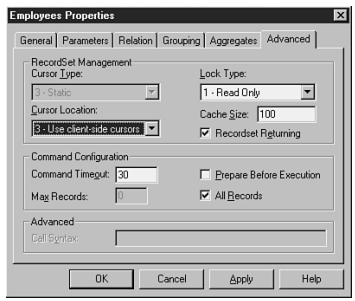


FIGURE 8.9▶
Advanced settings for a `Command` object in a Data Environment Designer.

6. Rename the object if you want.
7. Click OK. Your `Command` object is finished.

NOTE

Cursor Options on the `Advanced` Tab
You can fine tune the `Recordset`'s behavior by setting the `CursorLocation`, `CursorType`, and `Locktype` properties with the correspondingly labeled fields on the Advanced tab. These properties and their meaning are discussed at length in the next chapter.

Binding VB Objects to Data Environment Objects

You can automatically create data-bound controls with the following steps:

STEP BY STEP

8.4 Automatically Creating Data-Bound Controls

1. Select or create a form to hold data-bound controls.

2a. Use the left (primary) mouse button to drag a `Command` object from Data Environment Designer onto the form, where it will automatically create bound controls for you, as shown in Figure 8.10.
or

2b. Use the right (alternate) mouse button to drag a `Command` object onto the form. Upon releasing, you can choose the type of object to drop (`DataGrid`, for example), as shown in Figure 8.11.

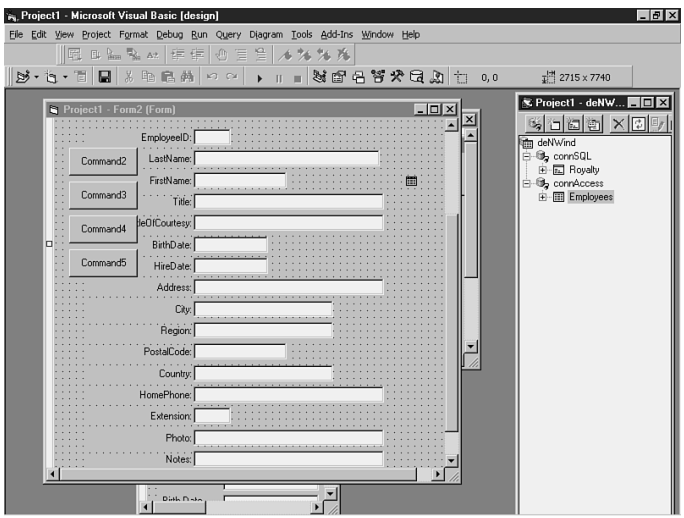


FIGURE 8.10
Dropping a `Command` object from a Data Environment Designer onto a form to create bound controls for data fields (figure shows both the action of dragging and the state of the form after dropping).

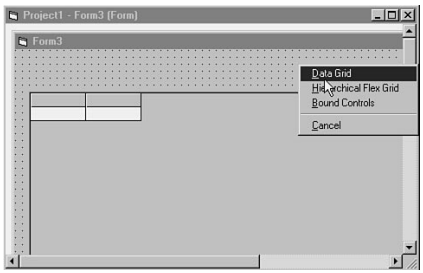


FIGURE 8.11▲
Dropping a `Command` object from a Data Environment Designer onto a form to create a control bound to an entire `Recordset`. This figure shows both the dialog box at the instant you drop the object and the result of the drop.

3. Test the bound controls by running the project and observing the contents of the controls. (They should display the contents of fields in the underlying data.)

You can use the Properties window to examine the properties of a VB data-bound object created with one of the previously discussed methods. Notice that its `DataSource` property points to the `Data Environment` and its `DataMember` property points to the `Command` object dragged onto the form (see Figure 8.12).

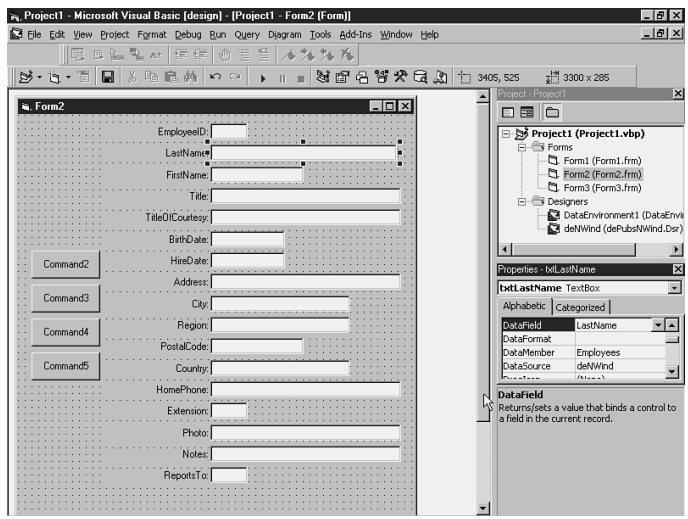


FIGURE 8.12▶
Data property settings of a control bound to a Data Environment's `Command` object.

Consider the case of a control that is bound to an individual data field (this could be a `TextBox` control that you might create in step 2a). Such a bound control's `DataField` property will contain the name of the individual field from the `Recordset` returned by the `Command` object (refer again to Figure 8.12).

Programming With a Data Environment Designer

Although you can bind controls to `Command` objects of the Data Environment Designer in the VB design environment, you will sooner or later need more control over this data than that afforded by automated design-time binding of controls.

To manipulate the data exposed through a `Data Environment`'s `Connections` and `Commands`, you typically program the `Recordset` object returned by a `Command` object.

When you add a `Data Environment` to your VB project, the `Data Environment` becomes available to the project's code as a variable, taking on the name you gave it when you created it (or the default name if you did not change the name). This is similar to what happens when you add a form to your project. (The form becomes available in code under the name you gave it or allowed it to take by default.)

Just as the objects contained in a form or in containers contained by the form are available in code by referring to them with dotted syntax (*`FormName.Object`* or *`FormName.ContainerObject.Object`*), so objects contained in the `Data Environment` become available in code in the following formats:

```
DataEnvironmentName.ConnectionName
DataEnvironmentName.CommandName
```

Even though a `Command` object is logically subordinate to a `Connection` object in the Data Environment Designer's visual hierarchy and, in fact, depends on a `Connection` object for its existence, the `Command` object must be referred to as directly belonging to the `Data Environment`.

This is in keeping with the ADO object model's "flat" object hierarchy. This also implies that no two `Command` objects can have the same name in a `Data Environment`, even if they are under different `Connection` objects.

When your project runs and the physical connection is established, the `Command` objects return `Recordset` objects. The name of each `Recordset` object is available in code in the following format:

```
DataEnvironmentName.rsCommandName
```

In other words, the environment automatically creates a `Recordset` object as a property of `DataEnvironment`. The environment automatically assigns the `Recordset` a name based on the name of the `Command` object that supports it: Its name will be "rs" plus the `CommandObject` name.

If you had created a `Command` object named `Employees` under a `Data Environment` named `deNWind`, for example, you could refer to the `Recordset` in your code as this:

```
deNWind.rsEmployees
```

You can manipulate a `Recordset` programmatically with methods and properties, as discussed in the section titled "Programming with ADO" and more particularly under the section titled "Programming with the `Recordset`."

If you bind `TextBox` controls to a `Data Environment`'s `Recordset` as discussed in the preceding section, for example, you will usually want to give the user the ability to navigate the rows of the `Recordset` as displayed in the controls. One way to accomplish user navigation would be to add four `CommandButtons` whose `Click` event procedures are called, respectively, the `MoveFirst`, `MoveLast`, `MoveNext`, and `MovePrevious` methods of the `Recordset` object, as illustrated in Listing 8.1.

The code in the listing assumes that there is a Data Environment Designer named `deNWind` and a `Command` object named `Employees`. The environment then creates a `Recordset` named `rsEmployees` (based on the `Command` object's name).

The code in the listing manipulates this `Recordset` object through the `Data Environment`.

LISTING 8.1

Implementing User Navigation on the Recordset Object Furnished by a Data Environment's Command Object

```
Private Sub cmdMoveFirst_Click()
    deNWind.rsEmployees.MoveFirst
End Sub

Private Sub cmdMoveLast_Click()
    deNWind.rsEmployees.MoveLast
End Sub

Private Sub cmdMoveNext_Click()
    deNWind.rsEmployees.MoveNext
    If deNWind.rsEmployees.EOF Then
        deNWind.rsEmployees.MoveLast
    End If
End Sub
```

```
Private Sub cmdMovePrevious_Click()
    deNWind.rsEmployees.MovePrevious
    If deNWind.rsEmployees.BOF Then
        deNWind.rsEmployees.MoveFirst
    End If
End Sub
```

Accessing ADO Events for Objects Under a `Data Environment`

To program event procedures for `Data Environment` objects, you can do the following:

- Double-click a `Connection` object in the Data Environment Designer to see a code window for the `Connection`'s event procedures.
- Double-click a `Command` object in the Data Environment Designer to see a code window for its corresponding `Recordset`'s event procedures. (`Command` objects do not themselves support events.)

Because ADO events are identical under the `Data Environment` and in straight ADO programming, the programming of events is not discussed here. Instead, you should refer to the sections that discuss ADO event programming later in this chapter.

Because `Recordset` object programming is discussed at greater length throughout this chapter, `Recordset` object manipulation with the `Data Environment` is also not discussed here.

NOTE

`Recordset` Objects Discussed Throughout This Chapter For more information on programming with `Recordset` objects, see the section on programming the ADO Data Control and on straight ADO object programming later in this chapter.

Accessing Data With ADO and the ADO Data Control

- Access and manipulate a data source by using ADO and the ADO Data Control.

So far in this chapter, you have seen how to use automated, dialog-driven VB design-time facilities to set up a data environment that exposes a `Recordset` to the programmer through `Connection` and `Command` objects.

There are two other major ways to program with ADO:

- The ADO Data Control
- Directly programming the ADO object model

The following sections discuss these major techniques.

Using the ADO Data Control

Like a `Data Environment`, the ADO Data Control also simplifies, automates, or even eliminates some data programming tasks. It has the following similarities to the Data Environment Designer:

- Both the `Data Environment` and the ADO Control expose a `Recordset` to the programmer
- Both the `Data Environment` and the ADO Control are used to bind VB controls (such as `DataGrid` or `TextBox` controls) to a `Recordset`.
- Both the `Data Environment` and the ADO Control enable you to determine the `Recordset`'s cursor type, cursor location, locking strategy, and cache size.
- When necessary, the programmer can bypass the automated user interface and directly manipulate the `Recordset` in code. `Recordset` manipulation is the same in code for both the ADO Data Control and the Data Environment, with but a single syntactic difference: You refer to the ADO Data Control's `Recordset` with the following syntax:

  ```
  ADDataControlName.Recordset
  ```

  You refer to the `Data Environment`'s `Recordset` with this syntax:

  ```
  DataEnvironmentName.rsCommandName
  ```

The ADO Data Control differs in the following ways from a `Data Environment`, however:

- The ADO Data Control supports only one `Recordset` at a time.
- The ADO Data Control does not directly expose `Command` or `Connection` objects.
- The ADO Data Control is visible at runtime and furnishes a visual navigation interface to the user.

The following sections discuss how to set up an ADO Data Control, how to manipulate it programmatically, and how to bind controls to its `Recordset`.

Setting Up the ADO Data Control

To create an ADO Data Control that exposes a `Recordset` in your application, at the minimum you need to do the following:

- Specify a `Connection` by filling in the `ConnectionString` property.
- Specify how to derive a `Recordset` by setting the `RecordSource` property (which is a complex property requiring its own dialog box to set up).

The detailed steps are as follows:

STEP BY STEP

8.5 Creating an ADO Data Control

1. Add the Microsoft ADO DataControl 6.0 (OLEDB) from the Project, Components menu dialog box, as in Figure 8.13. The ADO Data Control icon should now appear in the VB toolbox.
2. Place an instance of the ADO Data Control on the form (see Figure 8.14).

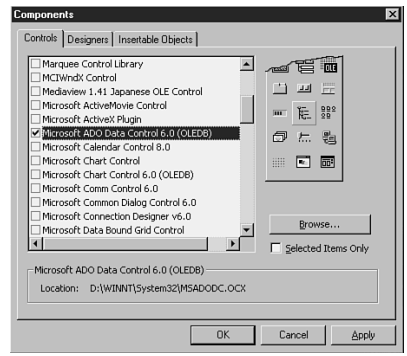


FIGURE 8.13
Adding the Microsoft ADO Data Control to your project's components.

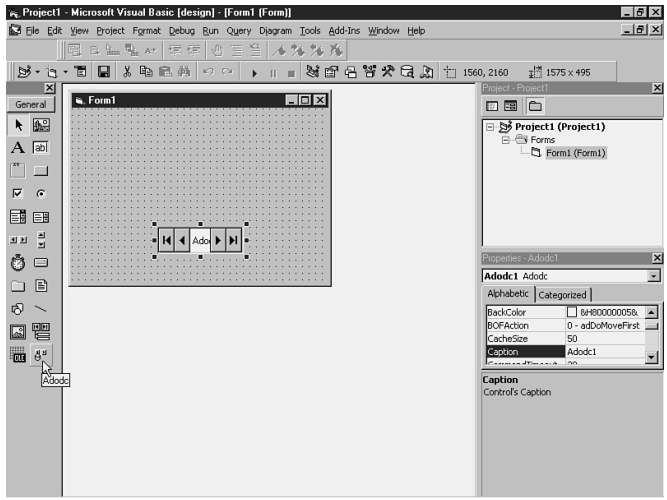


FIGURE 8.14▶
Placing an instance of the ADO Data Control on a form.

3. Change the control's `Name` and `Caption` from their default values. (The `Caption` is for information only, so you can set it to whatever you think will be most informative for the user.)

4. Set the `ConnectionString` property using steps 5–9.

5. Click the ellipsis next to the `ConnectionString` property in the ADO Data Control's Properties window to bring up the Property Page dialog box for this property, as shown in Figure 8.15.

6. As Source of Connection, choose one of the following three options:

   - **Use Data Link File**. If you choose this option, you will be able to click the Browse button to specify an existing *.UDL file).
   - **Use ODBC Data Source Name**. If you choose this option, you will be able to choose an existing ODBC DSN from the drop-down list, or you can create a new DSN by clicking the New button.

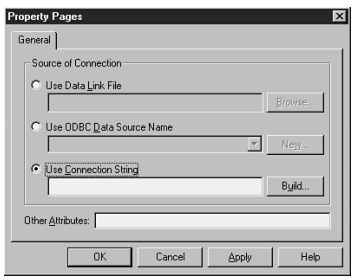


FIGURE 8.15▲
The first and only Property Page dialog box for the ADO Data Control's `ConnectionString` property.

- **Use Connection String**. If you choose this option, you will be able to click the Build button to bring up the Data Link Properties tabbed dialog box.

The following steps assume that you have chosen this option.

7. On the Provider tab of the Data Link Properties tabbed dialog box, choose an OLE DB data provider, such as Microsoft Jet 3.51 OLE DB (see Figure 8.16).

8. The `Connection` tab of the Data Link Properties tabbed dialog box will vary in appearance, depending on the provider specified in the preceding step. In the case of the Microsoft Jet 3.51 OLE DB, you are prompted to choose an Access data file and set some security options (see Figure 8.17).

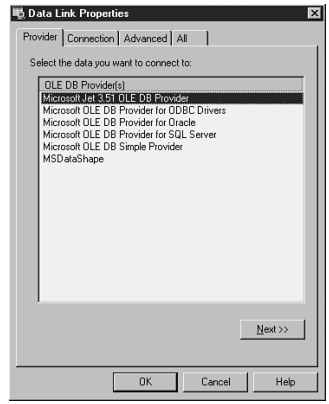
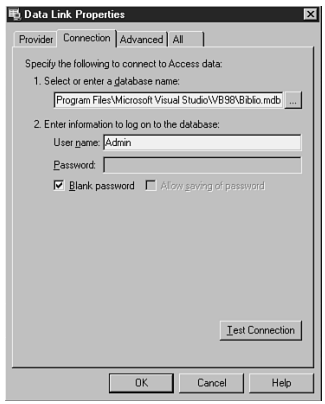


FIGURE 8.16▲
Choosing a provider for the ADO Data Control's `ConnectionString` property.



◀FIGURE 8.17
Setting up connection information for the Jet provider.

9. Click OK to accept the `ConnectionString` options you have built.

10. Still in the ADO Data Control's Properties window, navigate to the `RecordSource` property and click the ellipsis button.

11. On the `RecordSource` tab (see Figure 8.18) of the resulting Property Page dialog box, choose the `CommandType` (`adCmdUnknown`, `adCmdText`, `adCmdTable`, `adCmdStoredProc`).

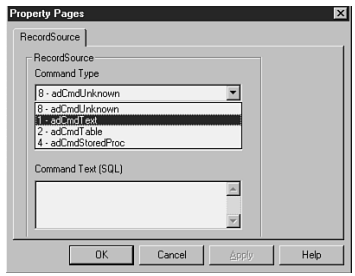


FIGURE 8.18▶
The Property Page dialog box for an ADO Data Control's `RecordSource` property.

12. Complete the dialog box appropriately for the CommandType that you chose:
    - If you chose `adCmdText`, fill in the text of a valid `Select` statement in the Command Text field (see Figure 8.19).
    - If you chose `adCmdTable` or `adCmdStoredProc`, fill in the appropriate table or stored procedure name in the Table or Stored Procedure Name drop-down list (see Figure 8.20).

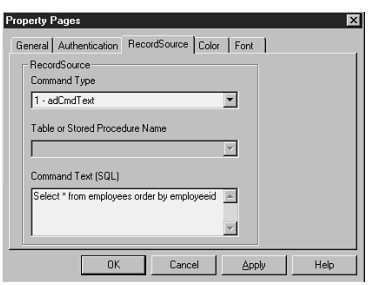


FIGURE 8.19▲
A valid `Select` statement as command text for the `RecordSource` property.

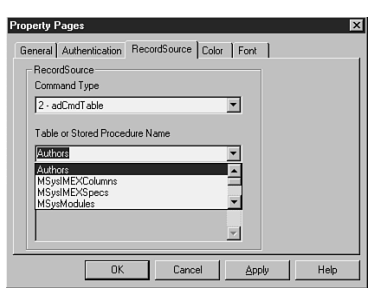


FIGURE 8.20▶
A table or stored procedure name for the `RecordSource` property.

13. Click OK to end the RecordSource dialog box.

> **NOTE**
> **Additional ADO Data Control Properties That Affect the `Recordset`**
> You will eventually need to fine-tune the `Recordset`'s behavior by setting the `CursorLocation`, `CursorType`, and `Locktype` properties. These properties correspond to the standalone `Recordset` object's properties of the same name. The next chapter discusses these properties and their meaning in more detail.

After you have set the ADO Data Control up to expose a `Recordset`, you can bind VB controls to the ADO Data Control, as discussed in the following sections.

The ADO Data Control's `EOFAction` and `BOFAction` Properties

As you will see shortly, the user can manipulate the ADO Data Control to navigate the `Recordset`. When the user attempts to move forward through the `Recordset` on to the end-of-file buffer,

Visual Basic must take some action to ensure that there won't be a problem the next time the user tries to move forward.

The `EOFAction` property tells Visual Basic what to do when the user has moved the ADO Data Control's record pointer on to the end-of-file buffer. The three values of `EOFAction` are as follows:

- **`0 AdDoMoveLast` (default)** The record pointer repositions itself to the last true record in the `Recordset`, and thus avoids any future problems.
- **`1 adStayEOF`** The record pointer stays on the end-of-file buffer. If you choose this option, you must programmatically provide for the record pointer pressing End-of-file.
- **`2 adDoAddNew`** The `AddNew` method of the `Recordset` will execute, adding a new record to the `Recordset` and enabling the user to edit its blank fields.

`BOFAction` has two possible values:

- **`0 adDoMoveFirst` (default)** The record pointer repositions itself to the first true record in the `Recordset`, and thus avoids any future problems.
- **`1 adStayBOF`** The record pointer stays on the beginning-of-file buffer. If you choose this option, you must programmatically provide for the record pointer by pressing Beginning-of-file.

NOTE

Setting the `EOFAction` Property Set the Data Control's `EOFAction` property to `adStayEOF` when you want to directly program the `Recordset`'s behavior. You will then need to put code in the ADO Data Control's `MoveComplete` or `EndOfRecordset` event procedures to handle this possibility.

NOTE

The `BOFAction` Property Set the Data control's `BOFAction` property to `adStayBOF` when you want to directly program the `Recordset`'s behavior. You will then need to put code in the ADO Data Control's `MoveComplete` event procedure to handle this possibility.

Binding VB Controls to the ADO Data Control's `Recordset`

When the form containing the ADO Data Control loads into memory, the ADO Data Control will connect to data using the `ConnectString` property and return a `Recordset` fitting the specifications of the `RecordSource` property.

To see any of the information in the records, you will have to put VB controls on the form and bind them to specific fields in the ADO Data Control's `Recordset`. Follow these steps to bind controls to the ADO Data Control's `Recordset`:

STEP BY STEP

8.6 Binding Controls to the ADO Data Control's `Recordset`

1. Place a VB control such as a `TextBox` on the form.
2. In the control's Properties window, activate the drop-down list on the `DataSource` property. The list will include the names of any ADO Data Controls on the current form, any Data Environment Designers in the current project, and any other types of `DataSource` (such as RDO or DAO Data Controls) available to this control (see Figure 8.21).

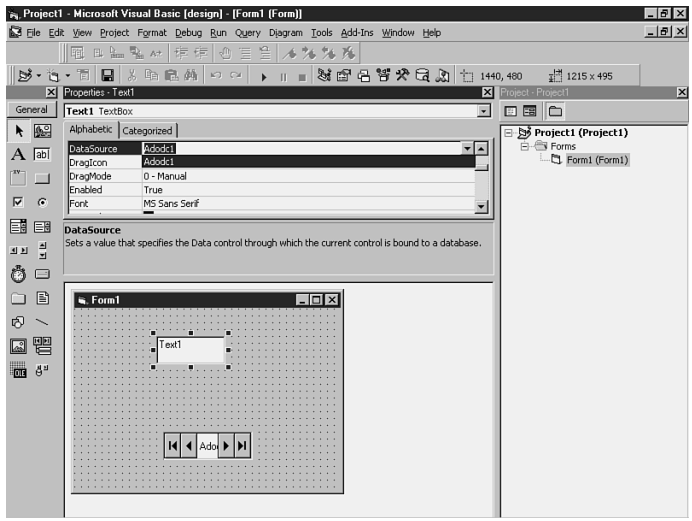


FIGURE 8.21
Setting the `DataSource` property for a control that you will bind to an ADO Data Control or another data source.

3. Choose as the `DataSource` the ADO Data Control whose `Recordset` you want to bind this control to.
4. Now set the control's `DataField` property by activating the drop-down list belonging to the `DataField` property in the Properties window. You should see a list of fields available from the ADO Data Control's `Recordset`. Choose the field that you want to bind to this control (see Figure 8.22).

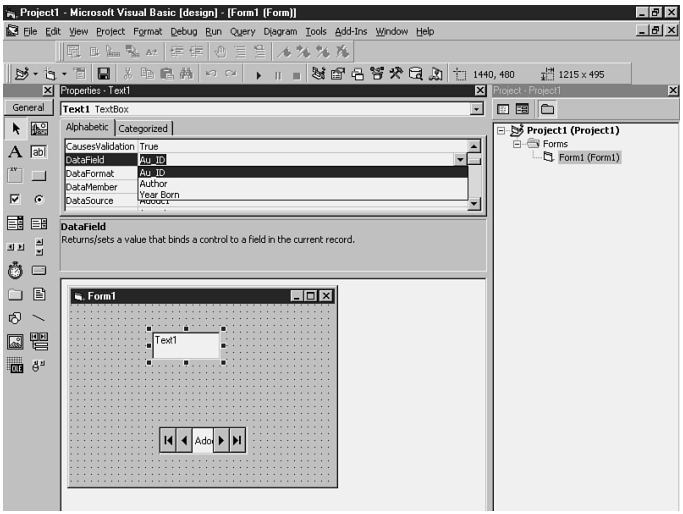


FIGURE 8.22
Setting the `DataField` property of a bound control.

5. The control is now bound to the desired field in the `Recordset`.

After you have bound controls to the ADO Data Control, you can run the project to test the connection. You should see data from the `Recordset` displayed in the bound controls. When you click the ADO Data Control's navigation buttons, you should see the contents of the bound controls change as the `Recordset`'s cursor move through different records.

Depending on the cursor options that you have chosen, you might also see that you can make changes to the underlying data fields by changing the contents of the bound controls.

Adding Records With the ADO Data Control

The user can view records and even write changes back to the data by just moving the record pointer.

To enable the user to add new records, you can do this:

- Set the Data Control's `EOFAction` property to `2- DoAddNew`. You can set this at design time in the Properties window or in code using the internal constant `adDoAddNew`.

The user will see and be able to edit a temporary blank record when there is an attempt to move past the last record in the `Recordset`.

- Call the `Recordset`'s `AddNew` method programmatically to add a record. `AddNew` blanks out the fields in the record buffer and refreshes the bound controls accordingly. The user sees the blank controls and can edit the fields of the potential new record (see Figure 8.23).

After the user has edited the originally blank copy buffer in either of these scenarios, the record must be saved in one of two ways:

- The user must move the record pointer.
- You can programmatically call the `Update` method.

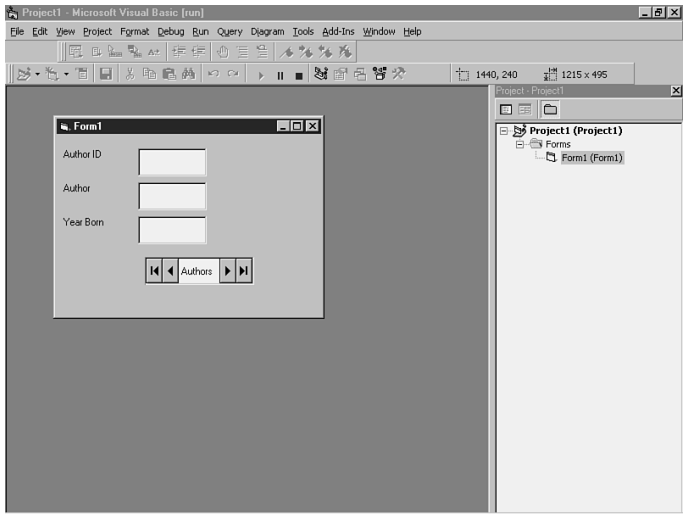


FIGURE 8.23
What the user will see at the end of file with the ADO Data Control's `EOFAction` property set to `adDoAddNew`, or when you call the `AddNew` method in code.

Editing Existing Records With the ADO Data Control

Any time the user edits controls that are bound to the Data Control, the user has changed information in the copy buffer.

Changes to records will save automatically when a user or the programmer moves the ADO Data Control `Recordset`'s record pointer away from the current record by any means, including the following:

- Using the `Find` method
- Resetting the `Recordset`'s bookmark—even if it's only to set the `Bookmark` back to the current record
- Using any of the `Move` methods on the `Recordset`
- Adding another record
- Clicking one of the four arrow buttons on the ADO Data Control

This is because moving the record pointer writes the contents of the record buffer out to the underlying data.

You also can save changes programmatically by calling the `Update` method of the `Recordset` or the `UpdateRecord` method of the ADO Data Control.

You can take more control of whether changes are saved by writing code in the `Will` event procedures, as described later.

Canceling Pending Editing Changes on Bound Controls

You can use the `CancelUpdate` method of the `Recordset` to cancel pending changes to the current record so that the changes just made won't be written to the underlying data.

The `CancelUpdate` method of the `Recordset` basically cancels a pending edit defined by the `AddNew` methods or by the user's or the program's changes to contents of fields in the record buffer.

If there is no pending edit, `CancelUpdate` is just ignored.

The `CancelUpdate` method doesn't move the record pointer. An example of a call to `CancelUpdate` is this:

```
adcEmployees.Recordset.CancelUpdate
```

Programming Other Actions on the ADO Data Control's `Recordset`

Other activities that you will want to perform programmatically on an ADO Data Control's `Recordset` include the following:

- Saving editing changes
- Deleting records
- Moving between records
- Locating records

How to program these actions is not discussed here because the techniques and the coding are the same as for straight ADO programming. See the sections later in this chapter on programming the ADO `Recordset` for more information on how to program these actions.

The ADO Control's `Error` Event

This event happens when the ADO Data Control's own internal data navigation and manipulation routines encounter a runtime error.

If the user clicks one of the ADO Data Control's navigation buttons and Visual Basic attempts to move to a record that has already been deleted, for instance, a runtime error could happen.

Because none of your code will be running when the user clicks the Data Control, you must put error-handling code for such eventualities in the ADO Data Control's `Error` event procedure.

The `Error` event's parameters help you to get information about the source and nature of the error that has fired this event. The final parameter also enables you to decide whether to allow VB to show an automatic error message to the user.

The parameters are as follows:

- **`ErrorNumber`** The internal error number.
- **`Description`** Error description string.
- **`Scode`** A native error code number from the data server.
- **`Source`** A String indicating where this error occurred.
- **`Helpfile`** Path and name of a help file containing information about this error (supplied by the object that raised this error).

WARNING

`Error` Event Procedure Code Not a Replacement for Your Own Error Handlers The ADO Data Control's `Error` event does not free you from writing error handlers in your own data manipulation code!

The `Error` event handles errors that are not caused by your code. You still must handle the runtime errors generated by your code.

- **`HelpContext`** A Help topic context number for the `Helpfile` (supplied by the object that raised this error).
- **`fCancelDisplay`** Set this parameter to `True` to keep VB from displaying an automatic error message to the user.

Other Events of the ADO Data Control

The remaining events of the ADO Data Control are just exposed events of the underlying `Recordset`. As such, they are identical to the `Recordset` object's events discussed later in this chapter. These events include the following:

- `EndOfRecordset`
- `FieldChangeComplete`
- `MoveComplete`
- `RecordChangeComplete`
- `RecordsetChangeComplete`
- `WillChangeField`
- `WillChangeRecord`
- `WillChangeRecordset`
- `WillMove`

Refer to the discussion of these events in the sections of this chapter that discuss `Recordset` object events.

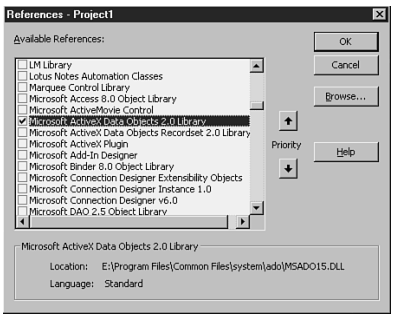
Programming With ADO

So far, this chapter has mostly discussed automatic ways to set up ADO `Connection`, `Command`, and `Recordset` objects using Data Environment Designers or the ADO Data Control.

For most serious applications, however, you will need to directly program the ADO object model in VB code.

To program ADO objects directly, you must set your VB project to refer to the latest version of the Microsoft ActiveX Data Objects Library from the Project, References menu dialog box, as shown in Figure 8.24.

FIGURE 8.24
Setting a project reference to the ADO Library.



You basically have three main concerns when you program with ADO in VB:

- Setting up and maintaining a connection to data with the `Connection` object
- Retrieving rows or otherwise manipulating the data with the `Recordset` object, the `Command` and `Parameter` objects, or the `Connection` object
- Determining the exact behavior and nature of the rows of data returned by a `Connection` or `Command` and manipulating the data's individual fields and rows with the `Recordset` and `Field` objects

The following sections discuss the programming of these ADO objects.

Initializing the `Connection` Object in Code

A `Connection` object provides you with a connection to the data that the `Command` object will operate on and the `Recordset` will retrieve.

If you plan to connect your `Recordset` to data outside of your application, you must have at least one `Connection` object in the application.

The minimum steps you need to take to have a functional `Connection` object are as follows:

STEP BY STEP

8.7 Establishing a `Connection` Object

1. Make sure that you have set a reference to the ADO Library, as discussed in Chapter 8, under the section "Programming with ADO."
2. Declare an object variable whose type is `ADODB.Connection`. If you want to program the event procedure of the object, use the `WithEvents` keyword.
3. Set the `Connection` object's `ConnectString` property to reflect a valid OLE DB provider, or specify the provider string in the next step, as follows:

```
cnNWind.ConnectString = "Provider=Microsoft.
➥Jet.OLEDB.3.51;"
```

4. Call the `Connection` object's `Open` method. If you didn't specify the `ConnectString` property in the preceding step, you need to pass the connection string as the first argument to the `Open` method, as in the following example :

```
cnNWind.Open  "Provider=Microsoft.Jet.
➥OLEDB.3.51;" & _
"Data Source= C:\DataSamples\Nwind.mdb"
```

Connection Object Events

Only two ADO object class types support events: the `Connection` and the `Recordset`. You can therefore declare object variables of these two types using the `WithEvents` keyword.

NOTE

Further References for the `Execute` Method See more information about the `Execute` method in "Using Stored Procedures to Return Records to an Application" in Chapter 9, and the sections under "ADO Data Access Models" in the next chapter.

NOTE

Can't Use `As New` in a Declaration with `WithEvents` You can't use the `As New` keyword in a declaration that uses the `WithEvents` keyword. This means that an object variable doesn't get instantiated when you declare it using `WithEvents`. Therefore, when your code is ready to initialize an ADO `Connection` or `Recordset` object that has been declared using `WithEvents`, you should use a statement of this form:

```
Set objName = New Class
```

The `Connection` Object's `Will` Events

The `Connection` object has two `Will` events:

- The `WillConnect` event, which happens just before a connection to a provider
- The `Execute` event, which happens just before a pending command executes on the current connection

Both events have an `adStatus` parameter that enables you to cancel the pending action. In addition, they have other information that tells you about the settings of the proposed open connection or command.

Both event procedures are therefore ideal places to validate the pending actions and cancel them if necessary, as discussed in the following sections.

The `WillConnect` Event

You can put code in the `Connection` object's `WillConnect` event procedure to monitor information about the pending `Connection` that's about to be opened to a provider. You can also validate and, if necessary, cancel the pending connection by setting the `adStatus` parameter to a value of `adCancel`.

The most important parameter of `WillConnect` is the `adStatus` parameter, which you can set to `adStatusCancel` to stop the pending execution. You can also set `adStatus` to `adStatusUnwantedEvent` to prevent the event from firing again.

Setting the `adCancel` parameter to `adStatusCancel` will have no effect if the original value is `adStatusCantDeny`.

The other parameters (`ConnectionString`, `UserID`, `Password`, and `Options`) represent settings of the current `Connection` object that's about to open. You can change them here to change the behavior of the new connection that's about to be opened.

The `WillExecute` Event

The `WillExecute` event's name implies a close link to the `Execute` method of a `Connection` object, and it's true that it will fire when a `Command` object's `Execute` method runs.

`WillExecute` does not just happen when a `Connection` object's `Execute` method runs, however. It also can happen whenever a `Recordset` object that depends on the current `Connection` object is opened, regardless of whether that `Recordset` object was initialized by a `Connection` object's `Execute` method.

The parameters for `WillExecute` enable you to examine and change the settings for the action that will be executed on the provider, and even to cancel the execution altogether.

The most important parameter of `WillExecute` is the `adStatus` parameter, which you can set to `adStatusCancel` to stop the pending execution. You can also set `adStatusCancel` to `adStatusUnwantedEvent` to prevent the event from firing again.

Setting the `adStatus` parameter to `adStatusCancel` will have no effect if the original value is `adStatusCantDeny`.

The other parameters (`Source`, `CursorType`, `LockType`, and `Options`) represent settings of the current request that's about to execute. You can change them here to change the behavior of the request and the behavior of any `Recordset` that may be created.

The `Connection` Object's Transaction Completion Events

The more important of these events include the following:

- `BeginTransComplete`
- `CommitTransComplete`
- `RollbackTransComplete`

These events are described in greater detail in the following chapter, "Creating Data Services: Part II" in the section titled "Managing Database Transactions."

The `Connection` Object's `ConnectComplete` Event

The `ConnectComplete` event has the following parameters:

- **`pError`** An `Error` object containing either `Nothing` or a description of a connection error (the value of the `adStatus` parameter will be `adStatusErrorsOccurred` in this case).
- **`adStatus`** Can be `adStatusOK` or `adStatusErrorsOccurred`, or `adStatusCancel`. It's `adStatusCancel` if the preceding `WillConnect` event procedure cancelled the connection.

You can also set it to `adStatusUnwantedEvent` if you don't want to see this event fire again during the current session.

- **`pConnection`** Not used in VB.

The `Connection` Object's `ExecuteComplete` Event

The `ExecuteComplete` event has the following parameters:

- **`pError`** an `Error` object containing either `Nothing` or a description of a connection error (the value of the `adStatus` parameter will be `adStatusErrorsOccurred` in this case).
- **`adStatus`** Can be `adStatusOK` or `adStatusErrorsOccurred`. You can also set it to `adStatusUnwantedEvent` if you don't want to see this event fire again during the current session.
- **`pCommand`** If the action just executed was based on a `Command` object, this parameter points to the `Command` object. Otherwise, its value is `Nothing`.
- **`pRecordset`** If the action just executed returned a `Recordset`, this parameter points to the `Recordset`. Otherwise, its value is `Nothing`.
- **`pConnection`** Not used in VB.

The `Disconnect` Event

The `Disconnect` event happens after the `Connection` is closed with the `Close` method or by going out of scope.

It takes as its first parameter `adStatus`, indicating whether there were errors upon disconnection (`adStatusOK` or `adStatusErrorsOccurred`).

Its second parameter points to the current `Connection` object and is not needed or used in VB programming.

You can put code in the `Disconnect` event procedure to perform post-connection cleanup.

Initializing `Command` Objects in Code

To programmatically initialize a `Command` object, you should take the following steps:

1. Make sure that you have a valid `Connection`, as discussed earlier in the section "Programming the `Connection` Object."
2. Declare an object variable of the type `ADODB.Command`.
3. Set the `CommandType` property.
4. Set the `CommandText` property.
5. Call the `Execute` method.
6. Set the `ActiveConnection` property to point to a valid existing `Connection` object.

NOTE

Further References For examples and more information about the `Execute` method, see "Using Stored Procedures to Return Records to an Application" in Chapter 9 and the sections under "ADO Data Access Models" in the next chapter, "Creating Data Services: Part II."

Also see "Using the Parameters Collection to Access Parameters for Stored Procedures" in the next chapter for information about how to pass parameters to a stored procedure.

`Command` Object Events

Surprise! ADO `Command` objects don't have any events. If you try to declare a `Command` object variable using the `WithEvents` keyword, you will receive a compiler error. Only `Connection` and `Recordset` objects support events in the ADO object model.

EXAM TIP

`Command` Objects Questions
Beware of trick questions on the exam that assume you can program with `Command` object events.

`Recordsets`

The general steps you need to take to initialize a data-connected `Recordset` in your code are as follows:

1. Make sure you have a valid `Connection` or `Command` object.
2. Declare an object variable of the type `ADODB.Recordset`.
3. Set the `Source` property (typically, a SQL statement or the name of a stored procedure or table) and the `ActiveConnection` property (use the `Set =` syntax to cause this property to point to a valid ADO `Connection` object). You can also omit this step and pass information about the `Source` and `ActiveConnection` as arguments in the next step.
4. Call the `Recordset`'s `Open` method. If you omitted step 3, indicate the `Recordset`'s `Source` and `ActiveConnection` as the `Open` method's first and second arguments, respectively.

Listing 8.2 illustrates the property-driven technique described in step 3 for opening a `Recordset`.

LISTING 8.2

OPENING AN ADO RECORDSET BY SETTING THE ACTIVECONNECTION AND SOURCE PROPERTIES

```
    Set cnNWind = New ADODB.Connection
    Set rsEmployees = New ADODB.Recordset
    Dim sConnect As String
    sConnect = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
               "Data Source= NWind.mdb"
    cnNWind.Open sConnect
    rsEmployees.Source = "Select * From Employees Order By
LastName,FirstName"
    Set rsEmployees.ActiveConnection = cnNWind
    rsEmployees.Open
```

Listing 8.3 illustrates the use of command-line arguments to accomplish the same result, as discussed in step 4.

LISTING 8.3

OPENING AN ADO RECORDSET WITH ARGUMENTS TO THE OPEN METHOD

```
    Set cnNWind = New ADODB.Connection
    Set rsEmployees = New ADODB.Recordset
    Dim sConnect As String
    sConnect = "Provider=Microsoft.Jet.OLEDB.3.51;" & _
               "Data Source= NWind.mdb"
    cnNWind.Open sConnect
    rsEmployees.Open  _
      "Select * From Employees Order By LastName,FirstName", _
      cnNWind
```

NOTE

Other Methods for Opening a `Recordset` The section titled "Accessing Data with the Execute Direct Model" in the following chapter discusses how to use the `Execute` method of ADO `Connection` and `Command` objects to open a `Recordset`.

Manipulating a `Recordset`'s Data With Its Methods

The most direct manipulation of data with ADO takes place through the `Recordset`. The methods for ADO's `Recordset` object are basically the same as the methods for the `Data Environment`'s or ADO Data Control's `Recordset` objects. You can therefore read the following sections on the specific `Recordset` methods and properties as applying to both the `Recordset` of ADO, and to the `Recordset` that belongs to the `Data Environment` and ADO Data Control.

One activity varies significantly between the two classes of `Recordset`, however: Notice that the technique for adding a record in ADO code significantly differs from the technique for adding a record for the `Data Environment` or ADO Data Control.

There is no design-time binding of controls to the data as there is with the `Data Environment` or the ADO Data Control. Because straight ADO objects lack this automatic binding of controls to data, the programmer must write code to refresh variables or user-interface controls whenever any action happens that would move the record pointer or otherwise change the contents of the fields.

Similarly, the programmer must explicitly move data from controls or variables to the record buffer whenever data should be saved.

Because a data access program must perform these two tasks so often, it is most efficient for the programmer to provide one general routine to read data from the record buffer into controls and another general routine to write data from controls into the record buffer. The program can then call these routines whenever it needs to perform these tasks.

The following sections, "Referring to `Recordset` Field Contents," "Programmatically Reading a Record into VB Controls," and "Programmatically Writing VB Controls to a Record," describe how you can write routines to manually refresh data in both directions (reading and writing) when you directly program ADO.

Referring to `Recordset` Field Contents

Every open `Recordset` must be associated with a *data cursor*.

Behind every `Recordset` cursor, there is, among other things, a buffer representing the values of fields in the current record that the cursor points to. The `Fields` collection of the `Recordset` exposes this record buffer.

You can always programmatically read the values of individual fields in the record buffer. Depending on whether the current `Recordset`'s cursor type permits writes to the data, you can also assign values to the record buffer's fields.

There are several syntactic styles for referring to an individual field in the current `Record`, as follows:

- You can use the numeric index of the field in the `Fields` collection (`Fields` is zero-based, so the first field is element `0` in the collection):

  ```
  rs.Fields(2).Value
  ```

  This technique is flexible (you could use a numeric variable as the index for the `Fields` collection), but it's not quite as useful as the following technique, because unless you know the position of fields in the data, you will have a hard time getting the right index.

- Because the `Fields` collection also supports index key strings, you can use the field's name in a string literal or variable to refer to the field in the `Fields` collection:

  ```
  rs.Fields("LastName").Value
  rs("LastName").Value
  ```

  Notice the second alternative form of this example, which bypasses an explicit reference to the `Fields` collection. This technique is perhaps the most useful from a programming standpoint, but it also requires the most runtime overhead and so is the slowest of the three techniques.

- You can also refer to the field as a temporary property of the `Recordset`, using the bang (`!`) syntax:

  ```
  rs!LastName
  ```

  This technique is the most efficient, but it's the least flexible of the three (because you hard-code the field name in the program, whereas you could substitute variables for the index and key values in the first and second techniques).

As already mentioned, you can both read and write these fields programmatically. However, no changes are transferred to the underlying data until you call the `Update` method as discussed in the following section.

Unless you need flexibility at runtime, the last method listed is probably the best to use; it's fast and explicitly identifies the field you want to access. If you want to read the contents of the current record's `Last Name` field into the `Text` property of `txtLastName`, for example the line would look like this:

```
txtLastName.Text = rsEmployees![Last Name] & ""
```

The `Text` property of a `TextBox` control does not accept null data. The use of the final characters `& ""` at the end of the line ensures that, even if the underlying field contains null data, an error will not occur. The `& ""` makes sure that at least a blank string is contained in the data being written to the `TextBox`.

Programmatically Reading a Record's Contents Into VB Controls

Typically, you will write a procedure such as that of Listing 8.4 to populate controls with field contents from a `Recordset`. You will call such a routine from every place in your application that potentially updates the record pointer.

The `MoveComplete` event procedure is often the best place from which to call such code.

LISTING 8.4

ROUTINE TO POPULATE CONTROLS FROM COPY BUFFER

```
Sub PlaceDataInControls()
    txtFirstName.Text = rsEmployees![First Name] & ""
    txtLastName.Text = rsEmployees![Last Name] & ""
    txtDepartment.Text = rsEmployees!Department & ""
    txtPhoneExt.Text = rsEmployees!PhoneExt & ""
    'assumes field will contain a valid
    'CheckBox value (0, 1, or 2)
    chkFullTime.Value = rsEmployees![Full Time]
End Sub
```

As an alternative to the use of the `& ""` characters, you could also trap null data more explicitly with code such as that shown in Listing 8.5.

LISTING 8.5

LOGIC TO EXPLICITLY TEST FOR NULL DATA

```
If IsNull(txtFirstName.Text) Then
txtFirstName.Text = rsEmployees![First Name]
Else
txtFirstName.Text = "<<NULL>>"
End If
```

NOTE

Square Brackets Around Field Names

Some DBMSs support spaces in their field names. MS Access supports spaces in field names, for example, but SQL Server does not. In Access and SQL Server 7.0, a field named `"Last Name"` would be acceptable, but in SQL Server 6.5 and before, it would not be acceptable. You must place the square bracket characters "[]" in your code around field names that have spaces, as in these examples:

```
Rs.fields("[Last Name]")
Rs![Last Name]
```

For consistency you can place square brackets around field names that don't contain spaces as well, but there is no need to do so.

Notice, however, that this second method requires you to write several lines of code, as opposed to just appending `& ""` to the field contents.

Programmatically Writing VB Controls to a Record

When the user makes changes to controls and you want to save to the underlying data, you must programmatically write the contents of a control back to the underlying data in two steps:

1. Write the contents of each control to its corresponding field in the copy buffer.
2. Save the contents of the copy buffer to the underlying data of the `Recordset`.

To update a field in the copy buffer with a control's contents, you might code the following:

```
RecordsetName![FieldName] = ControlContents
```

If you want to write the `Last Name` field of `rsEmployees` that you read in the last section, for example, you could write:

```
rsEmployees![Last Name] = txtLastName.Text
```

You would typically write a general procedure to write controls to their corresponding fields, with one line of code similar to the preceding example for each control-field assignment (see Listing 8.6 in the section titled "Updating a Record"). You could call this procedure as the first step in saving data from controls in the `Recordset`'s underlying data.

The second phase of writing to a record requires the use of the `Recordset`'s `Update` method. This phase is discussed in the following section, "Updating a Record."

Updating a Record

To write the contents of a `Recordset` object's copy buffer to its underlying data, you must follow these steps:

1. Write to each field in the copy buffer, as described in the preceding section.
2. Call the `Recordset`'s `Update` method.

Listing 8.6 illustrates using these steps to update a record. To review the purpose of the `WriteEmployeeRecord` procedure, refer to the above section, "Programmatically Writing VB Controls to a Record."

LISTING 8.6

ROUTINES FOR WRITING DATA FROM CONTROLS BACK TO THE DATABASE

```
Sub UpdateRecord
   'write controls to buffer
    WriteEmployeeRecord
    'call the Update method
    rsEmployees.Update
End Sub

Sub WriteEmployeeRecord()
    rsEmployees![First Name] = txtFirstName.Text
    rsEmployees![Last Name] = txtLastName.Text
    rsEmployees!Department = txtDepartment.Text
    rsEmployees!PhoneExt = txtPhoneExt.Text
    rsEmployee![Full Time]= chkFullTime.Value
End Sub
```

Canceling User Changes Before They Are Saved

To cancel pending user edits to the current record in unbound screen controls, you can call whatever procedure you have written to programmatically refresh controls with the data from the current record.

This action will cause the controls to reflect the existing state of the field in the record buffer, therefore overwriting any changes the user has made to controls.

The code for canceling user edits might look like Listing 8.7.

LISTING 8.7

CANCELING PENDING CHANGES

```
Private Sub cmdCancel_Click()
    'Call General procedure to populate
    'controls from current record
    PlaceDataInControls
End Sub

Private Sub PlaceDataInControls
    txtFirstName = rs!FirstName & ""
    txtLastName = rs!LastName & ""
    'and so on for each field
End Sub
```

Adding a Record

To add new records to a `Recordset` programmatically, you can use a combination of the `Recordset`'s `AddNew` and `Update` methods.

`AddNew` appends a temporary record buffer to the cursor's rowset.

You can take the following steps in code to add a new record:

1. If you are providing controls to the user for field editing, set the controls to blank or default values. Next, enable the user to add new data to the controls.
2. When the user is ready to save the new data, invoke the `AddNew` method of the `Recordset`. The cursor is now pointing to a temporary new record.
3. Assign the desired values to the individual fields of the current record (the temporary blank record). If you have given the user controls to edit field contents, then assign those control contents to the fields.
4. Call the `Recordset`'s `Update` method.

The code to implement these four steps might be contained in the `Click` event procedures for buttons with captions such as `Add` and `Save New`. The code in Listing 8.8 provides you with such an example of how to add a new record.

LISTING 8.8

ADDING AND SAVING A NEW RECORD

```
Private Sub cmdAdd_Click()
     cmdSaveNew.Enabled = True
     txtLastName = ""
     txtSalary = "0"
     txtFirstName = ""
End Sub

Private Sub cmdSaveNew_Click()
     rsEmployees.AddNew
     WriteControlsToData 'our routine to update copy buffer
     rsEmployees.Update
     cmdSaveNew.Enabled = False
End Sub
```

To enable the user to cancel adding a record while the user is editing fields, all you need to do is call the routine that refreshes controls from the copy buffer fields.

If you have already updated copy buffer fields programmatically, you will want to call the `CancelUpdate` method as well.

Deleting a Record

The `Recordset`'s `Delete` method will delete a record from the underlying data. Typically, you will want to take the record pointer to a different record after calling the `Delete` method. Before moving the record pointer, you should check the `RecordCount` property to make sure that at least one record is left in the `Recordset`.

After moving the record pointer, of course, you will need to check the `Recordset`'s `EOF` property to make sure you haven't moved beyond the end of the data. If you have, you will want to call the `MoveLast` method—but first check the `BOF` property to make sure that there are any records at all remaining in the `Recordset`. The code for these operations might look like Listing 8.9.

NOTE

Differences in Calls to `AddNew`

The timing of your calls to the ADO `Recordset`'s `AddNew` method will probably differ from the timing of your call to the `AddNew` method of the `Recordset` belonging to a `Data` Environment or to an ADO Data Control when there are bound controls.

Whereas you might call the ADO Data Control or `Data Environment` `Recordset`'s `AddNew` method as soon as the user decides to add a record, you probably don't want to call the ADO `Recordset`'s `AddNew` method until it's time to save the edited data for the new record. You don't want to allocate extra resources before you need to.

LISTING 8.9

DELETING A RECORD

```
Private Sub cmdDelete_Click()
     rsEmployees.Delete
     rsEmployees.MoveNext
     If rsEmployees.EOF Then
         If rsEmployees.BOF Then
             Msgbox ìNothing to Deleteî
             cmdDelete.Enabled = False
         Else
             RsEmployees.MoveLast
         End If
     End If
End Sub
```

That particular example makes users aware of what they have done if they accidentally tried to delete from a `Recordset` that contained no records.

Programmatically Navigating a `Recordset`

The `Recordset`'s five most common methods that enable you to programmatically position the record pointer are as follows:

- **`Move`** This method takes a positive or negative Long value as a required parameter. The parameter specifies the number of records to move away from the current record pointer position. Positive values indicate forward movement, while negative values indicate backward movement. An optional second parameter enables you to specify the `Bookmark` of a different record. Specifying this second parameter causes the movement to happen relative to the record of the `Bookmark`.
- **`MoveFirst`** Moves the record pointer to the first row of the `Recordset's` data.
- **`MoveLast`** Moves the record pointer to the last row of the `Recordset`'s data.
- **`MoveNext`** Moves the record pointer one row beyond its current position in the `Recordset`.
- **`MovePrevious`** Moves the record pointer one row before its current position in the `Recordset`.

You might call these methods to programmatically process records, or you might call them in response to some user action, such as clicking buttons labeled `Next`, `Previous`, `First`, or `Last`.

It is possible to move the record pointer too far (that is, past the beginning or end of the `Recordset`) with the `Move`, `MoveNext`, and `MovePrevious` methods. To help you avoid this problem, each `Recordset` has "buffer records" just before its first row and just after its last row. When you move the record pointer onto one of the beginning or ending buffer records, no error happens, but the `Recordset`'s Boolean property `BOF` (Beginning-of-file) or `EOF` (End-of-file) becomes `True`.

You should always test the `BOF` property immediately after calling the `MovePrevious` method and the `EOF` property after every call to `MoveNext`, and you should test one or both of the properties after calling the `Move` method. The examples in Listing 8.10 present code that you might put in the `Click` event procedures for Next and Previous `CommandButtons`. (Notice the call to `ReadFromData`, a procedure the programmer has written to populate controls with field data from the `Recordset`'s copy buffer.)

LISTING 8.10

Using the EOF and BOF Properties With MoveNext and MovePrevious

```
Private Sub cmdNext_Click()
     rsEmployees.MoveNext
     If rsEmployees.EOF Then
          rsEmployees.MoveLast
     EndIf
     ReadFromData
End Sub

Private Sub cmdPrevious_Click()
     rsEmployees.MovePrevious
     If rsEmployees.BOF Then
          rsEmployees.MoveFirst
     EndIf
     ReadFromData
End Sub
```

If you programmatically loop through a `Recordset`, you must also check for the `EOF` property. You can perform this type of navigation by writing a loop that keeps advancing the record pointer with `MoveNext` until `EOF` is `True`. An example of a record-processing loop might look like the code in Listing 8.11.

LISTING 8.11

A Record-Processing Loop

```
rsEmployees.MoveFirst
Do Until rsEmployees.EOF
     '...some code to process a record
     rsEmployees.MoveNext
Loop
```

In this example, you always start at the first record in the `rsEmployees Recordset` and go through the entire `Recordset` with `MoveNext`.

Locating Records

You can use the `Recordset`'s `Find` method to move the cursor to a record that fits a specified criterion.

The `Find` method takes up to four arguments:

- **`Criterion` (required)** A string with the same syntactic format as a SQL where clause. It specifies a condition that the record being sought must fulfill.
- **`SkipRows`** A number representing the offset of the starting position of the search from either the current row or from the row indicated by the `Start` argument. Assumed to be `0` if left blank.
- **`searchDirection`** A flag indicating the direction in which to search from the starting point. Values can be `adSearchForward` or `adSearchBackward`. Assumed to be `adSearchForward` if left blank.
- **`Start`** A Double-type value that gives the `Bookmark` of the record from which the search will begin in the direction indicated by `searchDirection`. Assumed to be the current row if left blank.

When you call the `Find` method, of course, it is not always certain that you will find a record that fits your criteria. If the `Find` method does not locate any records, the cursor ends up at the very beginning or end of the `Recordset` (depending on the setting of the `searchDirection` argument), and the `BOF` or `EOF` property will be `True`.

You should always check the `BOF` and `EOF` properties after you perform a `Find` so that you can gracefully recover from an unsuccessful attempt to find records.

Listing 8.12 gives an example of the use of the `Find` method. In this implementation, the user enters a string or partial string in a `TextBox`. The code incorporates the string into a condition and uses this condition as the `Criterion` argument to the `Find` method.

LISTING 8.12

USING THE FIND METHOD TO LOCATE RECORDS IN A RECORDSET

```
Private Sub cmdFind_Click()
    'Note:  Bookmark only works correctly
    'with Client-side cursors
    Dim dBookmark As Double
    dBookmark = rsEmployees.Bookmark
    Dim sFindCriterion As String
    sFindCriterion = "LastName like '" & _
                     txtLastNameToFind & "*'"
    rsEmployees.MoveFirst
    rsEmployees.Find sFindCriterion, , adSearchForward
    If rsEmployees.EOF Then
        rsEmployees.Bookmark = dBookmark
        MsgBox "Couldn't Find """ & txtLastNameToFind & """*"
    Else
        txtLastNameToFind = ""
    End If
End Sub
```

The `Recordset`'s `Bookmark` Property

If your search with the `Find` method was unsuccessful—a fact that you can detect by checking the `EOF` or `BOF` property—you're going to need to recover somehow, because the record pointer will probably end up at the very first or very last record of the `Recordset`. As noted in the preceding section, this won't cause an error, but it probably will be confusing to the user.

Even if you programmatically check the `EOF` or `BOF` property and display some sort of error message in a message box, your user may still be inconvenienced. This is because the user would probably just want to carry on with the record that was current before initiating the unsuccessful search.

Why not put the record pointer back to the record it was on before the unsuccessful search began?

Luckily, the `Recordset` object supports a `BookMark` property that enables you to store and, if necessary, reset the position of the record pointer. Therefore, the strategy for using the `Find` method would as follows:

WARNING

`Bookmark` Property Limited to `ClientSide` Cursors The `Bookmark` property only contains valid information for `ClientSide` cursors.

If you try to use a `Recordset`'s `Bookmark` property with a `ServerSide` cursor, you will receive a runtime error.

1. Store the `Bookmark` property to a `Double` variable.
2. Perform the `Find`.
3. If the `Find` is unsuccessful, restore the stored value of the `Bookmark` property and show an error message.

Listing 8.12 in the preceding section illustrates how to use the `Bookmark` property to restore cursor position after an unsuccessful call to the `Find` method.

`Recordset` Events

NOTE

Reminder See the section titled "`Connection` Object Events" for a detailed reminder note about programming objects using the `WithEvents` keyword.

As mentioned in the section titled "`Connection` Object Events," the `Recordset` is one of but two ADO objects that support events and that you can declare using the `WithEvents` keyword.

`Recordset` event names and functionality follow a similar pattern to those of `Connection` events: They are mostly divided into two groups. One group, the `Will` events, happens just before some action is about to take place; the other group, the `Complete` events, happens just after an action has occurred.

The `EndOfRecordset` Event

This event fires when the cursor attempts to move past the first or last row of the `Recordset` and the `BOF` or `EOF` property becomes `True`.

NOTE

`BookMark` Versus `AbsolutePosition` You might think that the `AbsolutePosition` property of the `Recordset` would work for storing and resetting the record pointer's position. However, a record's `AbsolutePosition` can change as other records are added or deleted. A `BookMark` is more stable because it always points to the same physical location for the record.

Also note that, like the `Bookmark`, the `AbolutePosition` property only works for `ClientSide` cursors.

You can use this event to add new records at the end of the `Recordset` when the record pointer attempts to navigate beyond the last record.

The `EndOfRecordset` event's parameters are as follows:

- **`fMoreData`** A Boolean that you must set to `True` if you add records during the course of this event procedure. This signals ADO to find a new end of the `Recordset`.
- **`adStatus`** If it is `adStatusOK`, you can set the value of `adStatus` to `adStatusCancel` (the EOF action is cancelled), to `adStatusCantDeny` (meaning that you can't cancel this event in the future) or `adStatusUnwantedEvent` (meaning that the event won't fire again).

- **pRecordset** Points to the current Recordset that raised this event (not needed in VB programming).

The Will Events

These Recordset events' names all begin with the word *Will* (hence, the term Will events). Each Will event happens just before some action on the Recordset.

A list of the Recordset's Will events follows:

- WillChangeField
- WillChangeRecord
- WillChangeRecordset
- WillMove

Probably the most important parameter of each Will event is the adStatus parameter: It tells you what the current status of the Recordset is, but, more interestingly, you can change its value in the event procedure to either

- prevent the pending action from occurring (set adStatus to adStatusCancel)

or

- change the event's behavior for the rest of the current session: set adStatus to adStatusUnwantedEvent to stop the event from firing again for this Recordset.

A brief description of each Will event and its parameters follows:

- **WillMove** Fires when the current row is about to change.
  - **adReason** An integer specifying the reason that the move is going to occur. Possible values for this parameter in the context of a move are these: adRsnMoveFirst, adRsnMoveLast, adRsnMoveNext, adRsnMovePrevious, adRsnMove, adRsnRequery.

- **`adStatus`** See earlier discussion.
- **`pRecordset`** Pointer to current `Recordset` (not used in VB).

- **`WillChangeField`** Fires when some action will cause one or more fields in the record buffer to change. This could be due to a user edit with the ADO Data Control or due to an assignment of a field's value in your code.
  - **`cFields`** Number of fields that will be affected, same as number of elements of the `Fields` parameter.
  - **`Fields`** Variant array of `Field` objects representing the fields in this record to be changed with this event.
  - **`adStatus`** See earlier discussion.
  - **`pRecordset`** Pointer to current `Recordset` (not used in VB).

- **`WillChangeRecord`** Fires when one or more records in the underlying data are to be changed through deletion, addition, or writing changes from the record buffer to the underlying data.
  - **`adReason`** An integer specifying the reason that records are going to be changed. Possible values for this parameter in the context of a record change are these: `adRsnAddNew`, `adRsnDelete`, `adRsnFirstChange`, `adRsnUndoAddNew`, `adRsnUndoDelete`, `adRsnUndoUpdate`, `adRsnUpdate`.
  - **`cRecords`** Number of records that will be affected with this event.
  - **`adStatus`** See earlier discussion.
  - **`pRecordset`** Pointer to current `Recordset` (not used in VB).

- **`WillChangeRecordset`** Fires before some action that will change the entire `Recordset` across the board, including setting the `Recordset` to `Nothing`.

- **`adReason`** An integer specifying the reason that the `Recordset` is going to be changed. Possible values for this parameter in the context of a Recordset change are these: `adRsnRequery`, `adRsnResynch`, `adRsnClose`, `adRsnOpen`.
- **`adStatus`** See earlier discussion.
- **`pRecordset`** Pointer to current `Recordset` (not used in VB).

The `Complete` Events

There is a correspondingly named `Complete` event for each of the `Recordset`'s `Will` events described earlier. Just as a `Will` event fires *before* the actual completion of an action, so a `Complete` event fires *after* the action has completed. The names of the `Complete` events are as follows:

- `ChangeFieldComplete`
- `ChangeRecordComplete`
- `ChangeRecordsetComplete`
- `MoveComplete`

The `Complete` events all take the same parameters in the same order as their respective `Will` events, with one addition: A `pError` parameter that comes just before the `adStatus` parameter in all four event pairs and contains an `Error` object that gives information about any error that occurred.

You can, of course, find out whether an error occurred by checking the `adStatus` parameter for the value `adStatusErrorsOccurred`.

Disconnected, Persistent, and Dynamic `Recordsets`

Because ADO has a "flat" object model hierarchy, with few dependent objects (that is, few objects that can only be accessed through other, parent objects), ADO `Recordsets` present some new possibilities that were not available with earlier Microsoft data object models, such as DAO.

In particular, a `Recordset` does not always need to be connected to a `Connection` object. In fact, it is possible to create a `Recordset` that never uses a `Connection` object or connects to data outside your application.

At first glance, these possibilities may seem more like curiosities than like practical features of the `Recordset` object, but let's look a bit more closely at the different possibilities of unconnected `Recordsets`:

- **Disconnected `Recordset` objects**. Enable you to reduce server overhead by processing data offline.
- **Dynamic `Recordset` objects**. Give you the full power of the ADO model to manipulate data that's completely internal to your application.
- **Persistent `Recordset` objects**. Free you from having to complete all actions on a `Recordset` during a single session of the application.

The following sections discuss these three types of `Recordset` objects.

Disconnected `Recordsets`

A disconnected `Recordset` object gets data from the server, but then goes offline to manipulate the data and reconnects to write the changes back to the server.

Such `Recordsets` must, of course, be implemented with client-side cursors.

They begin life connected to a `Connection` object. After the connection is made to the server and the `Recordset` retrieves its rows from the server, however, you can disconnect the `Recordset` from the `Connection` object and drop the `Connection` to the server entirely.

Your process or the user can then manipulate the `Recordset` completely offline.

If and when your application decides to update the server data with the offline changes, you can reconnect to a `Connection` object and send the changes to the server database.

You must take the following steps in code to implement a disconnected `Recordset`:

STEP BY STEP

8.8 Implementing a Disconnected `Recordset`

1. Create a `Recordset` object with its `CursorLocation` property set to `ClientSide`.
2. Connect the `Recordset` to a `Connection` object with the `Open` method.
3. After you have retrieved rows into the `Recordset`, set the `Recordset`'s `ActiveConnection` property to `Nothing` and close the `Connection` object. The `Recordset` is now disconnected.
4. Manipulate the data in the `Recordset` through user edits and/or your own processing logic.
5. When local processing is done, you can reestablish the `Recordset`'s connection to the server by re-opening the `Connection` object and reconnecting the `Recordset` to the `Connection` object.

Listing 8.13 gives an example of code that implements a disconnected `Recordset` with offline processing.

LISTING 8.13

CODE THAT IMPLEMENTS A DISCONNECTED RECORDSET

```
    'OPEN AN ADO CONNECTION
    Dim connPubs As ADODB.Connection
    Dim sConnString As String
    sConnString = "Provider=MSDASQL.1;Data
Source=TRASH1006;Initial Catalog=pubs"
    Set connPubs = New ADODB.Connection
    connPubs.ConnectionString = sConnString
    connPubs.Open

    'OPEN A RECORDSET FROM THE CONNECTION
    'BUT THEN DESTROY THE CONNECTION
    Dim rs As ADODB.Recordset
```

continues

LISTING 8.13 *continued*

CODE THAT IMPLEMENTS A DISCONNECTED RECORDSET

```
Set rs = New ADODB.Recordset
With rs
    ' Specify the cursor's location
    .CursorLocation = adUseClient
    .LockType = adLockBatchOptimistic
    .Source = "Select * from authors"
    Set .ActiveConnection = connPubs
    .Open
    .ActiveConnection = Nothing
End With
connPubs.Close
Set connPubs = Nothing

'DO SOME PROCESSING TO THE DISCONNECTED
'RECORDSET
'........
'RE-OPEN AN ADO CONNECTION
Set connPubs = New ADODB.Connection
connPubs.ConnectionString = sConnString
connPubs.Open

'CONNECT THE RECORDSET TO THE CONNECTION
Set rs.ActiveConnection = connPubs
'UPDATE THE CONNECTION WITH THE RECORDSET'S
'CHANGES
rs.UpdateBatch

'AND ONCE AGAIN DESTROY THE CONNECTION
connPubs.Close
Set connPubs = Nothing
```

Dynamic `Recordsets`

A dynamic `Recordset` object never uses a connection object and has nothing to do with any sort of external data storage.

To implement a dynamic `Recordset`, you never associate it with a `Connection` object. Instead, you dynamically define its structure by adding `Field` objects to its `Fields` collection. It is then up to the application (perhaps with the user's help) to populate the `Recordset` with rows and manipulate its data.

Your code should take the following steps to implement a dynamic `Recordset`:

1. Determine ahead of time the field structure that you want for each row of the dynamic `Recordset`.
2. Declare a `Recordset` object variable, but *don't* associate it with a `Connection`.
3. Repeat the following steps for each field that you want in the `Recordset`'s structure:
   - Call the `Append` method of the `Recordset` object's `Fields` collection to add the `Field` to the `Recordset`'s `Fields`.
   - In the first and second arguments of the `Append` method, set the `Field` object's `Name` and `Type` properties, respectively. If the `Field` has a type that can vary in size (such as `BSTR`, a Basic String type), also set its `AssignedSize` property in the third argument.
   - Repeat these steps for each `Field` that you want to have in the `Recordset`.
4. Call the `Recordset` object's `Open` method.
5. Load the `Recordset` with initial data through user edits and/or your own processing logic. Use the `Recordset` techniques discussed in this chapter.
6. Manipulate the data in the `Recordset` through user edits and/or your own processing logic. Again, use the `Recordset` techniques discussed in this chapter.

Listing 8.14 gives an example of code that implements a dynamic `Recordset` with offline processing.

LISTING 8.14

CODE THAT IMPLEMENTS A DYNAMIC RECORDSET

```
    Set rs = New ADODB.Recordset
    rs.CursorLocation = adUseClient
    rs.Fields.Append "FirstName", adBSTR, 25
    rs.Fields.Append "LastName", adBSTR, 25
    rs.Open
'After this point, programming is identical
    'to manipulating any other type of
    'ADO Recordset object
```

Persistent Recordsets

A persistent Recordset is a Recordset (typically a dynamic or disconnected Recordset as discussed in the previous two sections) whose information you save between sessions of your application into a local "holding file."

To implement a persistent Recordset, you call a Recordset object's Save method to save it to a file. When you want the saved data back, you call a Recordset object's Open method with an argument that indicates the name of the file where you previously saved Recordset data.

You need to take the following steps in your code to implement a persistent Recordset object:

1. Make sure you have an open Recordset object with which to work.
2. Call the Recordset object's Save method with two arguments:
   - The first argument represents the name of the file where you will save the Recordset object's data.
   - The second argument represents the data format that you will use to save data to the file. As of this writing, there's only one possible value for this argument, adPersistADTG.
3. When you want to retrieve previously saved information from a file into a Recordset, call the Recordset's Open method, passing to the Open method as its first and only argument the name of the file where you previously saved the Recordset information.

Listing 8.15 gives an example of code that implements a persistent disconnected Recordset.

LISTING 8.15

CODE THAT IMPLEMENTS A PERSISTENT RECORDSET

```
'Assumes that rs has been
'previously declared and initialized

Private Sub cmdSaveDisconn_Click()
    If Dir$("C:\trash\mydata.dat") <> "" Then
        Kill "C:\trash\mydata.dat"
    End If
    rs.Save "C:\Trash\MyData.dat", adPersistADTG
End Sub

Private Sub cmdRetrieveDisconn_Click()
    Set rs = Nothing
    Set rs = New ADODB.Recordset
    rs.Open "C:\Trash\MyData.dat"
End Sub
```

USING THE ADO ERRORS COLLECTION

▶ Use the ADO `Errors` collection to handle database errors.

The ADO `Errors` collection is a set of `Error` objects that contain information about the last ADO operation that caused a provider error.

The first thing to note about the `Errors` collection is that it does *not* contain information about *ADO* errors. Instead, the `Errors` collection contains information about *provider* errors. ADO errors are raised as normal runtime errors in the VB environment.

Assume, for example, that you attempt to assign an invalid cursor type to a `Recordset`'s `CursorType` property. Your attempt would generate a standard VB runtime error, because this is an ADO error.

On the other hand, if you call a SQL Server stored procedure and there is an abnormal termination in SQL Server, a provider error has occurred. The `Errors` collection will then get populated with one or more new entries.

Secondly, you should keep in mind that the `Errors` collection does *not* contain a *history* of errors generated during the current session. The `Errors` collection contains only the last set of `Error` objects generated by the provider when attempting to fulfill a single ADO request. ADO uses a collection to handle errors because a single request might generate more than one error message from the provider.

The `Errors` collection is not cleared with every new ADO request to the Provider. Instead, the `Errors` collection keeps information about the last request that caused errors even through subsequent ADO requests to the provider. The `Errors` collection is cleared only when a new set of errors occurs.

You can use the `Clear` method of the `Errors` collection to keep track of whether the most recent ADO operation has caused Provider errors.

CHAPTER SUMMARY

KEY TERMS

- ActiveX Data Objects
- Data consumer
- Data cursor
- Data provider
- Data Source Name
- DBMS
- Jet
- Open Database Connectivity
- Rowset
- SQL Server

This chapter covered the following topics:

- Programming `Connection`, `Command`, and `Recordset` objects with the Data Environment Designer
- Binding VB objects to Data Environment Objects
- Accessing ADO events for objects under a `Data Environment`
- Programming the ADO Data Control
- Adding, editing, and deleting records with the ADO Data Control
- Programming with the ADO object model
- Programming the `Connection` object
- Programming the `Command` object
- Programming `Recordset` methods and events
- Disconnected, Persistent, and Dynamic `Recordsets`
- The ADO `Errors` collection

APPLY YOUR KNOWLEDGE

Exercises

8.1 Programming the ADO Object Model

In this exercise, you perform common programming tasks with the main components of the ADO object model (`Connection`, `Command`, and `Recordset`). This exercise precedes exercises for more automated data manipulation techniques (Data Environment Designer and ADO Data Control) so that you will have a clearer understanding of what the automated tools do with the underlying ADO objects. Compare the data-access techniques of this exercise that directly manipulate the `Recordset` object's methods and properties with the techniques of the three data access models discussed in the next chapter and illustrated in exercises 9.3 and 9.4 of that chapter.

Estimated Time: 60 minutes

1. Start a new VB standard EXE project with a default startup form.
2. Add a reference in your project to the ADO 2.0 library by choosing Project, References from the VB menu and checking the Microsoft ActiveX Data Objects 2.0 Library in the list of Available References (refer back to Figure 8.24 in the section titled "Programming with ADO").
3. In the form's General Declarations, declare an ADO `Connection` object variable, an ADO `Command` object variable, and an ADO `Recordset` object variable. Implement event programming for the `Connection` and `Recordset` (remember that `Commands` don't support events: Try declaring the `Command` object using `WithEvents` to see the error message).

   Also declare a form-wide `Boolean` variable to act as a flag for an add action. This flag will be set to `True` if the user decides to add a record. You will see in subsequent steps how this flag is used.

   The code in General Declarations should look like this:

```
Option Explicit
Dim WithEvents cnNWind As ADODB.Connection
Dim WithEvents rsEmployees As ADODB.Recordset
Dim cmEmployees As ADODB.Command
Public gblnAddMode As Boolean
```

4. Add a `CommandButton` to the form and name it `cmdOpenConnection`, as shown in Figure 8.25.

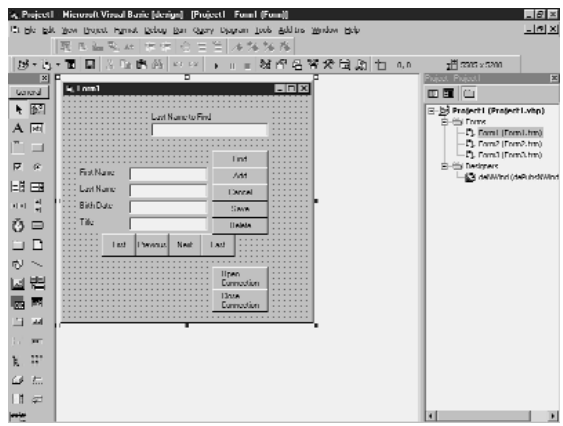


FIGURE 8.25
Completed form for Exercise 8.3.

   In the `CommandButton's Click` event procedure, instantiate and open the `Connection` and `Recordset` objects with code such as the following:

APPLY YOUR KNOWLEDGE

```
Private Sub cmdOpenConnection_Click()

    Set cnNWind = New ADODB.Connection
    Set rsEmployees = New ADODB.Recordset
    Dim sConnect As String
    sConnect = "Provider=Microsoft.Jet.
➥OLEDB.3.51;" & _
                "Data Source=
C:\DataSamples\Nwind.mdb"
    cnNWind.CursorLocation = adUseClient
    cnNWind.Open sConnect
    rsEmployees.CursorType = adOpenStatic
    rsEmployees.CursorLocation = adUseClient
    rsEmployees.LockType = adLockPessimistic
    rsEmployees.Source = "Select * From
➥Employees Order By LastName,FirstName"
    Set rsEmployees.ActiveConnection =
➥cnNWind
    rsEmployees.Open
End Sub
```

Remember that the provider and other elements of the `Connect` string will vary depending on your environment.

5. Add a second `CommandButton` and name it `cmdCloseConnection` (refer to Figure 8.25). Put code in its `Click` event procedure to close the `Recordset` and `Connection` (notice that you also set the form-wide flag for Adds to `False`, in case it had been turned on):

```
Private Sub cmdCloseConnection_Click()
    cnNWind.Close
    Set cnNWind = Nothing
    gblnAddMode = False
End Sub
```

6. Run the application to make sure that no errors occur when your code initializes and closes these objects.

7. Add four `Labels` and four `TextBoxes` to the form for data display as shown in Figure 8.25. Name the `TextBoxes` `txtEmployeeFirstName`, `txtEmployeeLastName`, `txtBirthDate`, and `txtTitle`. Give the `Labels` appropriate captions as shown in the figure.

8. Write a `Sub` Procedure called `ShowDataInControls`. This `Sub` will update the contents of the `TextBoxes` with the contents of the respective fields of the `Recordset`:

```
Private Sub ShowDataInControls()
    If rsEmployees.BOF Or rsEmployees.EOF
Then
        Exit Sub
    End If
    On Error Resume Next
    txtEmployeeFirstName = rsEmployees
➥("FirstName") & ""
    txtEmployeeLastName = rsEmployees
➥("LastName") & ""
    txtBirthDate = rsEmployees("BirthDate")
➥& ""
    txtTitle = rsEmployees("Title").Value
➥& ""
    If Err.Number <> 0 Then MsgBox
➥Err.Description
End Sub
```

9. Write a companion `Sub` procedure called `WriteDataFromControlsToBuffer`. This `Sub` will update the contents of the `Recordset`'s Title_ID and Title fields from the contents of the `TextBoxes`:

```
Private Sub WriteDataFromControlsToBuffer()
    If rsEmployees.BOF Or rsEmployees.EOF
Then
        Exit Sub
    End If
    On Error Resume Next
    rsEmployees("FirstName").Value =
➥txtEmployeeFirstName
    rsEmployees("LastName").Value =
➥txtEmployeeLastName
    rsEmployees("BirthDate").Value =
➥txtBirthDate
    rsEmployees("Title").Value = txtTitle
    If Err.Number <> 0 Then MsgBox
➥Err.Description
End Sub
```

Remember that the underlying data will not be updated until you also call the `Recordset`'s `Update` method.

APPLY YOUR KNOWLEDGE

10. Call the `RefreshControls` procedure in the `MoveComplete` event procedure of the `Recordset`:

```
Private Sub rsEmployees_MoveComplete( _
       ByVal adReason As
ADODB.EventReasonEnum,  _
       ByVal pError As ADODB.Error,  _
       adStatus As ADODB.EventStatusEnum,  _
       ByVal pRecordset As ADODB.Recordset)
    If Not gblnAddMode Then
        ShowDataInControls
    End If
End Sub
```

11. Place four navigation `CommandButtons` named `cmdNext`, `cmdPrevious`, `cmdFirst`, and `cmdLast` on the form as shown in Figure 8.25 and place code in their `Click` event procedures to navigate the `Recordset` as follows:

```
Private Sub cmdFirst_Click()
    On Error Resume Next
    rsEmployees.MoveFirst
End Sub
Private Sub cmdLast_Click()
    On Error Resume Next
    rsEmployees.MoveLast
End Sub
Private Sub cmdNext_Click()
    On Error Resume Next
    rsEmployees.MoveNext
End Sub
Private Sub cmdPrevious_Click()
    On Error Resume Next
    rsEmployees.MovePrevious
End Sub
```

12. Create a `Private Sub` procedure called `BlankControls` that will blank out the controls that hold data field contents:

```
Private Sub BlankControls()
    txtEmployeeFirstName = ""
    txtEmployeeLastName = ""
    txtBirthDate = ""
    txtTitle = ""
End Sub
```

13. Place a `CommandButton` named `cmdAdd` on the surface of the form as shown in Figure 8.25 and add the following code to its `Click` event procedure:

```
Private Sub cmdAdd_Click()
    BlankControls
    gblnAddMode = True
End Sub
```

Note that you don't actually call the `AddNew` method here, but merely blank out the contents of the controls and set a special flag variable to indicate that you are in the midst of an `Add` action. You will not call the `AddNew` method until users confirm that they definitely want to save changes.

14. Write a procedure to write the contents of the current controls to the fields of the record buffer and then update the underlying data with the contents of those fields:

```
Private Sub SaveRecord()

        On Error Resume Next

        If gblnAddMode Then
            rsEmployees.AddNew
        End If
        WriteDataFromControlsToBuffer
        rsEmployees.Update

gblnAddMode = False

End Sub
```

Note that we call the `Sub` procedure that will update field contents from this event procedure, just before calling the `Update` method. Also note the logic to check to see whether you are in the midst of an `Add` action. If so, you call the `AddNew` method.

APPLY YOUR KNOWLEDGE

15. Place a `CommandButton` named `cmdSave` on the surface of the form as shown in Figure 8.25 and add the following code to its `Click` event procedure to call the `SaveRecord` procedure:

```
Private Sub cmdSave_Click()
    SaveRecord

End Sub
```

16. Place a `CommandButton` named `cmdCancel` on the surface of the form as shown in Figure 8.25 and add the following code to its `Click` event procedure:

```
Private Sub cmdCancel_Click()
    ShowDataInControls
    gblnAddMode = False
End Sub
```

We just overwrite the user's changes by calling the routine to refresh `TextBox` contents with `Field` contents. You also turn off the `Add` flag, in case it had been set.

You need to call `CancelUpdate` if you were adding or editing (check `EditMode`). If you call `AddNew` a second time (if the user cancels, and then adds again), for example, you will get an error because you are already adding.

17. Place a `CommandButton` named `cmdDelete` on the surface of the form as shown in Figure 8.25 and add the following code to its `Click` event procedure:

```
Private Sub cmdDelete_Click()
    If (Not rsEmployees.EOF) And (Not
rsEmployees.BOF) Then
        cnNWind.BeginTrans
        rsEmployees.Delete
        cnNWind.CommitTrans
        rsEmployees.MoveNext
        If rsEmployees.EOF Then
            On Error Resume Next
            rsEmployees.MoveLast
            If Err.Number <> 0 Then
                'No more records left
                'perhaps take some action
➥here
            End If
            On Error GoTo 0
        End If
    End If
End Sub
```

18. Place a `TextBox` and accompanying `Label` on the surface of the form and name the `TextBox` `txtLastNameToFind` as shown in Figure 8.25. Place a button named `cmdFind` on the form as shown in the figure and place the following code in its `Click` event procedure:

```
Private Sub cmdFind_Click()
    Dim dBookmark As Double
    dBookmark = rsEmployees.Bookmark
    Dim sFindCriterion As String
    sFindCriterion = "LastName like '" & _
                     txtLastNameToFind & "'"
    rsEmployees.MoveFirst
    rsEmployees.Find sFindCriterion
    If rsEmployees.EOF Then
        rsEmployees.Bookmark = dBookmark
        MsgBox "Couldn't Find """ & _
                txtLastNameToFind & """
    Else
        txtLastNameToFind = ""
    End If
End Sub
```

Note that the `Bookmark` property works only with client-side cursors. Try changing the `CursorLocation` property of the `Recordset` to `adUseServer` and note the error message that appears when you run the application and attempt a `Find`.

19. You can test the application after each step above.

8.2 Binding Objects to Data with the Data Environment Designer

You program with a `Data Environment`'s `Connection` and `Command` objects, inserting bound controls on a form and programming the `Command` object's `Recordset`.

Estimated Time: 20 minutes

APPLY YOUR KNOWLEDGE

1. In a new VB project, choose the Project menu in VB. Make sure that Add Data Environment is one of the Project menu's options. If Add Data Environment does *not* appear on the Project menu, add it by performing the following steps:
   - Choose Project, Components and select the Designers tab on the Components dialog box (refer back to Figure 8.1 in the section titled "Managing ADO Objects with the Data Environment Designer").
   - In the Components dialog box, check the Data Environment box.
   - Click the OK button on the Components dialog box.
2. Choose Project, Add Data Environment from the VB menu.
3. Navigate to the Properties window of the `Data Environment` (see Figure 8.26).

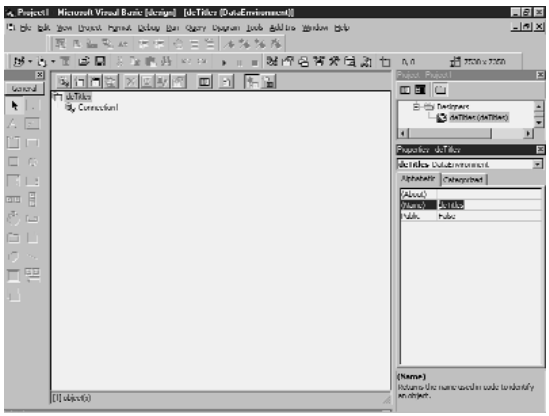


FIGURE 8.26▲
Changing the name of the `Data Environment`.

4. Change the `Name` property of the `Data Environment` object to `deTitles`.
5. Right-click on the `Data Environment`'s default `Connection` object, and then choose Properties from the drop-down menu.
6. On the Provider tab, choose Microsoft Jet 3.51 OLE DB as the OLE DB data provider (see Figure 8.27).

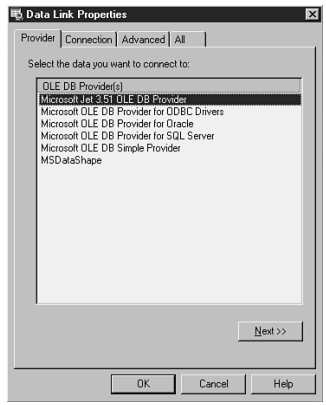


FIGURE 8.27▲
Specifying the connection's provider.

7. On `Connection` tab, set up the specific data connection with the following steps (note that the contents of the tab differ depending on the type of data provider selected on the `Connection` tab):
   - Set up the source of the data by specifying the NWIND.MDB file's name and path (see Figure 8.28).

APPLY YOUR KNOWLEDGE

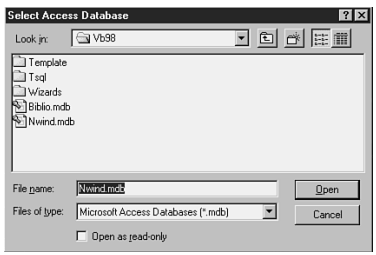


FIGURE 8.28▲
Setting NWIND.MDB as the data source for the `Connection` object.

- Fill in logon information about the username and password. (For an Access database, you typically leave these fields at their default values.)

8. Click OK to accept the data-source options you have built. You should now see a `Connection` object on the Data Environment Designer's surface.
9. Right-click the `Connection` object and choose Add Command from the drop-down menu, as in Figure 8.29.

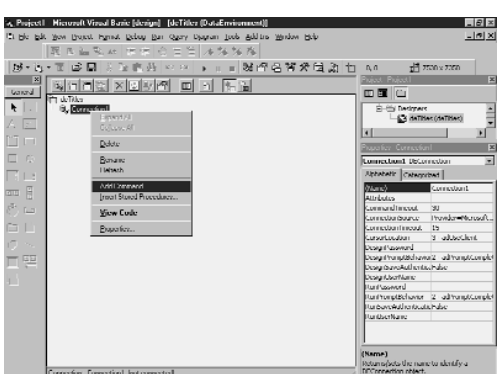


FIGURE 8.29▲
Adding a `Command` object to the `Connection`.

10. On the General tab (see Figure 8.30), set Database Object to `Table` in the drop-down list and set the Object Name to `Employees` from the drop-down list.

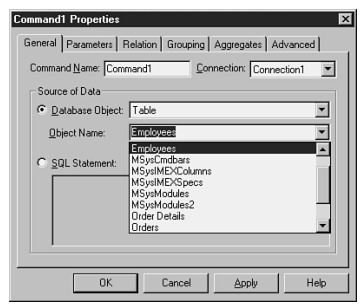


FIGURE 8.30▲
Setting general information for the `Command` object.

11. On the `Advanced` tab (see Figure 8.31), set cursor locking strategy to Optimistic. Click OK to apply changes and close the `Command1` Properties dialog box.

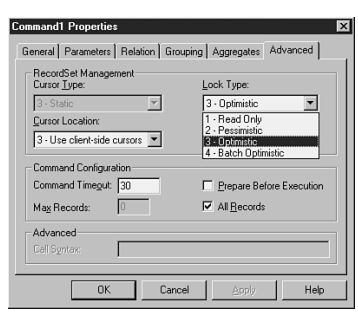


FIGURE 8.31▲
Setting cursor locking for the `Command` object.

APPLY YOUR KNOWLEDGE

12. Use the left (primary) mouse button to drag the newly created `Command` object from the Data Environment Designer onto the form, where it will automatically create bound fields for you, as shown in Figure 8.32.

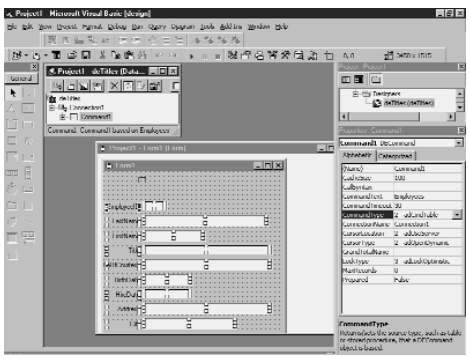


FIGURE 8.32▲
Automatically creating bound fields by dragging the `Command` object.

13. Test the bound controls by running the project, as shown in Figure 8.33.

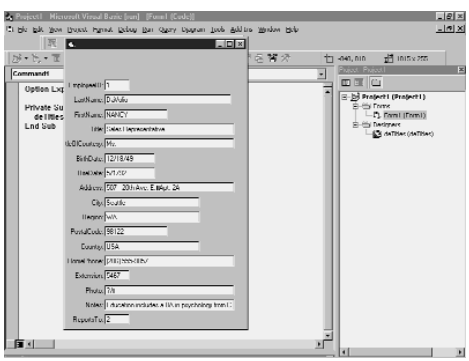


FIGURE 8.33▲
Testing the bound controls.

14. Notice that there is no built-in way to navigate the controls, as you have with the ADO Data Control. To remedy this, follow these steps:

    - Create a single `CommandButton` and name it `cmdNext` (see Figure 8.34).

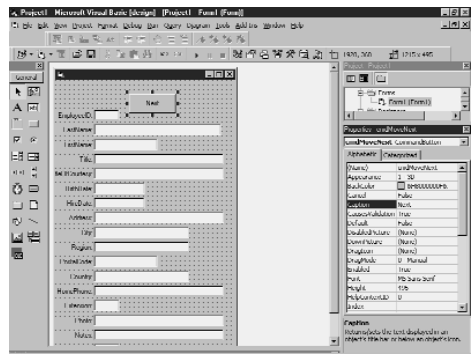


FIGURE 8.34▲
Form with a `MoveNext` command button.

In the `CommandButton`'s `Click` event procedure, put the following code:

```
Private Sub cmdNext_Click()
    deTitles.rsCommand1.MoveNext
    If deTitles.rsCommand1.EOF Then
        deTitles.rsCommand1.MoveLast
    End If
End Sub
```

Note again that this code is similar to code from straight ADO programming, with the exception of the need to refer to the `Recordset` through the `Data Environment`. Once again, the `Recordset` name is derived from the `Command` object's name.

- Create similar `CommandButtons` for the `MovePrevious`, `MoveLast`, and `MoveFirst` methods of the `Recordset`. You can use the model of step 10 in the preceding exercise to put code into their `Click` event procedures.

APPLY YOUR KNOWLEDGE

Just remember to add the `Data Environment`'s name to all references to the `Recordset` (as illustrated in the preceding step). Also, be sure to use the correct `Recordset` name.

8.3 Binding Objects to Data With the ADO Data Control

You create a small application that binds VB controls to the `Recordset` exposed by an ADO Data Control.

Estimated Time: 20 minutes

1. Start a new VB standard EXE project with a default form.
2. Add the ADO Data Control to your VB toolbox by calling the Project, Components menu option and then checking Microsoft ADO Data Control 6.0 (OLEDB) in the resulting dialog box. Click OK and you should see the ADO Data Control in your toolbox. Add an instance of the ADO Data Control to the form.
3. To build the connection string, open the ADO Data Control's Property Page by right-clicking on the ADO Data Control and selecting ADODC Properties. Now take the following steps:
   - The resulting dialog box has just one tab, General.

     Choose the Use Connection String option and click the Build button.
   - On the resulting Data Link Properties dialog box (see Figure 8.35), choose the Provider tab and choose Microsoft Jet 3.51 OLE DB Provider.

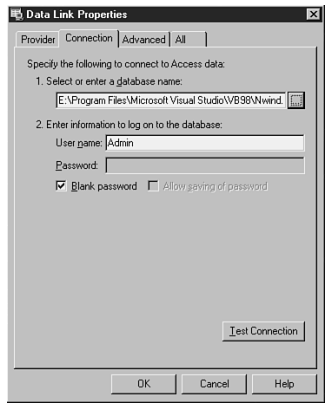


FIGURE 8.35▲
DataLink Properties dialog box for Build button of the `ConnectionString` property.

   - Select the Connection tab and click the ellipsis button (…) under Select or enter a database name. Find the NWIND.MDB database file on your system. (Figure 8.35 shows the completed information.)
   - Test the `Connection` to make sure that the connection works (see Figure 8.36).

FIGURE 8.36▲
Testing the `ConnectionString's` data link.

   - Click OK on both the dialog boxes to return to the ADO Data Control's Properties window.

APPLY YOUR KNOWLEDGE

4. Change the `Caption` property of the ADO Data Control to `Employees`.
5. Navigate to the `RecordSource` property and set it by choosing `Command Type 2 - adCmdTable` and the Employees table from the drop-down list of tables (see Figure 8.37).

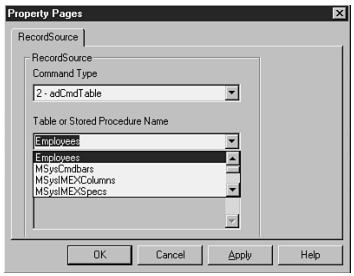


FIGURE 8.37▲
Setting the `Recordsource` property.

6. In the ADO Data Control Properties window, make sure that the `CursorType` property is set to `3-adOpenStatic`, the `CursorLocation` property to `3-adUseClient`, and the `LockType` property to `3-adLockOptimistic`.
7. Set the `EOFAction` property to `2-adDoAddNew`.
8. Add three `TextBox` controls and corresponding Labels to the form as shown in Figure 8.38. Name the `TextBox` controls `txtFirst`, `txtLast`, and `txtBirthDate`.

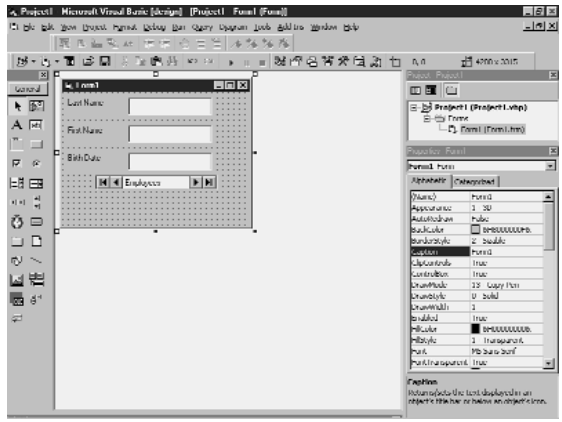


FIGURE 8.38▲
Data-bound `TextBox` controls and corresponding labels.

9. In their respective Properties windows, change the `DataSource` property of each `TextBox` control to point to the ADO Data Control (see Figure 8.39).

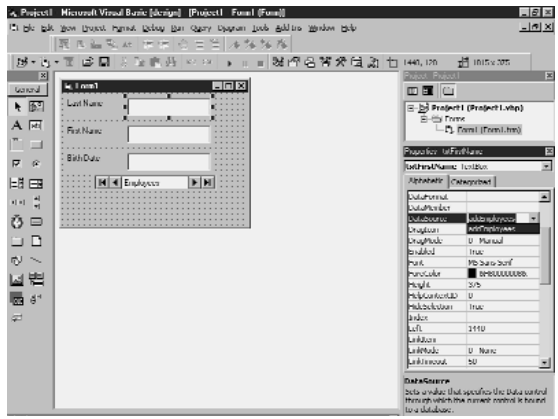


FIGURE 8.39▲
Setting the `DataSource` property of a `TextBox`.

APPLY YOUR KNOWLEDGE

For each `TextBox` in the Properties window, use the drop-down list of the `DataField` property to select the Employee table's `LastName`, `FirstName`, and `Birthdate` fields, respectively (see Figure 8.40).

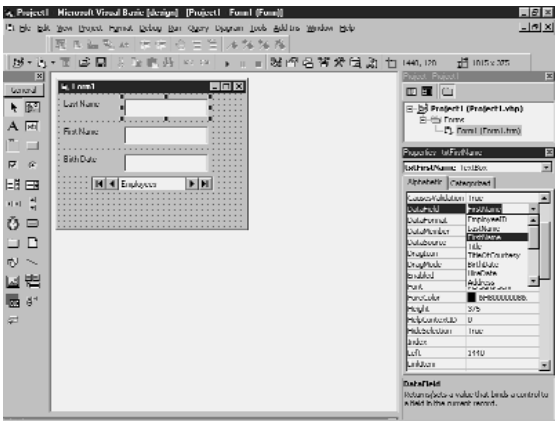


FIGURE 8.40
Setting the `DataField` property of a `TextBox`.

10. Run the application. The text boxes should be populated with field contents from the Employees table. Click the navigation buttons to verify that the text box contents change to reflect the different records. Attempt to navigate beyond the last record. Because of the setting of the `EOFAction` property, the fields should go blank and allow you to add a new record.

Review Questions

1. What are the two main objects that you can place on the surface of a Data Environment Designer?
2. Which ADO object is always associated with a cursor?
3. Which two ADO objects support events?
4. Which property of a `Recordset` object should you check to see if the preceding call to the `Find` method was successful?
5. How could you prevent one of ADO's `Will` event procedures from running multiple times during an application session?
6. What is the purpose of the ADO `Errors` collection?

Exam Questions

1. You create a `Connection` object named `connMy` and a `Command` object named `cmdMy` in a Data Environment Designer named `deMy`. In your code, you want to programmatically write changes to the underlying data. The line of code that you need to write would be:

   A. `deMy.rsconnMy.Update`

   B. `deMy.cmdMy.rs.Update`

   C. `deMy.connMy.rscmdMy.Update`

   D. `deMy.rscmdMy.Update`

2. How can you place a `DataGrid` on a form that is automatically bound to the `Recordset` of a `Data Environment`'s `Command` object? (Select all that apply.)

   A. Drag the `Command` object with the right mouse button from the `Data Environment` to the form.

APPLY YOUR KNOWLEDGE

B. Drag the `Command` object with the left mouse button from the `Data Environment` to the form.

C. Highlight the `Command` object with the mouse and then right-click the form.

D. Highlight the `Command` object with the mouse and then left-click the form.

3. The ADO Data Control's `RecordSource` property

A. Exposes a `Recordset`.

B. Contains the settings for creating a `Recordset`.

C. Exposes a `Fields` collection.

D. Exposes a `Command` object.

4. A VB control's `DataMember` property refers to

A. A `Recordset` object.

B. A `Connection` object.

C. A `Command` object.

D. A `Data Environment`.

5. You can set the contents of an ADO `Recordset` field named `ID` with the following code:

A. `sClients.ID.Value = strID`

B. `rsClients.Fields(ID).Value = strID`

C. `rsClients!Fields!(ID).Value = strID`

D. `rsClients!ID = strID`

6. The ADO `Errors` collection

A. Clears before every ADO action.

B. Contains a history of all errors during this session.

C. Contains the last errors generated by an ADO action.

D. Contains a lookup list of all possible ADO error codes.

7. Which code would appropriately display ADO errors (`connNwind` is an ADO `Connection` and `rsEmployees` is an ADO `Recordset` object)?

A.
```
Dim adoErr As ADODB.Error
For Each adoErr In connNWind.Errors
      MsgBox adoErr.Description
Next /
```

B.
```
Dim adoErr As ADODB.Error
For Each adoErr In rsEmployees.Errors
       MsgBox adoErr.Description
 Next adoErr
```

C.
```
Dim adoErr As ADODB.Error
For Each adoErr In connNWind
     MsgBox connNWind.Error.Description
Next adoErr
```

D.
```
Dim adoErr As Error
 For Each adoErr In connNWind.Errors
     MsgBox adoErr.Description
Next adoErr
```

8. You write the following code to search for a record in an ADO `Recordset`:

```
Dim sFindCriterion As String
sFindCriterion = "LastName like '" & _
txtLastNameToFind & "'"
rsEmployees.MoveFirst
rsEmployees.Find sFindCriterion
```

APPLY YOUR KNOWLEDGE

To test whether a record was encountered, you should write the following code:

A. `If rsEmployees.NoMatch Then`

B. `If rsEmployee.EOF Then`

C. `If rsEmployee.BookMark = 0 Then`

D. `If rsEmployee.BookMark = rsRmployee.RecordCount Then`

9. Setting the `adStatus` parameter of an ADO `Connection` object's `WillConnect` event procedure to a value of `adStatusUnwantedEvent`

   A. Causes the compiler to ignore this event procedure.

   B. Causes the event procedure to immediately terminate.

   C. Causes a runtime error if there is no error handler written in the event procedure.

   D. Causes the event procedure to run only once during the application session.

10. Assume that you have initialized an ADO `Connection` object named `cnNWind` and an ADO `Recordset` object name `rsEmployees` with the following code:

```
Set rsEmployees = New ADODB.Recordset
Set cnNWind = New ADODB.Connection
```

To prepare the `Recordset` for use within your program, you should write the following code:

A.
```
Dim sConnect As String
sConnect = "Provider=Microsoft.Jet.
➥OLEDB.3.51;" & _
"Data Source= NWind.mdb"
cnNWind.Open sConnect
rsEmployees.Open  _
"Select * From Employees Order By
EmployeeID", _
cnNWind
```

B.
```
Dim sConnect As String
sConnect = "Select * From Employees Order
➥By EmployeeID"
cnNWind.Open sConnect
rsEmployees.Open "Provider=Microsoft.Jet.
➥OLEDB.3.51;" & _
"Data Source= NWind.mdb" ,
cnNWind
```

C.
```
Dim sConnect As String
sConnect = "Select * From Employees Order
➥By EmployeeID"
cnNWind.Open sConnect
rsEmployees.Open cnNWind,
"Provider=Microsoft.Jet.OLEDB.3.51;" & _
"Data Source= NWind.mdb" ,
```

D.
```
Dim sConnect As String
sConnect = "Provider=Microsoft.Jet.
➥OLEDB.3.51;" & _
"Data Source= NWind.mdb"
cnNWind.Open sConnect
rsEmployees.Open  _
cnNWind,
"Select * From Employees Order By
➥EmployeeID"
```

11. Which of the following *must* you do at some point when programming with a disconnected `Recordset` that derives its records from a data source on the server? (Select all that apply.)

   A. Call `Fields.Append`.

   B. Set the `Recordset` object's `LockType` property to `adLockOptimistic`.

APPLY YOUR KNOWLEDGE

C. Set the `Recordset` object's `CursorLocation` property to `adUseClient`.

D. Set the `Recordset` object's `ActiveConnection` property to `Nothing`.

Answers to Review Questions

1. `Connection` and `Command` objects can be placed on the surface of a Data Environment Designer. See "Adding `Connection` and `Command` Objects With the Data Environment Designer."
2. A `Recordset` always requires an open cursor. See "Referring to `Recordset` Field Contents."
3. ADO `Connection` and `Recordset` objects are the only two ADO objects that support events. See "`Command` Object Events."
4. Check the `EOF` property of an ADO `Recordset` to see whether the result of the preceding `Find` method call was successful. Note that in the DAO object model, `Recordset` objects have a `NoMatch` property. ADO `Recordset` objects do not have the `NoMatch` property. See "Locating Records."
5. You could set the event's `adStatus` parameter to `adStatusUnwantedEvent`. See "The `WillConnect` Event."
6. The ADO `Errors` collection provides information on the most recent errors generated by an ADO action.

Answers to Exam Questions

1. **D**. `deMy.rscmdMy.Update` would call the `Update` method of the `Recordset` based on a `Data Environment`'s `Command` object (assuming the `Data Environment` were named `deMy` and the `Command` object were named `cmdMy`). Note that the `Recordset` object's name is automatically derived from the `Command` object's name by placing the letters `rs` before the `Command` object's name. For more information, see the section titled "Programming With a Data Environment Designer."
2. **A**. You can place a `DataGrid` on a form that is automatically bound to the `Recordset` of a `Data Environment`'s `Command` object by dragging the `Command` object with the right mouse button from the `Data Environment` to the form. None of the other options would work. For more information, see the section titled "Binding VB Objects to Data Environment Objects."
3. **B**. The ADO Data Control's `RecordSource` property contains the settings for creating a `Recordset`. The `Recordset` property actually exposes the `Recordset`. (The `Recordset` property is not available at design time.) For more information, see the section titled "Setting up the ADO Data Control."
4. **C**. A VB control's `DataMember` property refers to a `Command` object in a `Data Environment`. The `Datasource` property can refer to an ADO Data Control, a Data Environment, or one of VB's older Data Control types. For more information, see the section titled "Binding VB Objects to Data Environment Objects."

APPLY YOUR KNOWLEDGE

5. **D**. You can set the contents of a `Recordset`'s field with code such as `rsClients!ID = strID`. There are other syntactic possibilities, but none of the other choices get it right. Answer A would be correct if the first period were replaced with a bang (`!`). Answer B would be correct if the field name were in quotation marks. Answer C would be correct if the field name were in quotation marks and the bang (`"!"`) characters were replaced with periods. For more information, see the section titled "Referring to `Recordset` Field Contents."

6. **C.** The ADO `Errors` collection contains the last errors generated by an ADO action. It does not clear before every ADO action, but rather when a different error has occurred or when you call the `Clear` method of the `Errors` collection. For more information, see the section titled "Using the ADO `Errors` Collection."

7. **A**. The following code would be appropriate to display ADO errors (`connNwind` is an ADO `Connection` and `rsEmployees` is an ADO `Recordset` object):

```
Dim adoErr As ADODB.Error
For Each adoErr In connNWind.Errors
MsgBox adoErr.Description
Next adoErr
```

8. **B**.Use the `EOF` property to determine whether the result of an ADO `Recordset`'s `Find` method was successful. Answer A (using the `NoMatch` property) is somewhat of a trick for those familiar with the DAO object model: DAO `Recordsets` had a `NoMatch` property, but ADO `Recordset` objects do not have a `NoMatch` property. Answers C and D don't work because the `Bookmark` property gives no direct way of determining position in a `Recordset`. `Bookmark` is useful for saving and restoring the current row in a `Recordset`, but it doesn't give useful information if examined directly. For more information, see the section titled "Locating Records."

9. **D**. Setting the `adStatus` parameter of the `WillExecute` event procedure to a value of `adStatusUnwantedEvent` causes the event procedure to run only once during the application session. If the `Execute` action occurs subsequently in the same session, the event procedure will not run again. For more information, see the section titled "The `WillConnect` Event."

10. **A**.The code example in this choice illustrates the correct elements and syntax for initializing an ADO `Recordset`, given an existing ADO `Connection`: First, open the connection with an argument specifying the `Connection` string. The text for this argument should contain a reference to a valid OLE DB provider, and to a valid `DataSource`. Next, call the `Open` method of the `Recordset` object. You can specify its first two arguments as the data source (a SQL query) and the `Connection` object. The other choices for this question all violate these rules in one way or another. For more information, see the section titled "`Recordsets`." Note that an alternative solution (not listed in the choices for this question) would enable you to specify the `Recordset`'s `Source` and `ActiveConnection` properties before calling the `Open` method, which would then take no parameters.

APPLY YOUR KNOWLEDGE

11. **C, D**. When programming with a disconnected `Recordset` that derives data from a server-side database, you must set the `CursorLocation` property to `adUseClient` and you must set the `ActiveConnection` property to `Nothing`. Answer A is incorrect because you call `Fields.Append` only when you are dynamically defining the disconnected `Recordset` in your program (and not deriving it from a database). Answer B is incorrect because the `LockType` property must be set to `adLockBatchOptimistic`. For more information, see the section titled "Disconnected, Persistent, and Dynamic `Recordsets`" and the subsections under that section.

OBJECTIVES

This chapter helps you prepare for the exam by covering the following objectives and their subobjectives:

Access and manipulate data by using the Execute Direct model (70-175).

- *Execute Direct, Prepare/Execute, and Stored Procedures models*: These three main objectives and the two subobjectives under the Stored Procedure model concentrate on three different *data-access models*—ADO techniques for getting a data server to manipulate data or return it to your application (Execute Direct, Prepare/Execute, and Stored Procedures models). The Execute Direct model causes your application to request the server to immediately execute a query whose text your application passes to the database engine.

Access and manipulate data by using the Prepare/Execute model (70-175).

- The Prepare/Execute Direct model causes your application to request the database server to precompile your query to a temporary stored procedure that will be available during the current session.

Access and manipulate data by using the Stored Procedures model (70-175).

- **Use a stored procedure to execute a statement on a database.**
- **Use a stored procedure to return records to a Visual Basic application.**

- The Stored Procedures model causes your application to request the server to execute an existing, permanent stored procedure.
- The two subobjectives for stored procedures require you to know what a stored procedure is in a relational database and how to create stored procedures in SQL Server, as well as how to program with SQL Server stored procedures in ADO.

CHAPTER 9

Creating Data Services: Part II

Retrieve and manipulate data by using different cursor locations. Cursor locations include client side and server side (70-175).

- *Location and type*. The next two objectives reflect ADO's capability to give you control over a data *cursor*, which is essentially a software object that provides a connection and a pointer to a particular row in a data set.
- *Cursor location* refers to where the resources for a cursor are allocated—either on the client machine or on the server machine.

Retrieve and manipulate data by using different cursor types. Cursor types include `ForwardOnly`, `Static`, `Dynamic`, and `Keyset` (70-175).

- *Cursor type* refers to the type of behavior available from a cursor. You can balance richness of available features against resource usage when you choose cursor type.

Manage database transactions to ensure data consistency and recoverability (70-175).

- *Transactions*. The next objective requires you to have a knowledge of the general nature of a *database transaction* (a group of actions on data that must stand or fall together). You must also be able to implement database transactions in VB ADO programming.

Write SQL statements that retrieve and modify data (70-175).

- *SQL statements to retrieve, modify, and join data*. The two objectives that deal with SQL knowledge imply that you will need a basic familiarity with the major types of statements in Structured Query Language (SQL) for obtaining and manipulating data. SQL is the basis for stored procedures, so these objectives tie in with the previously mentioned objectives for stored procedure. SQL is also the basis for stored procedures and for other data manipulation techniques such as those provided by the Execute Direct and Prepare/Execute models mentioned in previous objectives.

Write SQL statements that use joins to combine data from multiple tables (70-175).

- This objective requires you to know how inner and outer joins function in SQL. Joins in SQL enable you to present the information from more than one table in the results of a single SQL query.

Use appropriate locking strategies to ensure data integrity. Locking strategies include Read-Only, Pessimistic, Optimistic, and Batch Optimistic (70-175).

- *Locking strategies*. The final objective addresses your ability to manage database *concurrency conflicts* (the problems caused when more than one user or process tries to write to the same data at the same time) with various strategies for *locking*. Locking is the name for the methods and techniques that you can use as a database programmer to stop other users and processes from writing (or perhaps even viewing) selected data while your process updates that data. In ADO, you implement locking by setting the `LockType` property of a `Recordset` object.

OUTLINE

ADO Data-Access Models **378**

Accessing Data With the Execute Direct Model 379

Accessing Data With the Prepare/Execute Mode 380

Accessing Data With the Stored Procedures Model 382

How to Choose a Data-Access Model 383

Using Stored Procedures **384**

Creating Stored Procedures 385

Using the `Parameters` Collection to Manipulate and Evaluate Parameters for Stored Procedures 388

Using Stored Procedures to Execute Statements on a Database 390

Using Stored Procedures to Return Records to an Application 396

Using Cursors **400**

Using Cursor Locations 400

Using Cursor Types 402

Managing Database Transactions **404**

Writing SQL Statements **408**

Writing SQL Statements That Retrieve and Modify Data 409

Writing SQL Statements That Use Joins to Combine Data from Multiple Tables 411

Using Locking Strategies to Ensure Data Integrity **413**

Choosing Cursor Options **414**

Chapter Summary **416**

STUDY STRATEGIES

- Become familiar with the Execute Direct and Prepare/Execute models by reading the section titled "ADO Data-Access Models" and working through Exercise 9.3.
- Make sure that you are familiar with enough SQL to be able to write simple `Select` queries and data-manipulation statements as discussed in the sections titled "Using Stored Procedures to Execute Statements on a Database," "Using Stored Procedures to Return Records to an Application," and "Writing SQL Statements." Also become familiar with the SQL statements found in the queries of Exercise 9.1 and the stored procedures discussed in Exercise 9.2.
- If you have access to SQL Server (it is included with the VB Enterprise Edition), become familiar with the SQL Server Enterprise Manager utility. In the sample `pubs` database, create and run stored procedures as discussed in the section "Using Stored Procedures" (and its subsections) and as illustrated in Exercise 9.2.
- Experiment with calling stored procedures from VB ADO as discussed in sections under "Using Stored Procedures" and in Exercise 9.4.
- Experiment with cursors and their locations and types, as well as locking strategies with cursors. These topics are discussed in the sections "Using Cursors," "Using Locking Strategies to Ensure Data Integrity," and "Choosing Cursor Options," and are illustrated in Exercise 9.6.
- Become familiar with database transactions and how to set them up in VB ADO, as discussed in the sections on database transactions and Exercise 9.6.

INTRODUCTION

This is the second chapter that discusses Microsoft's VB exam objectives for data access. The preceding chapter—Chapter 8, "Creating Data Services: Part I"—introduced ADO concepts and discussed general ADO programming techniques.

This chapter focuses on some of the more specialized concerns with ADO programming, including various data-access models, the use of SQL and stored procedures, record locking, and database transactions.

ADO DATA-ACCESS MODELS

ADO supports three *data-access models*. The term "data-access model" refers to the choice of technique that you use in your program to request the data provider to return data to a `Recordset` or to execute some action on the data.

Following are brief descriptions of each of the three data-access models covered in the certification exam:

- The *Execute Direct* model enables you to dynamically specify a SQL data-access statement every time you access data. The underlying provider then executes your statement to return the requested result or manipulate the data.
- The *Prepare/Execute* model also enables you to dynamically specify a data-access statement. The first time that you run the statement, the underlying provider compiles your statement as a temporary stored procedure (whose lifetime lasts as long as its associated `Connection`). The provider then executes the temporary stored procedure. On subsequent requests to run the same statement during the lifetime of the `Connection`, the provider runs the temporary stored procedure.
- The *Stored Procedures* model requires you to use an existing stored procedure that already belongs to the data that you are accessing.

The following sections discuss each of the three models in greater detail, as well as the reasons you might have for choosing a particular data-access model over the other two.

Accessing Data With the Execute Direct Model

- Access and manipulate data by using the Execute Direct model.

The Execute Direct model assumes the following steps:

1. You make an on-the-fly request to the provider.
2. The provider interprets your request.
3. The provider executes the interpreted request.
4. The provider returns the result to you.
5. When you make the same request in the future, the provider reruns steps 2–5 again, re-interpreting the request each time.

In ADO, you can use the Execute Direct model to implement a request to a data provider in one of several ways:

- Call a `Connection` object's `Execute` method with a single argument that is the text of the SQL statement that you want the provider to execute.
- Call a `Command` object's `Execute` method with the text of the SQL statement in the `Command` object's `CommandText` property.
- Call a `Recordset` object's `Open` method with the text of the SQL statement.

Listing 9.1 shows examples of these different types of Execute Direct calls.

LISTING 9.1

EXAMPLES OF THE EXECUTE DIRECT MODEL

```
'INITIALIZING THE RECORDSET WITH THE
'EXECUTE METHOD OF A CONNECTION OBJECT
    'Assumes connNWind was already initialized
    'as an ADODB.Connection object
    Dim rsEmployees As ADODB.Recordset
    Dim sExecuteString As String
    sExecuteString = "SELECT * FROM employees " & _
        "WHERE LastName = '" & _
        txtLastName & "'" & _
        "AND FirstName = '" & _
```

```
        txtFirstName & "'"
    Set rsEmployees = connNWind.Execute(sExecuteString)


    sExecuteString = "SELECT * FROM employees " & _
        "WHERE LastName LIKE '" & _
        txtLastName & "%'" & _
        "AND FirstName LIKE '" & _
    txtFirstName & "%'"

'INITIALIZING THE RECORDSET WITH THE
'EXECUTE METHOD OF A COMMAND OBJECT
    Dim comNWind As ADODB.Command
    Set comNWind = New ADODB.Command
    With comNWind
        Set .ActiveConnection = connNWind
        .CommandType = adCmdText
        .CommandText = sExecuteString
        .Prepared = True
        Set rsEmployees = .Execute(sExecuteString)
    End With

'INITIALIZEING THE RECORDSET WITH
'ITS OWN OPEN METHOD
    Set rs = New ADODB.Recordset
    With rs
        .CursorLocation = adUseClient
        .LockType = adLockBatchOptimistic
        .Source = "Select * from employees"
        Set .ActiveConnection = connNWind
        .Open
    End With
```

Accessing Data With the Prepare/Execute Model

- Access and manipulate data by using the Prepare/Execute model.

The Prepare/Execute model assumes the following steps:

1. You make an on-the-fly request to the provider, instructing it to compile your request.
2. The provider compiles your request.
3. The provider executes the compiled request.

4. The provider returns the result to you.
5. When you make the same request in the future during this session, the provider repeats steps 3 and 4, using the already compiled statement.

You can make a request using the Prepare/Execute model by using a `Command` object in the following steps:

STEP BY STEP

9.1 Making a Request Using the Prepare/Execute Model

1. Set the `Command` object's `Prepared` property to `True`.
2. Set the `Command` object's `CommandType` property to `adCmdText` if you are executing a SQL statement, or `adCmdTable` if the request is just a table name (or skip this step and use the `Options` parameter in step 4).
3. Set the `Command` object's `CommandText` property to the text of the SQL statement that you want to execute (or skip this step and pass the SQL statement as an argument to the `Execute` method in the following step).
4. Call the `Command` object's `Execute` method. If you did not set the `CommandType` property in step 2, pass the appropriate value as the `CommandType` argument. If you did not set the `CommandText` property in step 3, pass the SQL statement as the `Options` argument (the third argument) to the `Execute` method.

The `Prepared` property of a `Command` object makes the difference between the Prepare/Execute and the Execute Direct models. When the `Prepared` property is set to `True`, the provider compiles the SQL statement to a temporary stored procedure before running it. The provider persists this temporary stored procedure until the `Command` object is changed or destroyed.

Accessing Data With the Stored Procedures Model

- Access and manipulate data by using the Stored Procedures model.

The Stored Procedures model assumes the following steps:

1. You request the provider to run an already compiled stored procedure.
2. The compiler runs the stored procedure.
3. The compiler returns the results of the stored procedure to you.

You can implement a Stored Procedures data-access model in one of several ways:

- Call a `Connection` object's `Execute` method with a single argument that is the name of the stored procedure that you want the provider to execute. The `Connection` object's `Options` argument must be set to `adCmdStoredProc`.
- Call a `Recordset` object's `Open` method with the name of the stored procedure. The `Recordset`'s `Options` argument must be set to `adCmdStoredProc`.
- Call a `Command` object's `Execute` method after setting the `Command` object's `CommandText` property to the name of the stored procedure. Either the `Execute` method's `Options` argument (third argument) must be `adCmdStoredProc`, or the `Command` object's `CommandType` property must be `adCmdStoredProc`. You may also have to set the `Command` object's `Parameters` collection if the stored procedure requires or returns parameters.

Listing 9.2 gives an example of a stored procedure in a SQL Server database, and Listing 9.3 gives examples of the Stored Procedures data-access model in VB.

LISTING 9.2

A Stored Procedure in SQL Server

```
create procedure Titles_All
AS
        Select * from Titles Order By title
GO
```

LISTING 9.3

Examples of the Stored Procedures Model in VB

```
cmdPubs.CommandText = "Titles_All"
cmdPubs.CommandType = adCmdStoredProc

Set rsPubs = cmdPubs.Execute
```

NOTE

Stored Procedures and Parameters
The information under the section "Using Stored Procedures" contains more information on how to program with stored procedures in the ADO object model.

Also see the section "Using the Parameters Collection to Access Parameters for Stored Procedures" for more information on how to work with a `Command` object's `Parameters` collection.

How to Choose a Data-Access Model

Each of the three data-access models discussed in the previous sections has advantages and disadvantages. Which data-access model you choose for a particular task depends on the requirements of the situation in which you are manipulating data.

Following are the considerations that you should bear in mind when choosing one data-access model over the others:

- The Execute Direct data-access model is best suited for one-time data-access statements. This is because the Execute Direct model executes more efficiently than the Prepare/Execute model *the first time* that you run a statement. The Execute Direct model is more efficient than the Prepare/Execute model the first time it is run, because the Prepare/Execute model requires the provider to compile the statement the first time that it runs. So, the first time through, Execute Direct runs faster.

Examples of one-time data-access statements that you could use with the Execute Direct model might include SQL statements entered on-the-fly by users (it would be impossible for the programmer to know these in advance) or statements formed dynamically by your program.

- The Prepare/Execute model is best suited for data-access statements that you will use more than once during the same program session, but that won't be used outside of the program session. As just stated, statements that are used only one time should be executed with the Execute Direct model, because they will run faster the first time than statements executed with Prepare/Execute. On subsequent executions, however, the Prepare/Execute model will be more efficient.

  Statements formed on-the-fly by your application to be used more than one time during the current session are good candidates for the Prepare/Execute model. If the statement's contents can be known between sessions and are valid for more than one user, it is better to use stored procedures.

- The Stored Procedures model is best suited for statements that will be in permanent use. Obviously, one-time-only statements won't work with stored procedures, because you can't know them ahead of time.

Using Stored Procedures

Stored procedures exist outside of your application in the database exposed by the provider.

At their simplest, stored procedures are precompiled SQL `Select` statements.

In most modern DBMSs, however, stored procedures have additional capabilities beyond just basic SQL syntax. These additional capabilities allow stored procedures to receive and return parameters. For DBMSs such as Microsoft's SQL Server, stored procedures can operate as functions with their own return values, as well as accept and return parameters.

Stored procedures can even behave as programming routines in their own right, permitting flow-of-control constructs such as looping and branching, and allowing temporary storage of intermediate information in declared, typed variables.

Stored procedures have several advantages over on-the-fly SQL statements:

- **Stored procedures are efficient**. A stored procedure is compiled when it is created and is stored and invoked as a compiled routine. Therefore, when your application causes the provider to run a stored procedure, the stored procedure will run more efficiently than a dynamically created SQL statement, which must be compiled by the provider before it can run. Also, the stored procedure will be executed server side, and not with the client workstation's resources.
- **Stored procedures enforce standards**. Because a stored procedure exists in the database, it can be maintained by the appropriate administrators who can change it to keep up with changing requirements. Stored procedures are therefore one way to enforce middle-tier business rules throughout a database. For instance, a stored procedure named "Compute_Commission" might use a different algorithm this year from the algorithm that it used last year.
- **Stored procedures make programming simpler**. Because a stored procedure moves processing and logic to the middle tier of a client/server system, there is just that much less processing and logic to perform in the user-interface or application tier. After it has been written, tested, and established in a database, a stored procedure can be used as a "black box" component of an application.

The following sections discuss how to create stored procedures.

Creating Stored Procedures

You can create a stored procedure in SQL Server with a special `CREATE PROCEDURE` query. The `CREATE PROCEDURE` query uses the following format:

```
CREATE PROCEDURE ProcName As ProcText
```

NOTE

More Information on Creating Stored Procedures in SQL Server See Exercise 9.2 for step-by-step, illustrated information on how to create stored procedures in Microsoft's SQL Server Enterprise Manager.

where `ProcName` is the name that you will give to the stored procedure and `ProcText` is one or more lines of SQL syntax.

Listing 9.4 gives an example of the creation of a simple stored procedure that contains a `Select` statement to retrieve records from a single table.

LISTING 9.4

SQL Server Query to Create a Simple Stored Procedure

```
CREATE PROCEDURE Employee_All As
Select * from Employee
```

A stored procedure is usually more complicated than the simple example of Listing 9.4. In SQL Server, for example, a stored procedure can use flow-of-control keywords (such as `if` and `while`), support local memory variables, receive and modify parameters, use temporary tables, and return a value to the requester as if the stored procedure were a function.

The syntax of a SQL Server stored procedure's execution language should look very familiar to a Visual Basic programmer, although it is not exactly the same. Listing 9.5 gives an example of a more complex stored procedure. Note the use of local variables, parameters, a return value, and nonexecutable comment lines. It is also possible to use flow-of-control constructs such as `if` and `while`.

LISTING 9.5

SQL Server Query to Create a Complex Stored Procedure

```
—local variables
declare @chTaxRegNbr char(10)
declare @dtToday datetime
declare @MaxDPD int

—Get TaxRegNbr for this customer
select @chTaxRegNbr = taxregnbr
    from customer
    where custid = @parmCustID
```

```
—Initialize variable for Max DPD & current date
select @MaxDPD = 0
select @dtToday = GetDate()

—Open table that holds CUSTIDS from CUSTOMER
—for records with same TAXREGNBR
select custid into #tempCustID_TaxReg from customer
       where taxregnbr = @chtaxregNbr
       order by CustID

—get max DPD of any invoice
select @MaxDPD = MAX(datediff(day,duedate,@dtToday)) from
➥ardoc
        where custid in (select custid from
#tempCustID_TaxReg)
                AND DocType IN ('IN','DM','FI')
                AND Rlsed = 1
                AND OpenDoc = 1
                and duedate > 'JAN 01, 1900'

—destroy the temporary table
drop table #tempcustid_taxreg

—Make sure intermediate result is 0 if no records found
select @MaxDPD = ISNull(@MaxDPD,0)

—return final value
return @MaxDPD

GO
```

Of particular interest for the following discussion are the parameters and the return value of a stored procedure. The parameters in a SQL Server stored procedure are declared immediately after the procedure name in the `CREATE PROCEDURE` query and before the `As` keyword.

On the other hand, the return value of a stored procedure is passed back to the requester by placing the keyword `return` in front of the value to be returned.

Refer again to Listing 9.5 for examples of parameters and a `return` statement. Note that parameters are declared using C-style syntax, as are local variables:

```
@parm_name datatype
```

where `datatype` is an appropriate SQL Server data type.

Parameters can be used to pass information back to the requester. These parameters are called *output parameters* and are designated in the parameter's declaration with the keyword `output` after the datatype.

Multiple parameter declarations are separated by commas, as illustrated in the listing.

Using the `Parameters` Collection to Manipulate and Evaluate Parameters for Stored Procedures

The `Parameters` collection of a `Command` object represents the parameters passed between the requester and a stored procedure. If the stored procedure returns a value, the `Parameters` collection also contains an additional member (element `0`) that holds the return value.

Before you can call a stored procedure that uses parameters, you must add a `Parameter` object to the `Parameters` collection for each parameter that the stored procedure uses. There are two methods for populating the `Parameters` collection with a stored procedure's parameters:

- Explicitly create a `Parameter` object in code and append it to the `Command` object's `Parameters` collection for each parameter that the stored procedure uses. Listing 9.6 gives an example of this technique.

LISTING 9.6

POPULATING THE `Parameters` COLLECTION EXPLICITLY

```
Dim cmdPubs As ADODB.Command
Set cmdPubs = New ADODB.Command
Set cmdPubs.ActiveConnection = connPubs

cmdPubs.CommandText = "Publishers_All"
cmdPubs.CommandType = adCmdStoredProc


Dim param As ADODB.Parameter

Set param = cmdPubs.CreateParameter("Return",  _
                    adInteger,  _
                    adParamReturnValue, , 0)
cmdPubs.Parameters.Append param

Set rsPubs = cmdPubs.Execute
```

- Automatically populate the `Parameters` collection with the appropriate members by calling the `Refresh` method of the `Command` object's `Parameters` collection. You must, of course, do this only after you set the `Command` object's `CommandType` and `CommandText` properties for the stored procedure that you want to call. Listing 9.7 gives an example of this technique.

LISTING 9.7

POPULATING THE `Parameters` COLLECTION AUTOMATICALLY WITH THE `Command` OBJECT'S `Refresh` METHOD

```
Dim cmdPubs As ADODB.Command
Set cmdPubs = New ADODB.Command
Set cmdPubs.ActiveConnection = connPubs
With cmdPubs
    .CommandType = adCmdStoredProc
    .CommandText = "Update_Titles_Title"

    'Refresh Parameters collection
    .Parameters.Refresh

    'and then set properties of each parameter:

    '@id parameter
    .Parameters("@id").Value = txtUpdateTitleID
    .Parameters("@id").Direction = adParamInput

    '@title parameter
    .Parameters("@title").Value = txtUpdateTitleName
    .Parameters("@title").Direction = adParamInput

    .Execute
End With
```

After you have populated the `Parameters` collection and called the stored procedure with the `Command` object's `Execute` method, you can then check the values of any `Output` parameters by checking the `Value` property of the appropriate `Parameter` objects.

If the stored procedure furnishes a return value, the return value will appear in element `0` of the `Parameters` collection, as illustrated in Listing 9.8.

LISTING 9.8

CALLING A STORED PROCEDURE THAT FURNISHES AN OUTPUT PARAMETER AND A RETURN VALUE

```
Dim cmdPubs As ADODB.Command
Set cmdPubs = New ADODB.Command
Set cmdPubs.ActiveConnection = connPubs

cmdPubs.CommandText = "Count_Titles_For_PubID"

With cmdPubs.Parameters
    'append directly to parameters collection, using
    'the return value of the CreateParameter method

     .Append cmdPubs.CreateParameter("@PubID", _
                                     adVarChar, _
                                  adParamInput, _
                                             4, _
                     RTrim(txtCountPubID.Text))

    .Append cmdPubs.CreateParameter("@NumTitles", _
                                        adInteger, _
                                   adParamOutput, _
                                                , _
                                                0)

End With

cmdPubs.Execute
lblCountTitlesPublisher =  _
           cmdPubs.Parameters("@NumTitles").Value
blnSuccess =  _
           cmdPubs.Parameters(0).Value
```

Using Stored Procedures to Execute Statements on a Database

You can create stored procedures to do most data maintenance chores such as adding and deleting records or updating individual fields in existing records.

Each of these types of stored procedure is based on a type of SQL statement, as described in the following list:

- `INSERT` statements in SQL add new records and populate their fields with initial values at the same time.
- `DELETE` statements in SQL delete existing records.
- `UPDATE` statement change the values of individual fields in existing records.

The following sections describe each type of SQL statement in more detail and explain how to use stored procedures based on each type of SQL statement to implement cursorless processes from ADO.

`INSERT` Statements in SQL

The SQL `INSERT` statement adds new records to a table. The `INSERT` statement has this general format:

```
INSERT tablename
    [(field list...)]
VALUES
    (value list...)
```

where `tablename` is the name of a table in the current database, `field list` is a comma-separated list of field names in the table, and `value list` is a comma-separated list of values to assign to each field. The entries in the `value list` must match up in order, number, and type with the entries in the `field list`.

Note that the field list is optional. If you leave it out of the `INSERT` statement, however, you must supply a value for every field in the table in the value list, and the values must be listed in the same order as the fields are listed in the original table structure.

A simple example of an `INSERT` statement might be this:

```
INSERT employees
(LastName, FirstName, HireDate)
VALUES
("Brunner", "Melanie", #7/15/98#)
```

This would insert a record for Melanie Brunner with a hire date of July 15, 1998 into the `employee` table.

Listing 9.9 shows an example of a stored procedure that uses an `INSERT` statement.

LISTING 9.9

A Stored Procedure Based on an Insert Statement

```
create procedure insert_titles
        @id varchar(6),
        @title varchar(80),
        @PubID varchar(4)
AS
        INSERT titles
        (title_id, title, Pub_ID)
        VALUES
        (@id, @title, @PubID)
GO
```

UPDATE Statements in SQL

The SQL UPDATE statement changes the values in one or more fields in designated rows in a table:

```
UPDATE tablename
SET fieldname = expression[,...]
[WHERE condition]
```

where tablename is a valid table name for the current database, fieldname is a valid field name in that table, expression is a valid expression that gives a value appropriate for the field, and condition is an expression to filter rows. As the ellipses imply, you can list modification statements for more than one field. Just separate each modification clause from the others with commas.

A simple example of an UPDATE statement is this:

```
UPDATE employees
SET salary = salary * 1.05
WHERE employeeid = 432
```

This would give Employee #432 a five percent raise.

Note that the WHERE clause is optional (as it always is in SQL). If you leave the WHERE clause out of an UPDATE statement, you will update the designated fields in all the rows in the table.

Listing 9.10 provides an example of a stored procedure based on an UPDATE statement.

LISTING 9.10

A STORED PROCEDURE BASED ON AN UPDATE STATEMENT

```
create procedure update_titles_title
        @id varchar(6),
        @title varchar(80)
AS
        UPDATE titles SET title = @title WHERE title_id = @id
GO
```

DELETE Statements in SQL

The SQL DELETE statement removes rows from a table.

This statement has this general format:

```
DELETE tablename
[WHERE condition]
```

where tablename is a valid table name from the current database, and condition is any valid record selection criterion. Note that the WHERE clause is optional (as it always is in SQL). If you leave the WHERE clause out of a DELETE statement, you will delete all the rows in the table.

A simple example of a DELETE statement is this:

```
DELETE from employee where employeeid = 231
```

This would remove the record for Employee #231 from the table.

Listing 9.11 shows an example of a stored procedure created from a DELETE statement.

LISTING 9.11

A STORED PROCEDURE BASED ON A DELETE STATEMENT

```
create procedure delete_titles_by_id
        @id varchar(6)
AS
        DELETE titles
        WHERE title_id = @id
GO
```

Using Stored Procedures to Execute Processes Without Cursors

The ADO `Recordset` object is versatile enough that you can perform any needed data maintenance action by programming its object model.

You can forego a `Recordset` in your code for many routine data maintenance activities, however, and instead use the `Execute` method of a `Connection` or `Command` object to make calls to SQL statements or stored procedures that implement those SQL statements. Here are the main areas where you could make such substitutions:

- **Deleting records**. Instead of calling the `Recordset`'s `Delete` method, execute a stored procedure that uses the SQL `DELETE` statement.
- **Adding records**. Instead of using the `AddNew` and `Update` methods of the `Recordset`, execute a stored procedure that uses the SQL `INSERT` statement.
- **Updating existing records**. Instead of changing fields in a `Do...Loop` through the `Recordset` and then calling the `Update` method, execute a stored procedure that uses the SQL `UPDATE` statement.

Listing 9.12 shows ADO code that uses `Recordset` manipulation (and therefore cursors) to add, delete, and modify records.

Listing 9.13 shows ADO code that calls the `Execute` method of `Command` or `Connection` objects (which are cursorless) to accomplish the same tasks with stored procedures. Listing 9.13 gives the texts of the stored procedure creation statements as they would appear in SQL Server.

LISTING 9.12

EXAMPLE OF A PROCESS THAT USES A CURSOR

```
Private Sub cmdInsert_Click()
    Set rsPubs = cmdPubs.Execute
    RsPubs.Fields("ID").Value = txtTitleID
    RsPubs.Fields("Title").Value = txtTitle
    RsPubs.Update
End Sub

x
y
```

LISTING 9.13

Examples of the Same Action of Listing 9.12, but Ran Without a Cursor

```
Private Sub cmdInsert_Click()
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection = connPubs
    With cmdPubs
        .CommandType = adCmdStoredProc
        .CommandText = "Insert_Titles"

        'Refresh Parameters collection
        .Parameters.Refresh

        'and then set properties of each parameter:

        '@id parameter
        .Parameters("@id").Value = txtTitleID
        .Parameters("@id").Direction = adParamInput

        '@title parameter
        .Parameters("@title").Value = txtTitle
        .Parameters("@title").Direction = adParamInput

        '@output parameter
        .Parameters("@PubID").Value = txtInsertPubID
        .Parameters("@PubID").Direction = adParamInput


        .Execute
    End With
    lblResults = ""
    lstResults.Clear
End Sub
```

LISTING 9.14

Stored Procedure Used by the Example of Listing 9.13

```
create procedure insert_titles
        @id varchar(6),
        @title varchar(80),
        @PubID varchar(4)
AS
        INSERT titles
        (title_id, title, Pub_ID)
        VALUES
        (@id, @title, @PubID)
GO
```

NOTE

More About Cursors For more information on cursors, see the section titled "Using Cursors" in this chapter.

There are two advantages to executing a stored procedure instead of manipulating an ADO Recordset:

- **Better resource management**. A `Recordset` object requires a cursor, and cursors can represent a considerable set of resources. By calling a stored procedure, you do not create another cursor and the server does the work.
- **Better management of tier integrity**. You can encapsulate standard business rules in a centralized place (the database) with stored procedures. The stored procedures can be a "black box" to your application. The stored procedures can be changed to meet changing business climates.

Of course, the `Recordset` is more appropriate where you need local control, or where the application itself must exercise a great deal of intelligence about the way it processes data.

Using Stored Procedures to Return Records to an Application

Whenever your application requires one or more records from the data, you will definitely use a cursor, whether you choose a stored procedure, or whether you choose an inline SQL statement.

The text of a stored procedure that returns records can look just like the text of an inline SQL statement (although it may also use more complicated logic than a simple SQL statement). Listing 9.15 gives an example of a simple stored procedure that returns records.

LISTING 9.15

A SQL Statement That Will Create a Stored Procedure to Return Rows in SQL Server

```
create procedure Publishers_All
AS
        Select * from publishers Order By pub_name
GO
```

There are several methods for getting records back from a stored procedure. These methods are discussed here at greater length. Before comparing the three following methods, bear in mind that all ADO techniques for handling rows of data end up with a `Recordset` object. Therefore, the three following methods are really just three different ways of populating a `Recordset` object's rows.

To get records back from a `Connection` object (see Listing 9.16), execute the following steps:

STEP BY STEP

9.2 Getting Records Back From a `Connection` Object

1. Make sure that the `Connection` object is either open or has a valid `ConnectionString` property.
2. Set a `Recordset` object variable to the results of the `Connection` object's `Execute` method.
3. When calling the `Execute` method, pass the following arguments to the method (or set the corresponding properties of the `Connection` object):
   - **`CommandText`.** The name of the stored procedure as a text string.
   - **`RecordsAffected` (optional).** A long variable that the provider will fill with the number of records affected by this query.
   - **`Options`.** Always set options to `adCmdStoredProc` when the `CommandText` argument represents a stored procedure name.

After you have executed the preceding steps, the `Recordset` object should be populated with the records returned by the stored procedure.

LISTING 9.16

USING A CONNECTION OBJECT TO RETURN RECORDS FROM A STORED PROCEDURE

```
Set rsPubs = connPubs.Execute "Titles_All", , adCmdStoredProc
```

To get records back from a stored procedure using a `Command` object (see Listing 9.17), execute the following steps:

STEP BY STEP

9.3 Getting Records Back From a Stored Procedure Using a `Command` Object

1. Set the `Command` object's `CommandType` property to `adCmdStoredProcedure`.
2. Set the `CommandText` property to a string representing the stored procedure's name.
3. Prepare the stored procedure's parameters by using the `Command` object's `Parameters` collection.

4a. Set a `Recordset` object variable to the results of the `Command` object's `Execute` method, or

4b. Run the `Execute` method without setting the result to point to a `Recordset`. Instead, make the `Recordset`'s `ActiveCommand` property point to the `Command` object, and then call the `Recordset`'s `Open` method.

5. In the call to the `Command` object's `Execute` method, you can pass the following arguments (all optional) to the method:
   - **`RecordsAffected`.** A long variable that the provider will fill with the number of records affected by this query.
   - **`Parameters`.** A variant array of values to pass as parameters to the stored procedure. Use this argument as an alternative to setting up the `Command` object's `Parameters` collection. Note that output parameters will not return the correct values when you use this `Parameters` argument.

- **`Options`**. Use this argument as an alternative to set the `CommandType` property. Always set to `adCmdStoredProc` when the `CommandText` argument represents a stored procedure name.

6. After you have executed the preceding steps, the `Recordset` object should be populated with the records returned by the stored procedure.

LISTING 9.17

USING A COMMAND OBJECT TO RETURN RECORDS FROM A STORED PROCEDURE

```
cmdPubs.CommandText = "Titles_All"
cmdPubs.CommandType = adCmdStoredProc

Set rsPubs = cmdPubs.Execute
```

To get records back from a stored procedure into a `Recordset` directly without using the `Execute` methods of `Connection` or `Command` objects, call the `Recordset`'s `Open` method. Make sure that first you set the appropriate properties or pass it the name of the stored procedure as its `Source` argument (first argument) and `adCmdStoredProc` as its `Options` argument (fifth argument) (see Listing 9.18).

LISTING 9.18

RETURNING RECORDS DIRECTLY INTO A RECORDSET FROM A STORED PROCEDURE

```
rsEmployees.CursorType = giCursorType
rsEmployees.CursorLocation = giCursorLocation
rsEmployees.LockType = giLocking
rsEmployees.Source =  _
     "Select * From Employees Order By LastName,FirstName"
Set rsEmployees.ActiveConnection = cnNWind
rsEmployees.Open
```

NOTE

Cursor Behavior Not Always as Advertised Because of the great array of DBMSs that lie behind cursor-implemented rowsets, you may find that some cursor types will give you more or less functionality than documented for a particular combination of cursor location, cursor type, or locking strategy.

USING CURSORS

A *cursor* in a database context represents a facility for managing recordsets as discrete rows of data. A cursor enables you to move through a set of records and implements a pointer to a current row in the set of records.

You don't have to use a cursor to access data from an ADO provider. Instead, you can manipulate the data through straight SQL statements or stored procedures—and you should do so when possible, because a cursor represents a certain amount of resource overhead.

Nevertheless, cursors are often necessary to efficient data processing. Anytime that you create a `Recordset` in ADO, you create a cursor to go along with it.

You have several choices to make when you use a cursor:

- **Cursor location**. You can determine whether a cursor is implemented by the local workstation, or whether it is implemented by the server.
- **Cursor behavior**. You can determine how freely a cursor can move at the request of the consumer and how dynamically it reflects concurrent changes to data made by other users.

The following sections discuss these two issues as well as how to implement various *locking strategies* with cursors.

Using Cursor Locations

▶ Retrieve and manipulate data by using different cursor locations. Cursor locations include client side and server side.

Cursor location is important, because you need to manage where cursors get their resources from (which would include CPU time, memory, and/or temporary storage space on either disk drives or in temporary database objects).

A cursor can be implemented at one of two general locations:

- *Client-side* cursors implement the cursor with resources on the local workstation (the "client machine").
- *Server-side* cursors implement the cursor with resources on the server machine.

You can determine the cursor location of a result set by setting the `CursorLocation` property of an ADO `Recordset` or of an ADO `Connection`. The `CursorLocation` property has two useful values:

- **`2 - adUseServer`.** The cursor will be server side. This is the default cursor location for ADO.
- **`3 - adUseClient`.** The cursor will be client side.

If you set the `CursorLocation` property of a `Connection` object, the `CursorLocation` property of any `Recordset` created from that `Connection` will default to the value of the `Connection`'s `CursorLocation` property, unless you explicitly set the `Recordset`'s `CursorLocation` property to some other value.

The following two sections discuss the consequences of cursor-location choice.

NOTE

Obsolete `CursorLocation` Values Supported For reasons of backward compatibility with earlier systems, the `CursorLocation` property also supports two obsolete values, `adUseNone` (whose value is `1`) and `adUseClientBatch` (whose value is `3`, or the same as `adUseClient`). These cursor location types are no longer used.

Client-Side Cursors

A client-side cursor uses local machine resources to implement a cursor and its set of records.

The advantages of client-side cursors are as follows:

- Because they run locally, they provide better performance when their result sets are a reasonable size.
- Client-side cursors generally provide better scalability, because their performance depends on each client, and not on the server. Therefore, client-side servers place less of a growing demand on the server as the number of a system's users increases.

The disadvantages of client-side cursors are as follows:

- When the rowset returned with the cursor is very large, the local workstation's resources may be "swamped" by the need to handle the high volume.
- Because a client-side cursor must bring all the data for its rowset over the network, larger result sets can increase network traffic.

NOTE

Client-Side and Static Cursors
When you set the `CursorLocation` property to `Client-Side`, the only available `CursorType` is `Static`.

Server-Side Cursors

A server-side cursor uses server resources to implement a cursor and its set of records.

The advantages of server-side cursors are as follows:

- Local workstation resources are never "swamped" by unexpectedly large rowsets.
- Because a server-side cursor does not transfer all the data in the rowset to the workstation, there is less network traffic with large rowsets when they are opened and less delay in opening them.

The disadvantages of server-side cursors are as follows:

- For smaller rowsets with a lot of activity performed by the application, server-side cursors do not perform as well, because each request to move the cursor and each response must travel over the network. It would be better to just transfer the smaller rowsets to the client to start with.
- As users are added to the system, the server receives a greater and greater resource demand as more and more concurrent users open server-side cursors. Server-side cursors therefore typically provide less scalability than client-side cursors.

Using Cursor Types

- Retrieve and manipulate data by using different cursor types. Cursor types include `Forward-Only`, `Static`, `Dynamic`, and `Keyset`.

A cursor's *type*, in ADO terminology, indicates some facts about how it behaves, what you can and can't do with it, and how thrifty or wasteful it is with system resources.

You can set a `Recordset`'s cursor type by setting its `CursorType` just before you open it, as shown in Listing 9.19.

LISTING 9.19

SETTING A Recordset'S CURSORTYPE PROPERTY

```
rsEmployees.CursorType = adOpenStatic
rsEmployees.Open
```

The following sections discuss the four cursor types available from the ADO Cursor library.

Forward-Only Cursors

A Forward-Only cursor behaves a lot like sequential file access: It only furnishes one record at a time, and then only in strict order from the beginning to the end of the rowset.

In other words, you can't use a Forward-Only cursor to skip around in a Recordset's rows. You can only move forward one record at a time until you reach the EOF condition at the end of the Recordset.

If you attempt to use any other Recordset navigation method besides MoveNext, you will generate a runtime error.

A Forward-Only cursor is the default ADO cursor, because it consumes the least resources of all cursor types.

Static Cursors

Static cursors are less economical than Forward-Only cursors, but they allow greater flexibility of movement through the rowset. A Static cursor supports navigation in all directions, and it enables you to make repeat visits to the same record during the same session.

The biggest drawback to a Static cursor is the fact that its rowset doesn't get updated with concurrent changes made by other users.

If User A opens a Static cursor on a set of records and User B makes changes to the records during User A's session, for example, User A will not see the changes made by User B. To see User B's changes, User A's Static cursor would have to close and then be reopened.

The user can make changes to the Static cursor's Recordset, but (once again) the user cannot see changes made by others during the time that the cursor is open. This includes additions and deletions to the records as well as editing changes to individual records.

NOTE

The Most Economical Cursor The most efficient cursor in terms of resource usage is a Forward-Only cursor with its lock type set to Read-Only. This type of cursor is also known as a "firehose cursor." See the section titled "Using Locking Strategies to Ensure Data Integrity" for more information on the Read-Only lock type.

NOTE

Static Cursors and Client-Side Cursors When you set the CursorLocation Property to Client-Side, the only available CursorType is Static.

`Keyset` Cursors

`Keyset`-type cursors have the same freedom of movement in any direction as `Static` cursors. In addition, `Keyset` cursors can immediately see changes to existing records made by other users. However, additions and deletions made by other users are not visible to a `Keyset`-type cursor.

`Dynamic` Cursors

`Dynamic` cursors have all the flexibility and visibility of `Keyset`-type cursors with an extra enhancement: Additions and deletions made by other users are visible to a `Dynamic` cursor.

`Dynamic` cursors are, however, the biggest resource hogs, so you should be very sure that you absolutely need a `Dynamic` cursor before deciding to use one.

Managing Database Transactions

- Manage database transactions to ensure data consistency and recoverability.

Client-server DBMSs support a feature known as the *database transaction* to help manage data integrity and (in some cases) improve performance.

A *transaction* is a set of actions performed on your data that you want to consider as a logical group. You need the concept of a transaction so that all the changes made by a group of actions can stand or fall together. This promotes better data integrity.

The need for database transactions arises when a user or process needs to make multiple changes to the data, and the changes must all stand or fall together as a logical unit. In other words, if only part of the changes end up in the data, the system's integrity will be compromised.

Consider, for example, an operation that attempts to eliminate a customer from the system, along with all the customer's orders and the detail lines for each order. In a relational database system, this will involve at least three tables. Now imagine that the process starts and successfully eliminates the customer's record, as well as some of the

order header records. At that point, the server goes down with only part of the process accomplished. When the system comes back up, there will be "orphaned" order header and perhaps order detail records.

The concept of a database transaction helps to avoid situations such as that given in the example. A database transaction defines a group of operations that must stand or fall together. Typically, the following steps occur:

1. A client process defines the operations that make up a group by *beginning a transaction*.
2. When the client finishes all the operations necessary for the transaction, the server then writes or *commits* all the operations at the same time on a signal from the client. At this point, the transaction is finished.
3. If something happens in the middle of the process to halt further activity, nothing is committed and the data integrity is preserved.
4. The client can also explicitly tell the server to abandon or *roll back* all the operations in a transaction.

In ADO, transactions are implemented with three methods (`BeginTrans`, `CommitTrans`, and `RollbackTrans`) and three events of the `Connection` object (`BeginTransComplete`, `CommitTransComplete`, and `RollbackTransComplete`), as illustrated in Listing 9.20.

The three transaction methods are described as follows:

- The `BeginTrans` method defines the start of a transaction in code. All subsequent data operations on the `Connection` object will be part of a single transaction until the `CommitTrans` or `RollbackTrans` method is called or the connection to the database ends without any action.
- The `CommitTrans` method terminates a transaction successfully. All data operations on the current connection that have taken place because of the matching `BeginTrans` will become permanent in the underlying data.
- The `RollbackTrans` method aborts the operations of a transaction. All data operations on the current connection that have taken place because of the matching `BeginTrans` will be abandoned and will not appear in the underlying data.

In Listing 9.20, the `Sub` procedure `FundsTransfer` manipulates data to transfer funds between two accounts. You begin a transaction with the `BeginTrans` method before manipulating data. If there is an error anywhere in the data manipulation process, you should call `Rollback` to cancel all changes. Otherwise, call `CommitTrans` to save all changes.

LISTING 9.20

USING A TRANSACTION

```
Public Sub FundsTransfer
    On Error GoTo Transfer_Error
    ConnAccounts.BeginTrans
       'Code to debit first account
       'Code to credit second account
     ConnAccounts.CommitTrans
Exit_Transfer:
    Exit Sub
Transfer_Error:
    ConnAccounts.Rollback
    Resume Exit_Transfer
End Sub
```

You can have more than one transaction pending at the same time on a `Connection` object. This will work as long as you are careful to pair `BeginTrans`/`CommitTrans`/`RollbackTrans` sequences. These nested transactions will work because a `CommitTrans` or `RollbackTrans` method only undoes the actions because the most recent pending `BeginTrans` method was called.

In Listing 9.21, the programmer has defined three nested transactions. Transaction A is the outer transaction; its `BeginTrans` and `CommitTrans` methods contain the others. Transaction B contains Transaction C for the same reason. Notice which of the transactions each action belongs to.

LISTING 9.21

NESTED TRANSACTIONS

```
ConnAccounts.BeginTrans ' start of Transaction A
'actions here are part of Transaction A
    ConnAccounts.BeginTrans 'start Transaction B
    'actions here are part of Transaction B
        ConnAccounts.BeginTrans 'start Transaction C
        'actions here are part of Transaction C
        ConnAccounts.CommitTrans 'end Transaction C
```

```
    'actions here are part of Transaction B
    IFollowing logic ends Transaction B
    If <something is wrong> Then
        ConnAccounts.RollbackTrans 'B and C roll back
    Else
        ConnAccounts.CommitTrans 'B and C are committed
    End If
'actions here are part of Transaction A
ConnAccounts.CommitTrans Îend Transaction A
```

An outer transaction controls whether an inner transaction's `CommitTrans` will be honored. If the `RollbackTrans` method is called for an outer transaction, the transactions nested inside it will be rolled back as well, regardless of whether they ended with a `CommitTrans` or a `RollbackTrans`.

If an inner transaction is cancelled with `RollbackTrans`, of course, the inner transaction (along with any other transactions nested further inside it) is rolled back, regardless of whether the outer transaction is committed or rolled back.

Review Listing 9.21 again, and notice that the innermost transaction, Transaction C, terminates with a call to `CommitTrans`. However, all of Transaction C's actions will be rolled back, along with those of Transaction B, if the code detects that something is wrong when it comes time to end Transaction B. In such a case, the code calls the `RollBackTrans` method on Transaction B, automatically ignoring the `CommitTrans` of any transactions nested within it (in this case, Transaction C).

The ADO `Connection` object also has three transaction-related events that correspond to the completion of their respective like-named methods:

- The `BeginTransComplete` event
- The `CommitTransComplete` event
- The `RollbackTransComplete` event

The `BeginTransComplete` event receives the following parameters:

- **`TransactionLevel As Long`**. A number telling you where this transaction is in the hierarchy of nested transactions (`1` is the highest, or outermost transaction, and the value of the parameter increases for more deeply nested transactions).

- **`pError As ADODB.Error`.** A single `Error` object that contains information if the value of the following parameter, `adStatus`, is `adStatusErrorsOccurred`.
- **`adStatus As ADODB.EventStatusEnum`.** If you examine the event procedure stub for this parameter in VB, you will notice that it is the only parameter that isn't passed `ByVal`. This is because you are allowed to change it. The original value of the parameter as passed to the event reflects, as its name implies, the status of the attempt to begin, commit, or Rollback a transaction (`adStatusOK` or `adStatusErrorsOcurred`). You can set the status in the event procedure's code so that this event does not fire again (`adStatusUnWantedEvent`).
- **`Connection As ADODB.Connection`.** Points to the `Connection` object that owns this transaction. Not needed in a VB environment, as each `Connection` object has its own separate set of transaction events (this is not the case, for example in some C++ environments that use ADO).

The other two transaction events, `RollbackTransComplete` and `CommitTransComplete`, do not have `BeginTransComplete`'s first parameter (`TransactionLevel`), but they do have the remaining three parameters (`pError`, `adStatus`, and `pConnection`).

Writing SQL Statements

Structured Query Language (usually referred to as SQL) is the language you use to specify the data included in a `Recordset`. It can also provide you with the vehicle for specifying changes to data through the `Database` object's `Execute` method.

You can use a SQL `Select` statement in the form of a string as the first argument to the `Database` object's `OpenRecordset` method when you want to open either a dynaset or snapshot-type `Recordset`.

You also can use a SQL statement to update data and even modify database structure (such as table definitions and fields) when you pass such a statement in a string to the `Database` object's `Execute` method.

The following sections under this heading detail some of the more important features of SQL `Select` statements.

Writing SQL Statements that Retrieve and Modify Data

▶ Write SQL statements that retrieve and modify data.

A SQL `Select` statement provides a query that can be interpreted by a particular DBMS to retrieve particular data from its tables.

The most basic form of the SQL statement specifies columns (fields) to retrieve from one or more tables in the rows (records) of its result set. In ADO terms, the result set will be the records of a `Recordset` object. The syntax for this most elementary SQL Select statement is this:

```
Select FieldList From TableName
```

where `FieldList` is a comma-delimited list of field names existing in the specified table denoted by `TableName`. You might specify a `Recordset` containing rows, for example, each of whose contents represented the `LastName` and `FirstName` fields from the `Employees` table of the current database:

```
Select FirstName, LastName From Employees
```

You can specify all fields from the table by using the asterisk character (*) instead of writing out all their names:

```
Select * From Employees
```

You could use a SQL statement such as this as the `CommandText` property of a `Command` object or as an argument to the `Execute` method of a `Connection` object or the `Open` method of a `Recordset` object, as illustrated in Listing 9.22.

LISTING 9.22

USING A SQL STATEMENT IN A STRING VARIABLE TO OPEN A RECORDSET

```
Dim strSQL As String
strSQL = _
"Select [First Name],[Last Name],HireDate From Employees"
```

The advantage of first storing the query text to a string variable is that it makes the line that manipulates the data object method more readable. More importantly, it enables you to possibly build the SQL statement in several steps in your code, thus permitting more complex logic to be used in your program to query data.

NOTE

Field Names with Spaces Some DBMSs (such as Microsoft Access) permit spaces in the names of fields. To refer to such a field in a SQL statement, you should surround it with square brackets. A field named `"Last Name"` would appear as `[Last Name]` in a SQL query.

NOTE

Other Types of Clauses In addition to the basic SQL structure, which specifies fields and a table, you can add a number of different types of clauses to the statement. Only the `Where`, `Order By`, and `Join` clauses are discussed in this chapter. You should be aware that other types of clauses exist for SQL `Select` queries.

Using the `Where` Clause to Filter Rows

You can use a `Where` clause in a SQL statement to filter which records are returned in the query's result set. The syntax of a SQL statement containing a `Where` clause is this:

```
Select FieldList From TableName Where Condition
```

The `Condition` of the `Where` clause can be one or more comparison statements using the usual comparison operators such as =, >, <, ≥, and . . ., as well as other operators more specific to the SQL language such as `Like` (for text comparisons) and `Between...And` (for specifying a range of values).

A `Where` clause that only returns records with a field matching a particular value would use the = operator, as in the following example:

```
Select * From Employees Where Dependents = 0
```

A `Where` clause to return values above a certain value would use the > operator, as in the following example:

```
Select * From Employees Where Salary > 40,000
```

You would use similar rules for the <, =, . . ., and ≥ operators.

Note that when quoting literal strings for comparison in a SQL `Where` clause, you use the single quotation mark character, as in the following example:

```
Select * From Employees Where LastName = 'Smith'
```

When using a literal date value in a `Where` clause comparison, you must use the U.S. date format (this format being mm/dd/yy)—even when you are not looking at data with a U.S. date format. You must then set the date off with the # character. The following example illustrates the use of a date value in a SQL `Where` clause:

```
Select * from Employees Where HireDate < #1/1/89#
```

You can use `Between...And` to specify a range of values to allow in the result set, as in the following example, which would allow all employee records with salaries between 40,000 and 80,000, inclusive, into the result set:

```
Select * From Employees Where Salary BETWEEN 40000 AND 80000
```

To obtain a text match with a field that contains a certain string combination (but does not exactly match the string), you can use the `Like` operator and specify "wildcard" characters similar to UNIX, DOS, or Windows operating system "wildcard" characters as used in file specifications. Use the `%` character to specify any number of characters and the _ character to specify a single character. The following example shows a query for all records with a Last Name field beginning with the letter *S*:

```
Select * From Employees Where LastName Like 'S%'
```

String comparisons in `Where` clauses are not case sensitive.

NOTE

Wildcard Characters Can Vary The version of SQL in VB (known as "Jet SQL") uses different wildcard characters from the wildcard characters of standard SQL ("ANSI SQL"). In ANSI SQL, the single-character wildcard is the underscore (_), and the multicharacter wildcard is the percentage symbol (%). In Jet SQL, the wildcard is `?` for single characters and `*` for multiple characters, the same as the wildcard characters used for filename specification in the DOS operating system.

Using the `Order By` Clause to Logically Sort Rows

The `Order By` clause in a SQL `Select` statement will put the rows of the result set in a specified order. The `Order By` clause contains one field name, or several field names separated by commas. If there are several field names, the major sort order starts with the first field name and works on down through the list of field names. The default sorting order for field names of any data type is Ascending (lowest to highest in numeric order and not case-sensitive alphabetic order for strings). You can specify descending order for any field in the `Order By` clause with the keyword `DESC` after the field name. In the following example, you order the employees by hire date (in Descending order), and then by last name:

```
Select * From Employees Order By [Hire Date] DESC, [Last
➥Name]
```

Although the default sorting order is Ascending order, you may specify Ascending order for clarity with the `ASC` keyword.

Writing SQL Statements That Use Joins to Combine Data from Multiple Tables

- Write SQL statements that use joins to combine data from multiple tables.

More often than not, you will need to relate together the records from more than one table in a SQL statement.

Using the `Where` Clause to Connect Tables

You can use the `Where` clause to bring data from more than one table into the result set of a query.

You can specify more than one table's data in the result set of a query by just specifying the field names from each table and the table names in the basic syntax of the `Select` statement. You will always want to explicitly specify the relation between the tables with either a `Where` clause or a `Join` clause, however, for the combination of the data from the two tables to be meaningful.

Assume, for example, that you would like to see a list of every order's date from an `Orders` table in your database along with the customer name for each order. Customer names are held in a `Customers` table and the `Orders` table contains a `CustomerID` field that keys to the `CustomerID` field in the `Customers` table. An example of a `Select` clause that would properly return the information you are seeking would read as follows:

```
Select [Order Date], [Customer  Name] From Orders,
➥Customers Where Orders.[Customer ID] = Customers.
➥[Customer ID]
```

If it didn't include the `Where` clause, the `Select` statement would return a Cartesian product of the two tables—that is, it would match every record in `Orders` with every customer and return a huge, meaningless result set! Note that you don't have to specify which table a field comes from so long as that field's name is unique within the tables you have specified in the `From` clause. If a field's name isn't unique within the tables used in the query, specify its originating table with the *`tablename.fieldname`* syntax.

Using `JOIN` Clauses to Connect Tables

You can create a multitable `Recordset` with a query that uses a `JOIN` clause. There are two types of `JOIN`, and these two `JOIN` types are implemented by three different possible `JOIN` clause types. These types of `JOIN` and the clauses that implement them are as follows:

- **An equi-join or inner join**. This type of join creates records in a result set only when there are matching records from both tables. You can use an `INNER JOIN` clause to create an equi-join.
- **An outer join**. Result sets created using this type of join contain all the records from a specified master table and only those

records from a related lookup table that match the records in the master table. You can implement an outer join with either the `LEFT JOIN` or `RIGHT JOIN` clause. The difference between these two types of join is the order in which you specify the master and lookup tables.

To specify an equi-join between the `Customers` table and the `Orders` table, write a query as in the following example:

```
Select [Company Name], [Order Date] From Customers
INNER JOIN Orders ON Customers.[Customer ID] =
Orders.[Customer ID]
```

The result set would contain only matching information from the `Customers` and `Orders` tables.

If you wanted to display a list of customers and the dates of their orders, but you wanted to include even customers without any orders, you could specify this result set with a `LEFT JOIN`, as in the following example:

```
Select [Company Name], [Order Date] From Customers
LEFT JOIN Orders ON Customers.[Customer ID] =
Orders.[Customer ID]
```

You could achieve the same effect with a `RIGHT JOIN` clause, as follows:

```
Select [Company Name], [Order Date] From Orders
RIGHT JOIN Customers ON Orders.[Customer ID] =
Customers.[Customer ID]
```

Using Locking Strategies to Ensure Data Integrity

- Use appropriate locking strategies to ensure data integrity. Locking strategies include Read-Only, Pessimistic, Optimistic, and Batch Optimistic.

To support data integrity and avoid conflicts between users trying to update the same data at the same time (*concurrency* conflicts), most modern DBMSs support some sort of *locking* scheme.

ADO recognizes four different types of data locking, represented by four enumerated constants:

NOTE

The Most Economical Cursor As stated earlier in this chapter, the most efficient cursor in terms of resource usage is a `Forward-Only` cursor with its lock type set to `Read-Only`. This type of cursor is also known as a "firehose cursor." See the subsection titled "`Forward-Only` Cursor" under the section "Using Cursor Types" for more information on the `Forward-Only` cursor type.

- **`adLockReadOnly` (default)** When a recordset is opened, the user may not make any changes to the data. This ensures that concurrency conflicts with other users are avoided.
- **`adLockPessimistic`** Provider guarantees that a record under editing will be able to have its changes saved. This is usually accomplished by locking the record as soon as it becomes the current record under a cursor. The lock is released when the cursor moves off the record or the recordset is closed.
- **`adLockOptimistic`** Provider does not guarantee that a record under editing will have its changes saved. Provider locks the record only during the update process.
- **`adLockBatchOptimistic`** For server-side cursors, this option guarantees that all cursor options will be supported in the most efficient way.

You can set the type of lock on the data underlying a `Recordset` by setting the `Recordset`'s `LockType` property to one of the previously mentioned values before you open it.

Choosing Cursor Options

As you have seen in the course of this chapter, you must make several decisions about cursors when you want to access data with VB and ADO.

These decisions fall into the following categories:

- Do you really need a cursor at all?
- Should you build the cursor on the client or on the server?
- How should the cursor behave?
- What locking strategy should the cursor implement?

The criterion to use for choosing the right cursor for the job can be stated in one simple, general rule: Build the cursor that uses the fewest resources and still does the job. Remember, the most economical cursor of all is the "firehose" cursor, discussed in the sections titled "`Forward-Only` Cursors" and "Using Locking Strategies to Ensure Data Integrity."

Because of the great array of DBMSs that can be implemented with cursor-implemented rowsets, you may find that some cursor types will give you more or less functionality than documented for a particular combination of cursor location, cursor type, or locking strategy.

To find out what the ADO "official line" is on any given `Recordset`'s capabilities (that is, what ADO believes that the `Recordset` can do, based on what the provider believes that it can do), you can use the `Recordset`'s `Supports` method, passing it one of the following nine constants (whose names are self-explanatory):

- `adAddNew`
- `adApproxPosition`
- `adBookmark`
- `adDelete`
- `adHoldRecords`
- `adMovePrevious`
- `adResync`
- `adUpdate`
- `adUpdateBatch`

Each of these listed constants refers to some capability of the `Recordset` (not explained here, because the `Supports` method is outside the scope of the certification exam). When one of the constants is passed to the `Supports` method, the `Supports` method returns a `True` or `False` value, indicating whether the `Recordset` has that capability.

The following line of code would set the variable `blnCanUpdate` to `True` or `False`, for example, and thus tell you whether the `Update` method would work for the `Recordset` named `rsEmployees`:

```
blnCanUpdate = rsEmployees.Supports(adUpdate)
```

CHAPTER SUMMARY

KEY TERMS

- ActiveX Data Objects
- Concurrency
- Data consumer
- Data cursor
- Data provider
- Data Source Name
- DBMS
- Firehose cursor
- Jet
- Locking
- Open Database Connectivity
- Optimistic locking
- Pessimistic locking
- Rowset
- SQL Server
- Stored Procedure
- Structured Query Language

This chapter covered the following topics:

- Execute Direct, Prepare/Execute, and Stored Procedures data-access models
- Creating and programming with stored procedures
- Using the `Parameters` collection to exchange information with stored procedures
- Cursor locations
- Cursor types
- Database transactions
- SQL statements and syntax
- Locking strategies

APPLY YOUR KNOWLEDGE

Exercises

9.1 Using SQL

This exercise provides some basic experience with SQL for those who are less familiar with it.

Estimated Time: 30 minutes

NOTE

SQL Server Assumed for This Exercise You can run SQL queries in the command environment of just about any contemporary DBMS. This exercise assumes that you are using SQL Server, which comes with the VB Enterprise Edition, and that you can run SQL Enterprise Manager from your workstation.

If the only DBMS that you have is Microsoft Access, you can run the query texts by bringing up a New Query and, in the resulting screen, choosing View, SQL from the menu. Of course, you will have to make up your own queries for the `Nwind` database, because the `pubs` database comes only with SQL Server. You might want to check a good book on SQL Server, such as *SQL Server Unleashed* (Sams Publishing) if you have problems finding or accessing the `pubs` database.

Most other DBMSs also have command-line utilities for executing SQL statements against their data.

1. Open SQL Enterprise Manager to the Server Manager window (see Figure 9.1).

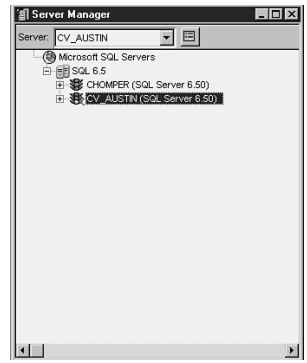


FIGURE 9.1
SQL Enterprise Manager's Server Manager window.

2. Still in the Server Manager window, open the appropriate server, open the `databases` folder under that server, and select the `pubs` database (see Figure 9.2).

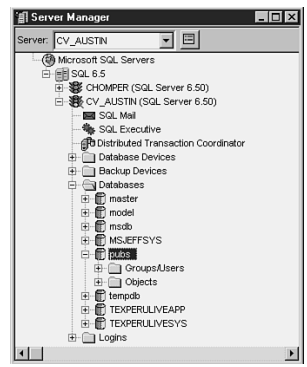


FIGURE 9.2
Selecting a database in SQL Enterprise Manager.

3. Open a Query window with Tools, SQL Query (see Figure 9.3).

APPLY YOUR KNOWLEDGE

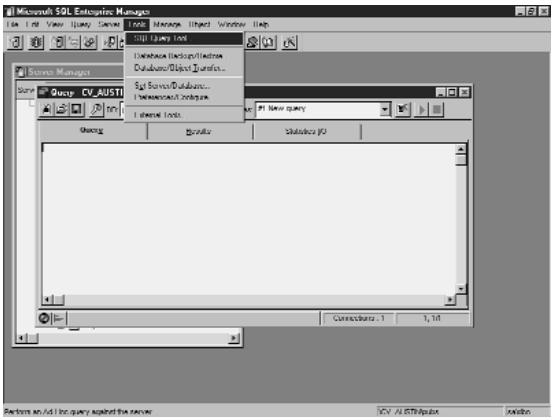


FIGURE 9.3
Opening a new Query window in SQL Enterprise Manager.

4. Run a basic `Select` query to view all records in the `titles` table by typing the following text into the Query window and clicking the run icon or keying Ctrl+E. The text of the query could be this:

```
SELECT * FROM titles
```

5. Modify the `Select` query of the preceding step to filter just certain records. The text of the query could be this:

```
SELECT * FROM titles WHERE title LIKE
➥'%computer%'
```

> **NOTE**
> **Running Just One Query When Several Are in the Query Window** If you have more than one query in the same Query window, you can run just one query at a time by highlighting its text with your mouse and then clicking the Run icon (VCR-style Run button) or keying Ctrl+E.

6. Run a query to insert records into the `titles` table. The text of the query could be this:

```
INSERT titles
(title_id, title)
VALUES
('TGC', 'The Green Computer')
```

Try inserting more records with values of your own invention. If you run exactly the same query a second time, you will get an error from SQL Server, because the `title_id` field is defined as a unique key, and so you can't have two records in the table with the same value in their `title_id` fields.

7. Run a query to modify the contents of one or more of the records that you just inserted. The text of a query could be this:

```
UPDATE titles SET notes = 'ABC'
WHERE title_id = 'TGC'
```

8. Run a query to delete one or more of the records that you inserted and modified in the preceding two steps. The text of the query could be this:

```
DELETE titles WHERE title_id = 'TGC'
```

9. Run a query with a `Where` clause to join records from the `publishers` and `titles` tables. The text of the query could be this:

```
SELECT publishers.pub_id, publisher_name,
➥title_id, title FROM publishers, titles
WHERE publishers.pub_id = titles.pub_id
```

Note that in the list of fields, you must precede the `pub_id` field name with the name of its table, because this field name occurs in both tables.

10. Run a query that performs an inner join, matching corresponding records from the `publishers` and `titles` tables. No records will show up from either table that do not have a corresponding matching record in the other table. The text of the query could be this:

APPLY YOUR KNOWLEDGE

```
SELECT publishers.pub_id, title_id, title
ROM publishers INNER JOIN titles
ON publishers.pub_id = titles.pub_id
```

11. Run a query that performs a left join, matching corresponding records from the `publishers` and `titles` tables, but showing all records from the `publishers` table, regardless of whether they have any matches in the `titles` table. The text of the query could be this:

```
SELECT publishers.pub_id, title_id, title
FROM publishers LEFT JOIN titles
ON publishers.pub_id = titles.pub_id
```

12. Run a query that performs a right join, matching corresponding records from the `publishers` and `titles` tables, but showing all records from the `titles` table, regardless of whether they have a matching record in the `publishers` table. The text of the query could be as follows:

```
SELECT publishers.pub_id, title_id, title
FROM publishers RIGHT JOIN titles
ON publishers.pub_id = titles.pub_id
```

NOTE

Saving Query Texts From the Query Window in SQL Enterprise Manager
You can save the text of SQL Enterprise Manager Query window queries for future use by clicking the Save icon (the disk) in the Query window or by choosing File, Save or File, Save As from the menu (see Figure 9.4). Be sure that you are in the Query window and not the Result window, however. If you are mistakenly in the Result window, you will save the result set and not the query.

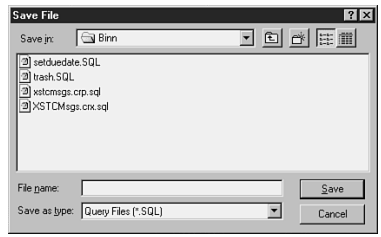


FIGURE 9.4
Saving query text.

9.2 Creating Stored Procedures

You create stored procedures in this exercise so that you can use them in the rest of the exercises. You will use most of these stored procedures in Exercise 9.4.

Estimated Time: 45 minutes

NOTE

SQL Server Assumed for This Exercise
You won't be able to perform this exercise unless you have access to SQL Server or to the command environment of some other client/server database such as Oracle. Although you could create and run `Query` objects in MS Access for the preceding exercise, you can't create stored procedures in that environment.

SQL Server comes with the VB Enterprise Edition.

1. Open SQL Enterprise Manager, select the `pubs` database, and open a Query window with Tools, SQL Query as discussed in the first three steps of the previous exercise.

APPLY YOUR KNOWLEDGE

2. Create a simple stored procedure that runs a basic `SELECT` query to return all records in the `titles` table. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE Titles_All
   AS
   SELECT * FROM Titles ORDER BY title
GO
```

If you are new to SQL Server, don't be alarmed by a message that will appear when you run a non-data query (such as a query to create a stored procedure or manipulate table or index structure): `This command did not return data, and it did not return any rows`. A moment's reflection will tell you that this is exactly the result you would expect if the query operates normally.

3. Create a simple stored procedure that runs a basic `SELECT` query to return all records in the `publishers` table. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE Publishers_All
AS
   Select * from publishers Order By pub_name
   ➥return @@rowcount
GO
```

4. Create a stored procedure by modifying the stored procedure of the preceding step to retrieve only certain records, based on a parameter passed to the stored procedure. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE Find_Titles_Like_Title
   @TitleName varchar(80)
AS
   SELECT * FROM titles WHERE title LIKE
   ➥@TitleName
GO
```

The use of `LIKE` rather than `=` in the `WHERE` clause makes the stored procedure more flexible, because the value of the parameter can contain wildcard characters and therefore doesn't have to specify an exact match.

You should note, however, that a `LIKE` clause does not perform as well as `=`, because `LIKE` requires the server to undertake more processing.

5. Create a stored procedure to insert a record into the `titles` table. The stored procedure will take parameters for the `title_id` and `title` fields of the new record. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE insert_titles
   @id varchar(6),
   @title varchar(80)
AS
   INSERT titles
   (title_id, title)
   VALUES
   (@id, @title)
GO
```

6. Create a stored procedure to return a count of the records that fit a condition. The stored procedure will take input parameters for `pub_id` and will implement an output parameter to return the value to the caller. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE Count_Titles_For_PubID_
➥@PubID varchar(4),
       @Result int output
AS
       select @Result = count(*) FROM titles
       WHERE Pub_ID = @pubid
GO
```

7. Create a stored procedure to modify the contents of the `title` field of a record in the `titles` table.

APPLY YOUR KNOWLEDGE

The stored procedure will take an input parameter for the key field `title_id` to find the record, and a second parameter containing the new value to assign to the `title` field. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE update_titles_title
   @id varchar(6),
   @title varchar(80)
AS
   UPDATE titles SET title = @title WHERE
   ➥title_id = @id
GO
```

8. Create a stored procedure to delete a record from the `titles` table, given an input parameter that provides the value of the `title_id` field of the record to delete. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE delete_titles_by_id
   @id varchar(6)
AS
   DELETE titles
   WHERE title_id = @id
GO
```

9. Create a stored procedure that returns records based on an *inner join* or *equi-join*, matching corresponding records from the `publishers` and `titles` tables. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE Pub_Title_EquiJoin
AS
SELECT pub_name, title
FROM publishers INNER JOIN titles ON
➥publishers.pub_id = titles.pub_id
    ORDER BY pub_name, title
GO
```

10. Create a stored procedure that returns records based on a *left join*, matching corresponding records from the `publishers` and `titles` tables, but showing all records from the `publishers` table, regardless of whether they have any matches in the `titles` table. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE Pub_Title_LeftJoin
AS
SELECT pub_name, title
    FROM publishers LEFT JOIN titles ON
➥publishers.pub_id = titles.pub_id
    ORDER BY pub_name, title
GO
```

11. Create a stored procedure that returns records based on a *right join*, matching corresponding records from the `publishers` and `titles` tables, but showing all records from the `titles` table, regardless of whether they have any matches in the `publishers` table. The text to create the stored procedure could run as follows:

```
CREATE PROCEDURE Pub_Title_RightJoin
AS
   Select pub_name, title from publishers
         Right Join titles
         On publishers.pub_id = titles.pub_id
         Order By pub_name, title
   GO
```

12. Create a stored procedure that counts all title records in the `Titles` table that share the same `Pub_ID` field. This stored procedure takes a parameter, `@PubID`, that is passed from the caller. The parameter represents the value to search for in the `Pub_ID` field:

```
create procedure Find_Titles_PubID
   @PubID varchar(4)
AS
   Select * from titles WHERE Pub_ID = @pubid
GO
```

13. Create a stored procedure that counts all title records in the `Titles` table that share the same `PubID` field. This stored procedure has two parameters. The first parameter gives the value of `Pub_ID` to filter on. The second parameter is an `output` parameter (notice the special keyword in the parameter declaration) that the caller can check after the call to the stored procedure:

APPLY YOUR KNOWLEDGE

```
CREATE PROCEDURE Count_Titles_For_PubID
   @pubID varchar(4),
   @Result int output
AS
   Select @Result = count(*) From titles
➥Where pub_id = @pubid
GO
```

14. If you need to change an existing stored procedure, you must first drop it from the database with a statement of the form:

    ```
    DROP PROCEDURE ProcName
    ```

    If you don't do this, SQL Server will refuse to create a procedure whose name already exists in the database.

9.3 Programming with the Execute Direct and Prepare/Execute Data-Access Models

You perform a few simple actions on data and retrieve records using the Execute Direct and Prepare/Execute data-access models. To implement these models, you will program with the `Command` object's `Execute` method (which always implements the Execute Direct model) and with the `Connection` object's `Execute` method (which can implement either Execute Direct or Prepare/Execute, depending on the setting of its `Prepared` property).

Compare these models with the Stored Procedures model illustrated in the next exercise and with the direct manipulation of the `Recordset` object illustrated in Exercise 8.1 in the preceding chapter.

Estimated Time: 25 minutes

1. Start a new VB standard EXE application. Set a reference to the ADO 2.0 library, as illustrated in step 2 of Exercise 8.1 of the preceding chapter.

2. In the default form's General Declarations, declare an ADO `Connection` object variable and an ADO `Recordset` object variable as follows:

   ```
   Option Explicit
   Private WithEvents connNWind As
   ➥ADODB.Connection
   Private WithEvents rsEmployees As
   ➥ADODB.Recordset
   ```

3. In the form's `Load` event procedure, initialize the ADO `Connection` object with the following code:

   ```
   Private Sub Form_Load()
       Set connNWind = New ADODB.Connection
       Dim sConnect As String
       sConnect = "Provider=
   ➥Microsoft.Jet.OLEDB.3.51;" & _
                  "Data Source= E:\Program
   Files\Microsoft Visual Studio\VB98\Nwind.mdb;"
       connNWind.CursorLocation = adUseClient
       connNWind.Open sConnect
   End Sub
   ```

   Of course, your `Connection` string will not be exactly the same as that of the example, because the path to NWIND.MDB varies from system to system.

4. Place `TextBoxes` on the form's surface and name them `txtLastName`, `txtFirstName`, and `txtHireDate`. Put an appropriate `Label` next to each `TextBox`. Also put a `Label` named `lblEmployeeID` and its `BorderStyle` set to `1-FixedSingle` (see Figure 9.5).

APPLY YOUR KNOWLEDGE

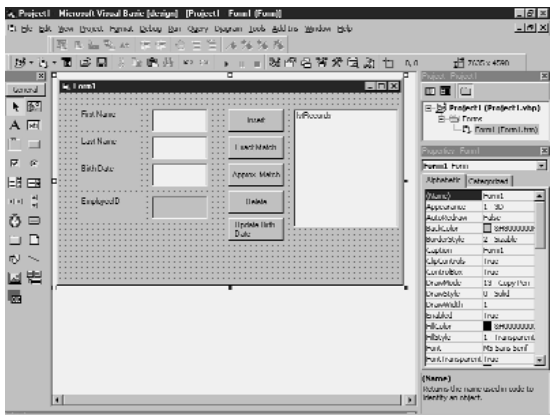


FIGURE 9.5
The completed form for Exercise 9.3.

5. Create a `Private Sub` routine named `ShowEmployeeInfo` to display a current row's fields in the `TextBoxes` and `Label`:

```
Private Sub ShowEmployeeInfo()
    lstRecords.Clear
    On Error Resume Next
    rsEmployees.MoveFirst
    txtFirstName = rsEmployees!firstname
    txtLastName = rsEmployees!lastname
    txtBirthDate = rsEmployees!birthdate
    lblEmployeeID = rsEmployees!employeeid
End Sub
```

6. Write a `Private Sub` procedure named `ShowADOErrors` to show the current elements of the `Connection` object's `Errors` collection:

```
Private Sub ShowADODBErrors()
        Dim err As ADODB.Error
        Dim strErrs As String
        strErrs = ""
        For Each err In connNWind.Errors
            strErrs = strErrs & _
                  err.Number - vbObjectError
➥& ":" & _
                  err.Description & vbCrLf
        Next err
        MsgBox strErrs, , "Please Try again"
End Sub
```

7. Place a `CommandButton` on the form named `cmdInsert` and `Caption` it appropriately, as in Figure 9.5. In its `Click` event procedure, place the following code:

```
Private Sub cmdInsert_Click()
'Uses a Connection object (always Execute
➥Direct model)
    Dim sExecuteString As String
    sExecuteString = "INSERT INTO employees " & _
        "(LastName, FirstName, BirthDate) " & _
        "VALUES (" & _
        "'" & txtLastName & _
        "','" & txtFirstName & _
        "', #" & txtBirthDate & "#)"
        connNWind.Errors.Clear
        On Error GoTo cmdInsert_Error
        connNWind.Execute sExecuteString
        Exit Sub
cmdInsert_Error:
        ShowADODBErrors
End Sub
```

8. Place a `CommandButton` on the form named `cmdDelete` and `Caption` it appropriately, as in Figure 9.5. In its `Click` event procedure, place the following code:

```
Private Sub cmdDelete_Click()
'Uses a Connection object (always Execute
➥Direct model)
    Dim sExecuteString As String
    sExecuteString = "DELETE FROM employees " & _
        "WHERE LastName = '" & _
        txtLastName & "'" & _
        "AND FirstName = '" & _
        txtFirstName & "'"
        connNWind.Errors.Clear
        On Error GoTo cmdDelete_Error
        connNWind.Execute sExecuteString
        lstRecords.Clear
        Exit Sub
cmdDelete_Error:
        ShowADODBErrors
End Sub
```

APPLY YOUR KNOWLEDGE

9. Place a `CommandButton` on the form named `cmdExactMatch` and `Caption` it appropriately, as in Figure 9.5. In its `Click` event procedure, place the following code:

```
Private Sub cmdExactMatch_Click()
'Uses a Connection object (always Execute
➥Direct model)
    Dim sExecuteString As String
    sExecuteString = "SELECT * FROM employees
➥" & _
        "WHERE LastName = '" & _
        txtLastName & "'" & _
        "AND FirstName = '" & _
        txtFirstName & "'"
        connNWind.Errors.Clear
        On Error GoTo cmdExactMatch_Error
        Set rsEmployees = connNWind.
➥Execute(sExecuteString)
        ShowEmployeeInfo
        Exit Sub
cmdExactMatch_Error:
        ShowADODBErrors
End Sub
```

10. Place a `CommandButton` on the form named `cmdApproxMatch` and `Caption` it appropriately as in Figure 9.5. In its `Click` event procedure, place the following code:

```
Private Sub cmdApproxMatch_Click()
'Uses a Command object with Prepare/Execute
➥model

    Dim sExecuteString As String
    sExecuteString = "SELECT * FROM employees
➥" & _
        "WHERE LastName LIKE '" & _
        txtLastName & "%'" & _
        "AND FirstName LIKE '" & _
        txtFirstName & "%'"
        connNWind.Errors.Clear

        Dim comNWind As ADODB.Command
        Set comNWind = New ADODB.Command
        Set comNWind.ActiveConnection =
➥connNWind
        comNWind.CommandType = adCmdText
        comNWind.CommandText = sExecuteString
        comNWind.Prepared = True

        On Error GoTo cmdApproxMatch_Error
        Set rsEmployees =
➥comNWind.Execute(sExecuteString)
        ShowEmployeeInfo
        Exit Sub
cmdApproxMatch_Error:
        ShowADODBErrors
End Sub
```

11. Place a `CommandButton` on the form named `cmdUpdateBirthDate` and `Caption` it appropriately, as shown in Figure 9.5. In its `Click` event procedure, place the following code:

```
Private Sub cmdUpdateBirthDate_Click()
'Uses a Command object with Execute Direct
➥model
    Dim sExecuteString As String
    sExecuteString = "UPDATE employees " & _
        "SET BirthDate = " & _
        "#" & txtBirthDate & "#" & _
        "WHERE LastName = '" & _
        txtLastName & "'" & _
        "AND FirstName = '" & _
        txtFirstName & "'"

        connNWind.Errors.Clear

        Dim comNWind As ADODB.Command
        Set comNWind = New ADODB.Command
        Set comNWind.ActiveConnection =
➥connNWind
        comNWind.CommandType = adCmdText
        comNWind.CommandText = sExecuteString

        On Error GoTo cmdUpdateBirthDate_Error
        comNWind.Execute sExecuteString
        Exit Sub
cmdUpdateBirthDate_Error:
        ShowADODBErrors
End Sub
```

APPLY YOUR KNOWLEDGE

9.4 Programming With the Stored Procedures Data-Access Model

You perform a few simple actions on data and retrieve records using the Stored Procedures data-access model. Compare this model with the other two models given in the preceding exercise and with the direct manipulation of the `Recordset` object illustrated in Exercise 8.1 in the preceding chapter.

This exercise depends on the stored procedures that you created in Exercise 9.2

Estimated Time: 90 minutes

> NOTE
> **SQL Server and Existing Queries Assumed for This Exercise** This exercise depends on the stored procedures that you created in Exercise 9.2. It also assumes that you have access to SQL Server and that you can program in VB ADO with a provider (either ODBC or SQL Server) that gives you access to the `pubs` SQL Server database.

1. Create a new VB standard EXE project with its default startup form.
2. Add a reference to version 2.0 of the ADODB library, as discussed in Exercise 8.1 of the preceding chapter.
3. In the default form's General Declarations section, declare form-wide object variables for an ADO `Connection` and ADO `Recordset`, as follows:

```
Option Explicit
Private WithEvents connPubs As
➥ADODB.Connection
Private WithEvents rsPubs As ADODB.Recordset
```

4. In the form's `Load` event procedure, initialize the connection to point to the `pubs` database in SQL Server, as in the following example:

```
Private Sub Form_Load()
    Dim sConnect As String
    sConnect = "Provider=MSDASQL.1;Data
Source=ODBCpubs"
    Set connPubs = New ADODB.Connection
    connPubs.Open sConnect
End Sub
```

This example uses an ODBC provider in its `Connect` string (assuming you have an appropriately named DSN—`ODBCpubs` in the example):

```
Provider=MSDASQL.1;Data Source=ODBCpubs
```

or you could use a SQL Server provider, as in this example:

```
Provider=SQLOLEDB.1;User
ID=sa;Password=lobster;Location=Chomper;
➥DataBase=pubs
```

> NOTE
> **Results May Vary** The sample provider in the `Connect` strings given here are only examples and may not work in your environment. You will need to investigate your own system's data setup to determine which `Connect` string will work for you.

5. To hold the results of stored procedures that return `Recordset` objects, add a `ListBox` control to the form and name it `lstResults`. Above `lstResults`, put a blank `Label` control and name it `lblResults`. This `ListBox` will contain a list of records returned by the stored procedures that you will run in this exercise (see Figure 9.6).

APPLY YOUR KNOWLEDGE

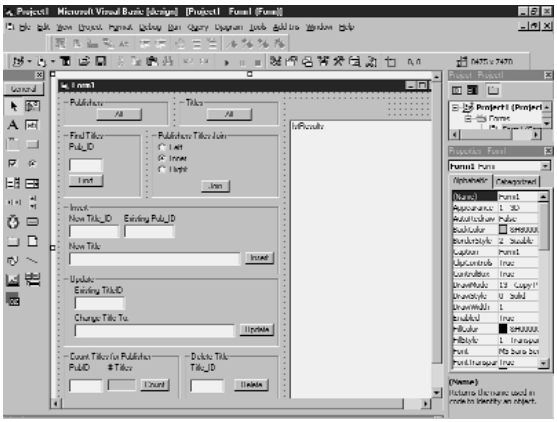


FIGURE 9.6
The completed form as it might appear at the end of Exercise 9.4.

6. Add a `Private Sub` procedure to the form that will display chosen fields from the `Recordset rsPubs` in the `ListBox` that you created in the preceding step:

```
Private Sub ShowFieldsFromRecords( _
       rs As ADODB.Recordset,  _
       ParamArray names())

    'Clear the ListBox
    lstResults.Clear

Dim valToShow As String
    Do Until rs.EOF
        Dim vName As Variant
        Dim strtext As String
        strtext = ""
        For Each vName In names
            If IsNull(rs.Fields(vName).Value)
➥Then
                valToShow = "<NULL>"
            Else
                valToShow =
rs.Fields(vName).Value
            End If
            strtext = strtext & "¦" &
➥valToShow
        Next vName
        strtext = Mid$(strtext, 2)
        lstResults.AddItem strtext
        rs.MoveNext
    Loop
End Sub
```

This routine assumes that the caller is sending it an initialized `Recordset` object with the data cursor currently pointing to the first row in the `Recordset`. It also assumes that you have sent a variable number of parameters (the `ParamArray`) representing names of the fields that you want to display in the `ListBox`.

The routine's `Do Until...` loop traverses all the records in the `Recordset`. Inside the loop, a `For Each...` loop builds a single line to display in the `ListBox` by traversing each element of the `ParamArray`. Each `ParamArray` element gives a `Field` name whose value you extract from the current row in the `Recordset`. If the `Field`'s value is `NULL`, you create a special string to represent that value. You place a "|" symbol between each field that you display.

At the end of the `Do Until...` loop, you drop the initial "|" character on the string that you built in the `For Each...` loop and add the string to the `ListBox`.

7. Return a list of all `Publisher` records as follows (assumes that the `Publishers_All` stored procedure already exists in the `pubs` database, as discussed in Exercise 9.2):

   - On the form's surface, place a `Frame` control and name it `fraPublishers`. Inside the `Frame`, place a `CommandButton` and name it `cmdPublishersAll`. Set the `Captions` of the `Frame` and `CommandButton` appropriately, as shown in Figure 9.6.

APPLY YOUR KNOWLEDGE

- In the `Click` event procedure of `cmdPublishersAll`, place the following code:

```
Private Sub cmdPublishersAll_Click()
    'Initiate a Command object and
    'point it to the form-wide
➥Connection object
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection =
➥connPubs

    'The Command will execute the
➥indicated stored procedure
    cmdPubs.CommandText =
➥"Publishers_All"
    cmdPubs.CommandType =
➥adCmdStoredProc

    'One technique for initializing a
➥stored procedure's parameters:
    'Create a parameter object and then
➥Append
    'it to the Parameters collection of
➥the
    'Command object. Give the Return
➥value its
    'own special name, set by you.
    Dim param As ADODB.Parameter
    Set param = cmdPubs.CreateParameter
➥("Return", _
                  adInteger,  _
                  adParamReturnValue, ,
➥0)
    cmdPubs.Parameters.Append param

    'Set the Recordset to the result of
➥the
    'call to the Stored Procedure
    Set rsPubs = cmdPubs.Execute

    'Display results and close the
➥Recordset
    ShowFieldsFromRecords rsPubs,
➥"pub_id", "pub_name"
    rsPubs.Close
    Set rsPubs = Nothing

    'Display the number of records
➥found
    '(information was put in the RETURN
➥parameter
    'of the Stored Procedure — name ➥427
➥("Return").Value & " publishers found."
    lblResults = "Publishers
➥(PubID¦Name)"
End Sub
```

- Run the application to test the process. When you click the `CommandButton`, you should see values in the `ListBox`.

8. Return a list of all `Title` records as follows:

   - On the form's surface, place a `Frame` control and name it `fraTitles`. Inside the `Frame`, place a `CommandButton` and name it `cmdTitlesAll`. Set the `Captions` of the `Frame` and `CommandButton` appropriately, as shown in Figure 9.6.
   - In the `Click` event procedure of `cmdTitlesAll`, place the following code:

```
Private Sub cmdTitlesAll_Click()
    'Initialize a Command object and
    'point it to the form-wide
➥Connection
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection =
➥connPubs

    'Make the Command use the
➥appropriate
    'stored procedure
    cmdPubs.CommandText = "Titles_All"
    cmdPubs.CommandType =
➥adCmdStoredProc

    'Another technique for programming
➥with the
    'return value of a stored procedure:
    'Refresh parameters:  no specific
➥assignments needed
    'to Parameters collection, because
➥only parameter of
```

APPLY YOUR KNOWLEDGE

```
    'this stored procedure is the Return
➥value,
    'and that's implicit
    cmdPubs.Parameters.Refresh

    'The Command's Execute method
    'returns a Recordset
    Set rsPubs = cmdPubs.Execute

    'We display the Recordset in the
➥ListBox
    ShowFieldsFromRecords rsPubs,
➥"Title_id", "Title"

    'Then we Close the Recordset
    rsPubs.Close
    Set rsPubs = Nothing

    'Return value is always element 0 of
➥Parameters collection
    MsgBox cmdPubs.Parameters(0).Value &
➥" titles found."
    lblResults = "Titles (TitleID¦Title)"
End Sub
```

- Run the application and click the new `CommandButton`. You should see the `ListBox` refresh with new data.

9. Return a list of all `Titles` for a given `Publisher` as follows:

   - Place a `Frame` on the form and name it `fraFind`. Within the `Frame` place a `Label`, a `TextBox` named `txtFindPubID`, and a `CommandButton` named `cmdFind`. Give appropriate `Captions` to the `Frame`, the `Label`, and the `CommandButton` as shown in Figure 9.6.

   - In the `Click` event procedure of `cmdFind`, place the following code:

```
Private Sub cmdFind_Click()
    'Initialize the Command object
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection =
➥connPubs
    cmdPubs.CommandType = adCmdStoredProc
    cmdPubs.CommandText = "Find_Titles_
➥PubID"

    'Create a parameter object
    'To match the parameter that the
    'stored procedure is expecting
    Dim parmCurr As ADODB.Parameter
    Set parmCurr = cmdPubs.
➥CreateParameter( _
                          "@PubID", _
                         adVarChar, _
                        adParamInput, _
                            4, _
                          txtFindPubID.
➥Text)

    'and then append it to the
➥parameters collection
    cmdPubs.Parameters.Append parmCurr

    'The Command's Execute method
    'returns a Recordset
    Set rsPubs = cmdPubs.Execute

    'Display and close the Recordset
    ShowFieldsFromRecords rsPubs,
➥"title_id", "title"
    rsPubs.Close
    Set rsPubs = Nothing
    lblResults = "Titles for Publisher " &
_
                 txtFindPubID & "
➥(title_id¦title)"
End Sub
```

   - Run the application and click the new `CommandButton`. You should see the `ListBox` refresh with new data.

10. Return joined `Publisher-Title` information with various `JOIN` types as follows:

   - Place a `Frame` on the form and name it `fraJoin`. Within the `Frame`, place three `Option Buttons` named `optInner`, `optLeft`, and `optRight`, respectively, and a `CommandButton` named `cmdJoin`. Give appropriate `Captions` to the `Frame`, the `Option Buttons`, and the `CommandButton`, as shown in Figure 9.6.

APPLY YOUR KNOWLEDGE

- In the `Click` event procedure of `cmdJoin`, place the following code:

```
Private Sub cmdJoin_Click()
    'Create and initialize a Command
➥object
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection =
➥connPubs
    cmdPubs.CommandType =
➥adCmdStoredProc

    'Decide which stored procedure to
➥use
    If optInner.Value Then
        cmdPubs.CommandText =
➥"Pub_Title_EquiJoin"
    ElseIf optLeft.Value Then
        cmdPubs.CommandText =
➥"Pub_Title_LeftJoin"
    Else
        cmdPubs.CommandText =
➥"Pub_Title_RightJoin"
    End If

    'Note:  No parameters for these
➥stored
    'procedures.
    cmdPubs.Parameters.Refresh

    'Execute method returns a Recordset
    Set rsPubs = Nothing
    Set rsPubs = cmdPubs.Execute

    'Display and close the Recordset
    ShowFieldsFromRecords rsPubs,
➥"pub_name", "title"
    rsPubs.Close
    Set rsPubs = Nothing
    lblResults = cmdPubs.CommandText &
➥" (Pub_Name¦Title)"
End Sub
```

11. Insert a new `Title` into the `Titles` tables as follows:
    - Place a `Frame` on the form and name it `fraInsert`. Within the `Frame`, place three `TextBoxes` named `txtInsertTitleID`, `txtInsertPubID`, and `txtInsertTitle`, respectively, and a `CommandButton` named `cmdInsert`. Place a `Label` in the `Frame` for each `TextBox`. Give appropriate `Captions` to the `Frame`, the `Labels`, and the `CommandButton` as shown in Figure 9.6.
    - In the `Click` event procedure of `cmdInsert`, place the following code:

```
Private Sub cmdInsert_Click()
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection =
➥connPubs
    With cmdPubs
        .CommandType = adCmdStoredProc
        .CommandText = "Insert_Titles"

        'Refresh Parameters collection
        .Parameters.Refresh

        'and then set properties of ➥each
parameter:

        '@id parameter
        .Parameters("@id").Value =
➥txtInsertTitleID
        .Parameters("@id").Direction =
➥adParamInput

        '@title parameter
        .Parameters("@title").Value =
➥txtInsertTitle
        .Parameters("@title").Direction
➥= adParamInput

        '@PubID parameter
        .Parameters("@PubID").Value =
➥txtInsertPubID
        .Parameters("@PubID").Direction
➥= adParamInput
        .Execute
    End With
    lblResults = ""
    lstResults.Clear
End Sub
```

    - Run the application and test by entering values in the `TextBox` for a new title and then clicking the `CommandButton`. If you now click the `Titles` button, you should see the new `Title` displayed in the `ListBox`.

APPLY YOUR KNOWLEDGE

12. Update an existing title's name as follows:

    - Place a `Frame` on the form and name it `fraUpdate`. Within the `Frame`, place two `TextBoxes` named `txtUpdateTitleID` and `txtUpdateTitleName`, respectively, and a `CommandButton` named `cmdUpdate`. Place a `Label` in the `Frame` for each `TextBox`. Give appropriate `Captions` to the `Frame`, the `Labels`, and the `CommandButton` as shown in Figure 9.6.

    - In the `Click` event for `cmdUpdate`, place the following code:

```
Private Sub cmdUpdate_Click()
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection =
➥connPubs
    With cmdPubs
        .CommandType = adCmdStoredProc
        .CommandText =
➥"Update_Titles_Title"

        'Refresh Parameters collection
        .Parameters.Refresh

        'and then set properties of ➥each
parameter:

        '@id parameter
        .Parameters("@id").Value =
➥txtUpdateTitleID
        .Parameters("@id").Direction =
➥adParamInput

        '@title parameter
        .Parameters("@title").Value =
➥txtUpdateTitleName
        .Parameters("@title").Direction
➥= adParamInput
        .Execute
    End With
    lblResults = ""
    lstResults.Clear
End Sub
```

    - Run the application and enter a known title ID in the `Title ID TextBox`. Type a new title for the record, and click the Update button. When you click the Titles button, you should see an updated entry in the `ListBox`.

13. Count the number of titles for a given publisher as follows:

    - Place a `Frame` on the form and name it `fraCountTitlePublisher`. Within the `Frame`, place a `TextBox` named `txtCountPubID`, a `Label` named `lblCountPublisher` with its `BorderStyle` set to `1-Fixed Single`, and a `CommandButton` named `cmdCountTitlePublisher`. Place a `Label` in the `Frame` above the `TextBox` and one above `lblCountPublisher`. Give appropriate `Captions` to the `Frame`, the two `Labels`, and the `CommandButton` as shown in Figure 9.6.

    - In the `Click` event procedure for `cmdCountTitlePublisher`, place the following code:

```
Private Sub
cmdCountTitlePublisher_Click()
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection =
➥connPubs
    cmdPubs.CommandText =
➥"Count_Titles_For_PubID"

   With cmdPubs.Parameters
        'append directly to parameters
➥collection, using
        'the return value of the
➥CreateParameter method
        .Append ➥cmdPubs.CreateParameter(
_
                                "@PubID", _
                                adVarChar,
```

APPLY YOUR KNOWLEDGE

```
                                adParamInput,
                                4, _
                                RTrim
➥(txtCountPubID.Text))
      .Append cmdPubs.CreateParameter(
                                "@Result",
                                adInteger,
                                adParamOutput,
                                   , _
                                   0)
    End With

    cmdPubs.Execute
    lblCountTitlesPublisher =
➥cmdPubs.Parameters("@Result").Value

End Sub
```

Note that this stored procedure uses an `Input` and an `Output` parameter (the publisher ID and the number of titles, respectively). When you set the `Parameter` objects for these parameters, you must make sure to specify the parameter types properly (`adParamInput` and `adParamOutput`).

- Run the application and place a known publisher ID in the `TextBox`. Click the Count button to verify that the Label refreshes to hold the number of titles for that publisher. (Several publishers have no titles, so it will be normal to see zero for these publishers.)

14. Delete a `Title` from the data as follows:

    - Place a `Frame` on the form and name it `fraDelete`. Within the `Frame`, place a `TextBox` named `txtDeleteTitleID` and a `CommandButton` named `cmdDelete`. Place a `Label` in the `Frame` above the `TextBox`. Give appropriate `Captions` to the `Frame`, the `Label`, and the `CommandButton` as shown in Figure 9.6.
    - Place the following code in the `Click` event procedure of `cmdDelete`:

```
Private Sub cmdDelete_Click()
    Dim cmdPubs As ADODB.Command
    Set cmdPubs = New ADODB.Command
    Set cmdPubs.ActiveConnection =
➥connPubs
    With cmdPubs
        .CommandType = adCmdStoredProc
        .CommandText =
➥"Delete_Titles_By_ID"

        'Refresh Parameters collection
        .Parameters.Refresh

        'and then set properties of ➥each
parameter:

        '@id parameter
        .Parameters("@id").Value =
➥txtDeleteTitleID
        .Parameters("@id").Direction =
➥adParamInput

        .Execute
    End With
    lblResults = ""
    lstResults.Clear
End Sub
```

- Run the application and enter the title ID of one of the `Titles` that you added when testing the `Insert` stored procedure. When you click the Titles button, the entry should be gone.

9.5 Programming With Cursor Locations, Types, and Locking Strategies

You experiment with the `CursorLocation`, `CursorType`, and `LockType` properties of the `Recordset`, `Connection`, and `Command` objects.

Estimated Time: 30 minutes

1. Use the application that you created in Exercise 8.1 of Chapter 8.

APPLY YOUR KNOWLEDGE

2. Add to the form four frames, as follows (see Figure 9.7):

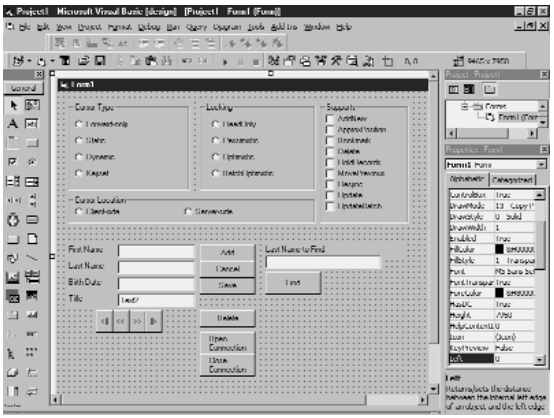


FIGURE 9.7
The completed form for Exercise 9.5.

- `fraCursorType` with `Caption "CursorType"` and containing the following `Option Buttons` with appropriate `Captions` as shown in the figure.
  - `optForwardOnly` (set `Value = True`)
  - `optStatic`
  - `optDynamic`
  - `optKeyset`
- `fraCursorLocation` with `Caption "Cursor Location"` and containing the following `Option Buttons` with appropriate `Captions` as shown in the figure.
  - `optUseClient` (set `Value = True`)
  - `optUseServer`
- `fraLocking` with `Caption "Locking"` and containing the following `Option Buttons` with appropriate `Captions` as shown in the figure.
- `optReadOnly` (set `Value = True`)
- `optPessimistic`
- `optOptimistic`
- `optBatchOptimistic`
- `fraSupports` with `Caption` "Supports," Enabled property = False, and containing the following `CheckBoxes`:
  - `chkAddNew`
  - `chkApproxPosition`
  - `chkBookmark`
  - `chkDelete`
  - `chkHoldRecords`
- `chkMovePrevious`
- `chkResync`
- `chkUpdate`
- `chkUpdateBatch`

3. Add three form-wide variable declarations to keep track of the current cursor location, cursor type, and locking strategy. The entire General Declarations section of the form should now look like the following (the last three lines are what you just added):

```
Option Explicit
Dim WithEvents cnNWind As ADODB.Connection
Dim WithEvents rsEmployees As ADODB.Recordset
Dim cmEmployees As ADODB.Command
Public gblnAddMode As Boolean
Dim iCursorType As Integer
Dim iCursorLocation As Integer
Dim iLocking As Integer
```

APPLY YOUR KNOWLEDGE

4. Add code to the `Click` events of the `Option Buttons` of step 2 that will set the form-wide variables as follows:

```
Private Sub optForwardOnly_Click()
    iCursorType = adOpenForwardOnly
End Sub
Private Sub optStatic_Click()
    iCursorType = adOpenStatic
End Sub
Private Sub optDynamic_Click()
    iCursorType = adOpenDynamic
End Sub
Private Sub optKeyset_Click()
    iCursorType = adOpenKeyset
End Sub
Private Sub optReadOnly_Click()
    iLocking = adLockReadOnly
End Sub
Private Sub optOptimistic_Click()
    iLocking = adLockOptimistic
End Sub
Private Sub optPessimistic_Click()
    iLocking = adLockPessimistic
End Sub
Private Sub optBatchOptimistic_Click()
    iLocking = adLockBatchOptimistic
End Sub
Private Sub optServerSide_Click()
    iCursorLocation = adUseServer
End Sub
Private Sub optClientSide_Click()
    iCursorLocation = adUseClient
End Sub
```

5. Add a `Sub` procedure named `MarkSupports`. It will examine the current `Recordset`'s features using the `Supports` method and will check the boxes in the `Supports` frame accordingly:

```
Private Sub MarkSupports(rs As
➥ADODB.Recordset)
    chkAddNew.Value = IIf(rs.Supports
➥(adAddNew), vbChecked, vbUnchecked)
    chkApproxPosition = IIf(rs.Supports
➥(adApproxPosition), vbChecked, vbUnchecked)
    chkBookmark = IIf(rs.Supports
➥(adBookmark), vbChecked, vbUnchecked)
    chkDelete = IIf(rs.Supports(adDelete),
➥vbChecked, vbUnchecked)
    chkHoldRecords = IIf(rs.Supports
➥(adHoldRecords), vbChecked, vbUnchecked)
    chkMovePrevious = IIf(rs.Supports
➥(adMovePrevious), vbChecked, vbUnchecked)
    chkResync = IIf(rs.Supports(adResync),
➥vbChecked, vbUnchecked)
    chkUpdate = IIf(rs.Supports(adUpdate),
➥vbChecked, vbUnchecked)
    chkUpdateBatch = IIf(rs.Supports
➥(adUpdateBatch), vbChecked, vbUnchecked)
End Sub
```

6. Modify the code in `cmdOpenConnection_Click` to use the form-wide variable values to determine the cursor location, cursor type, and locking strategy of the `Recordset`. Also, call the `MarkSupports` routine after the `Recordset` has been initialized. The modified `Click` event procedure will look like the following (changed lines are in bold italics):

```
On Error GoTo cmdOpenConnection_Error
    Me.MousePointer = vbHourglass
    Set cnNWind = New ADODB.Connection
    Set rsEmployees = New ADODB.Recordset
    Dim sConnect As String
    sConnect = "Provider=Microsoft.
➥Jet.OLEDB.3.51;" & _
                "Data Source=C:\DataSamples\
➥Nwind.mdb "
    cnNWind.CursorLocation = iCursorLocation
    cnNWind.Open sConnect
    rsEmployees.CursorType = iCursorType
    rsEmployees.CursorLocation =
➥iCursorLocation
    rsEmployees.LockType = iLocking
    rsEmployees.Source = "Select * From
➥Employees Order By LastName,FirstName"
    Set rsEmployees.ActiveConnection =
➥cnNWind
    rsEmployees.Open
    rsEmployees.MoveFirst
 MarkSupports rsEmployees
    Exit Sub
cmdOpenConnection_Error:
        Dim adoErr As Error
        For Each adoErr In cnNWind.Errors
            ' Show each error description to
➥the user
            MsgBox adoErr.Description,
➥vbOKCancel, "MY ERROR MESSAGE"
        Next adoErr
End Sub
```

APPLY YOUR KNOWLEDGE

In the `Form_Load` event procedure, place the following code to initialize the three `Option Button` groups. This will also trigger the `Click` events of the respective buttons and therefore set the global variables:

```
Private Sub Form_Load()
    optForwardOnly.Value = True
    optReadOnly.Value = True
    optClientSide = True
End Sub
```

7. Test the application with different combinations of cursor options. Note the differing cursor capabilities shown in the `Supports` frame. Figure 9.7 gives an example of what you might see. Experiment with various actions on the data with different cursor options selected. Note which actions generate runtime errors with given cursor options.

9.6 Managing Database Transactions

In this exercise, you experiment with database transactions as implemented with the `Connection` object's `BeginTransaction`, `CommitTransaction`, and `RollbackTransaction` methods.

Estimated Time: 20 minutes

1. Start a new VB standard EXE project with a single form.
2. Add a reference to the ADO 2.0 library as discussed in Exercise 8.1 in the preceding chapter.
3. Add three `CommandButtons` and name them `cmdBeginTransaction`, `cmdRollbackTransaction`, and `cmdCommitTransaction`. Give them `Captions` as illustrated in Figure 9.8.

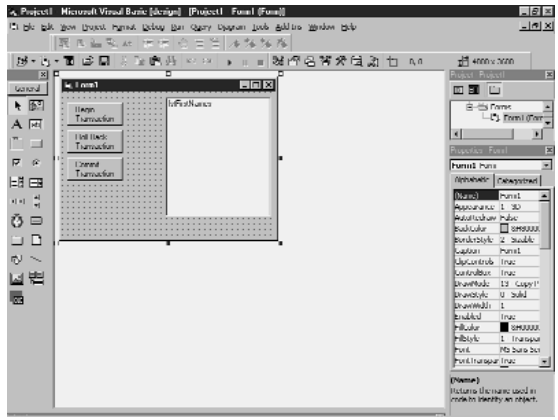


FIGURE 9.8
The completed form for Exercise 9.6.

4. Add a `ListBox` and name it `lstFirstNames` (refer to Figure 9.8). You will use it to display the first names of employees from the `Employees` table in the `Nwind` database.
5. Declare a `Connection`, a `Command`, and a `Recordset` object at the form level, as follows:

```
Option Explicit
Private WithEvents connNwind As ADODB.
➥Connection
Private comNWind As ADODB.Command
Private WithEvents rsEmployees As ADODB.
➥Recordset
```

6. Write a `Sub` procedure called `ShowFirstNames` that will display the values of the `FirstName` fields of all employee records in the `ListBox`:

```
Private Sub ShowFirstNames()
    lstFirstNames.Clear
    rsEmployees.Requery
    rsEmployees.MoveFirst
    Do Until rsEmployees.EOF
        lstFirstNames.AddItem rsEmployees!
➥firstname
        rsEmployees.MoveNext
    Loop
End Sub
```

APPLY YOUR KNOWLEDGE

7. In the `Form_Load` event procedure, initialize the `Connection` and `Recordset` as follows, calling `ShowFirstNames` after the `Recordset` is initialized:

```
Private Sub Form_Load()
    Set connNwind = New ADODB.Connection
    Dim sConnect As String
    sConnect = "Provider=Microsoft.Jet.
➥OLEDB.3.51;" & _
               "Data Source= E:\Program Files\
➥Microsoft Visual Studio\VB98\Nwind.mdb;"
    connNwind.CursorLocation = adUseClient

    connNwind.Open sConnect
    Set comNWind = New ADODB.Command
    Set rsEmployees = New ADODB.Recordset
    rsEmployees.CursorType = adOpenStatic
    rsEmployees.LockType = adLockOptimistic

    rsEmployees.Source = "Select * From
➥Employees Order By LastName,FirstName"
    Set rsEmployees.ActiveConnection =
➥connNwind
    rsEmployees.Open
    ShowFirstNames
End Sub
```

8. Write a `Sub` procedure called `SetButtons-ForPendingTran`. It will take a Boolean parameter that can be used to determine how the three buttons are enabled:

```
Private Sub SetButtonsForPendingTran
➥(TranIsPending As Boolean)
    cmdBeginTrans.Enabled = Not
➥TranIsPending
    cmdRollBackTrans.Enabled = TranIsPending
    cmdCommitTrans.Enabled = TranIsPending
End Sub
```

9. In the `Click` event procedure for `cmdBeginTrans`, write the following code to initiate a database transaction and switch the case of all the employees' first names. At the end of the event procedure, display the first names in the `ListBox` and call `SetButtonsForPendingTran` with a `True` argument:

```
Private Sub cmdBeginTrans_Click()
    connNwind.BeginTrans

    rsEmployees.MoveFirst
    If rsEmployees!firstname =
➥UCase$(rsEmployees!firstname) Then
        Do Until rsEmployees.EOF
            rsEmployees!firstname =
➥LCase$(rsEmployees!firstname)
            rsEmployees.MoveNext
        Loop
    Else
        Do Until rsEmployees.EOF
            rsEmployees!firstname =
➥UCase$(rsEmployees!firstname)
            rsEmployees.Update
            rsEmployees.MoveNext
        Loop
    End If
    SetButtonsForPendingTran (True)
    ShowFirstNames
End Sub
```

10. In the `Click` event procedure for `cmdCommitTrans`, write the following code to commit the pending database transaction, redisplay the first names, and call `SetButtonsForPendingTran` with a `False` argument:

```
Private Sub cmdCommitTrans_Click()
    connNwind.CommitTrans
    SetButtonsForPendingTran (False)
    ShowFirstNames
End Sub
```

11. In the `Click` event procedure for `cmdRollBackTrans`, write the following code to roll back the pending database transaction, re-display the first names, and call `SetButtonsForPendingTran` with a `False` argument:

```
Private Sub cmdRollBackTrans_Click()
    connNwind.RollbackTrans
    SetButtonsForPendingTran (False)
    ShowFirstNames
End Sub
```

APPLY YOUR KNOWLEDGE

12. Run the application and note how changes to the records are discarded on a rollback but accepted on a commit.
13. Also note the effect of beginning a transaction and then exiting the application without either a commit or a rollback. The next time you run the application, you will see that exiting the application with a transaction pending has the same effect as a rollback.

Review Questions

1. Name some advantages of stored procedures over inline SQL statements.
2. What keyword begins a SQL statement to retrieve records?
3. What are the two cursor locations?
4. What's the most resource-efficient combination of cursor type and cursor locking strategy?
5. What is meant by the term "nested transaction?"
6. Write a statement using the `JOIN` clause to display matching records between a table named `Customer` and a table named `Orders`, based on fields in each named `CustID`.
7. Describe the difference between pessimistic and optimistic locking.
8. What property of what ADO object do you need to implement the Prepare/Execute model?
9. Describe a situation where the Execute Direct data-access model would be appropriate.

Exam Questions

1. The Execute Direct data-manipulation model is appropriate when
   - A. You need to perform one-time-only operations on the data.
   - B. You need to execute queries typed by users.
   - C. You need to execute the same dynamic query several times during a single session of your application.
   - D. You need to execute the same query over many sessions and from many different workstations.
2. You can implement the Execute Direct model with (Select all that apply.)
   - A. An argument to the `Connection` object's `Execute` method.
   - B. An argument to the `Recordset` object's `Open` method.
   - C. The `Command` object's `CommandText` property.
   - D. An argument to the `Command` object's `Execute` method.
3. The Prepare/Execute data manipulation model is appropriate when
   - A. You need to perform one-time-only operations on the data.
   - B. You need to execute queries typed by users.
   - C. You need to execute the same dynamic query several times during a single session of your application.
   - D. You need to execute the same query over many sessions and from many different workstations.

APPLY YOUR KNOWLEDGE

4. You can implement the Prepare/Execute model with (Select all that apply.)

   A. An argument to the `Connection` object's `Execute` method.

   B. An `argument` to the `Recordset` object's `Open` method.

   C. The `Command` object's `CommandText` property.

   D. An argument to the `Command` object's `Execute` method.

5. The Stored Procedures data manipulation model is appropriate when (Select all that apply.)

   A. You need to perform one-time-only operations on the data.

   B. You need to execute queries typed by users.

   C. You need to execute the same dynamic query several times during a single session of your application.

   D. You need to execute the same query over many sessions and from many different workstations.

6. You can implement the Stored Procedures model with (Select all that apply.)

   A. An argument to the `Connection` object's `Execute` method.

   B. An argument to the `Recordset` object's `Open` method.

   C. The `Command` object's `CommandText` property.

   D. An argument to the `Command` object's `Execute` method.

7. Which of these SQL Server statements will correctly create a stored procedure with two parameters and a return value?

   A.
```
CREATE PROCEDURE Find_Result AS
     Int @MyParm1 Output
     Int @MyParm2
@MyParm1 = Select LastName from Employee
➥Where
             EmployeeID = @MyParm2
If ISNULL @MyParm1
     Return 0
Else
     Return 1
GO
```

   B.
```
CREATE PROCEDURE Find_Result
     Int @MyParm1 Output
     Int @MyParm2
AS
@MyParm1 = Select LastName from Employee
➥Where
             EmployeeID = @MyParm2
If ISNULL @MyParm1
     Find_Result = 0
Else
     Find_Result = 1
GO
```

   C.
```
CREATE PROCEDURE Find_Result
     Int @MyParm1 Output,
     Int @MyParm2
AS
@MyParm1 = Select LastName from Employee
➥Where
             EmployeeID = @MyParm2
If ISNULL @MyParm1
     Return 0
Else
     Return 1
GO
```

APPLY YOUR KNOWLEDGE

D.
```
CREATE PROCEDURE Find_Result
AS
     Int @MyParm1 Output
     Int @MyParm2
@MyParm1 = Select LastName from Employee
➥Where
            EmployeeID = @MyParm2
If ISNULL @MyParm1
     Find_Result = 0
Else
     Find_Result = 1
GO
```

8. After you have executed a SQL Server stored procedure that implements a return value, you can check the value that the stored procedure returned by

   A. Using the individual variables corresponding to the stored procedure's `Output` parameters in the `Parameters` array argument to the `Command` object's `Execute` method.

   B. Checking the final element of the `Command` object's `Parameters` collection.

   C. Checking element `0` of the `Command` object's `Parameters` collection.

   D. Checking the element of the `Command` object's `Parameters` collection that corresponds to the last `Output` parameter declared in the stored procedure.

9. A SQL Server stored procedure designed to return records to a VB app using ADO (Select all that apply.)

   A. Must have at least one `Output` parameter.

   B. Must have at least one `Input` parameter.

   C. Must implement a return value.

   D. Doesn't necessarily have any parameters.

10. A client-side cursor (Select all that apply.)

    A. Can be better than a server-side cursor for smaller rowsets.

    B. Can be better than a server-side cursor if you need visibility of other users' changes.

    C. Can be more scalable than a server-side cursor as users are added to the system.

    D. Is the only option for persistent `Recordset` objects.

11. A server-side cursor does not support which of the following `Recordset` properties? (Select all that apply.)

    A. `AbsolutePosition`

    B. `Bookmark`

    C. `RecordCount`

    D. `EOF`

12. The `Static` cursor type (Select all that apply.)

    A. Does not allow user updates of any kind on the data.

    B. Allows user updates, but only on the local copy of the data.

    C. Doesn't make other users' updates visible.

    D. Doesn't make other users' deletions or inserts visible.

APPLY YOUR KNOWLEDGE

13. A `Keyset` cursor has which of the following attributes? (Select all that apply.)

    A. Gives visibility of other users' deletions of records

    B. Gives visibility of other users' edits to existing records

    C. Allows movement in any direction through the `Recordset`

    D. Allows the user to update, add, and delete records in the underlying database

14. Calling a `Rollback` on a transaction will

    A. Cause an error if the current transaction is nested within other transactions.

    B. Have no effect if it is performed on transactions nested within the current transaction.

    C. Cancel all transactions nested within the current transaction.

    D. Cause an error if other transactions are nested within the current transaction.

15. A transaction will be committed

    A. When the connection to the data provider is dropped or closed.

    B. When the application ends.

    C. When there is an update without an explicitly defined transaction.

    D. When the transaction was rolled back, but is nested inside another transaction that is committed.

16. A SQL statement to delete all records from the `employees` table would read

    A. `DELETE "%" FROM employees`

    B. `DELETE * FROM employees`

    C. `DELETE FROM employees`

    D. `DELETE FROM employees WHERE *`

17. A SQL statement to insert a new record might read:

    A.
    ```
    INSERT (LastName, FirstName)
    INTO employees
    VALUES ("Romero", "Jose Antonio")
    ```

    B.
    ```
    INSERT INTO employees
    ("LastName", "FirstName")
    VALUES ("Romero", "Jose Antonio")
    ```

    C.
    ```
    INSERT ("LastName", "FirstName")
    INTO employees
    VALUES ("Romero", "Jose Antonio")
    ```

    D.
    ```
    INSERT INTO employees
    (LastName, FirstName)
    VALUES ("Romero", "Jose Antonio")
    ```

18. A SQL statement that uses the `JOIN` clause to match records from two tables, showing all records from one of the tables regardless of whether they have matches in the other table might be called what? (Select all that apply.)

    A. An outer join

    B. An inner join

APPLY YOUR KNOWLEDGE

C. An equi-join

D. A right join

19. The SQL statement

```
SELECT * FROM customers LEFT JOIN orders
ON customers.custid = orders.custid
```

will:

A. Display all customers, regardless of whether they have any orders.

B. Display all orders, regardless of whether they have any customers.

C. Display all orders and all customers, regardless of whether they have matching records in the other table.

D. Display only orders and customers that have matching records in the other table.

20. An optimistic locking strategy does what?

A. Locks data when the cursor moves to that data

B. Locks data when the user begins to edit

C. Locks data when the `Update` method is called

D. Locks data when a second user attempts to move the cursor to that data

21. What is VB ADO's default locking strategy?

A. `adLockReadOnly`

B. `adLockPessimistic`

C. `adLockOptimistic`

D. `adLockBatchOptimistic`

Answers to Review Questions

1. Stored procedures take up fewer workstations resources than inline SQL statements, provide persistent data-manipulation models, help to encapsulate business rules, and perform faster. See "Using Stored Procedures."

2. The `SELECT` keyword begins a SQL statement to retrieve records. See "Writing SQL Statements that Retrieve and Modify Data."

3. The two cursor locations are client side and server side. See "Using Cursor Locations."

4. The most resource-efficient combination of cursor type and cursor locking strategy is the so-called *firehose cursor*, which is a `Forward-Only`, `Read-Only` cursor. See "Choosing Cursor Options."

5. A nested transaction is one that occurs completely within another transaction. See "Managing Database Transactions."

6. A SQL statement to display matching records between a table named `Customer` and a table named `Orders` might read as follows:

```
SELECT * FROM Orders
         INNER JOIN Customers
         ON Orders.CustID = Customer.CustID
```

See "Writing SQL Statements that Use Joins to Combine Data from Multiple Tables."

7. Pessimistic locking strategies typically lock a record early on in the retrieve-edit-save cycle; optimistic locking strategies wait until the last possible moment to lock the record (the moment the record's changes are saved). See "Using Locking Strategies to Ensure Data Integrity."

APPLY YOUR KNOWLEDGE

8. You can use the ADO `Command`'s `Prepare` property to implement the Prepare/Execute model. See "Accessing Data With the Prepare/Execute Model."
9. The Execute Direct data-access model would be most appropriate in situations where you want to run a query just once that will not be run again. See "How to Choose a Data-Access Model."

Answers to Exam Questions

1. **A, B**. The Execute Direct data-manipulation model is appropriate when you need to perform one-time-only operations on the data or you need to execute queries typed by users. The Prepare/Execute model would be more appropriate when you need to execute the same dynamic query several times during a single session of your application, and the Stored Procedures model would be more appropriate when you need to execute the same query over many sessions and from many different workstations. For more information, see the section titled "How to Choose a Data-Access Model."
2. **A, B, C, D**. You can implement the Execute Direct model with an argument to the `Connection` object's `Execute` method, an argument to the `Recordset` object's `Open` method, the `Command` object's `CommandText` property, or an argument to the `Command` object's `Execute` method. For more information, see the section titled "Accessing Data With the Execute Direct Model."
3. **C**. The Prepare/Execute data manipulation model is appropriate when you need to execute the same dynamic query several times during a single session of your application. The Execute Direct would be more appropriate in the first two cases, because it might not be repeated (the Execute Direct model is more efficient for a single execution, but Prepare/Execute is more efficient for subsequent executions after the first one). The Stored Procedures model is more appropriate when you need to execute the same query over many sessions and from many different workstations. For more information, see the sections titled "Accessing Data with the Prepare/Execute Model" and "How to Choose a Data-Access Model."
4. **C, D**. You can implement the Prepare/Execute model only with a `Command` object, because you prepare the data statement by setting the `Command` object's `Prepared` property to `True`. For more information, see the section titled "Accessing Data With the Prepare/Execute Model."
5. **D**. The Stored Procedures data-manipulation model is appropriate when you need to execute the same query over many sessions and from many different workstations. See the explanations for answers 5 and 7. For more information, see the section titled "Accessing Data with the Stored Procedures Model" and "How to Choose a Data-Access Model."
6. **A, B, C, D**. You can implement the Stored Procedures model with an argument to the `Connection` object's `Execute` method, an argument to the `Recordset` object's `Open` method, the `Command` object's `CommandText` property, or an argument to the `Command` object's `Execute` method. For more information, see the section titled "Accessing Data With the Stored Procedures Model."
7. **C**. The following SQL Server statement will correctly create a Stored Procedure with two parameters and a return value:

APPLY YOUR KNOWLEDGE

```
CREATE PROCEDURE Find_Result AS
    Int @MyParm1 Output,
    Int @MyParm2
@MyParm1 = Select LastName from Employee
➥Where
           EmployeeID = @MyParm2
If ISNULL @MyParm1
    Return 0
Else
    Return 1
GO
```

The problem with answer A is that it incorrectly declares the parameters after the `AS` keyword. Answer B incorrectly uses the stored procedure's name to set the return value. Answer D has both A's and B's problems. For more information, see the section titled "Creating Stored Procedures."

8. **C**. After you have executed a SQL Server stored procedure that implements a return value, you can check the value that the stored procedure returned by checking the value of element `0` of the `Parameters` collection. Answers A and D are incorrect, because a stored procedure's return value is not implemented as an `Output` parameter. Answer B is incorrect because it is the first element of the `Parameters` collection, not the last, that holds the return value. For more information, see the section titled "Using the `Parameters` Collection to Manipulate and Evaluate Parameters for Stored Procedures."

9. **D**. A SQL Server stored procedure designed to return records to a VB app using ADO doesn't necessarily have any parameters. For instance, it could be a simple `SELECT` statement returning all rows in a table. The situations described by the other three options (at least one output parameter, at least one input parameter, implement a return value) are all possibilities for such a stored procedure, but none of them are necessary. For more information, see the section titled "Using Stored Procedures."

10. **A, C, D**. A client-side cursor can be better than a server-side cursor for smaller rowsets, can be more scalable than a server-side cursor as users are added to the system, and is the only option for persistent `Recordset` objects. Answer B is incorrect, because a server-side cursor provides better visibility of other users' changes. For more information, see the sections titled "Client-Side Cursors," "Server-Side Cursors," and "Choosing Cursor Options."

11. **A, B, C**. A server-side cursor does not support the `AbsolutePosition`, `Bookmark`, and `RecordCount` properties of the `Recordset`. For more information, see the section titled "Server-Side Cursors."

12. **C, D**. The `Static` cursor type doesn't make other users' updates visible, and it doesn't make other users' deletions or inserts visible. Answers A and B are wrong, however, because a `Static` cursor allows user updates on server data. For more information, see the section titled "`Static` Cursors."

13. **B, C, D**. A `Keyset` cursor gives visibility of other users' edits to existing records; allows movement in any direction through the `Recordset`; and allows the user to update, add, and delete records in the underlying database. However, a `Keyset` cursor does not give visibility of other users' additions or deletions of records. Only a `Dynamic` cursor does this. For more information, see the section titled "`Keyset` Cursors" and "`Dynamic` Cursors."

14. **C**. Calling a rollback on a transaction will cancel all transactions nested within the current transaction. For more information, see the section titled "Managing Database Transactions."

APPLY YOUR KNOWLEDGE

15. **C**. A transaction will be committed when there is an update without an explicitly defined transaction. In the cases of the other answers, a rollback will occur. For more information, see the section titled "Managing Database Transactions."

16. **C**. SQL statement to delete all records from the `employees` table would read:

    ```
    DELETE FROM employees
    ```

    For more information, see the section titled "`DELETE` Statements in SQL."

17. **D**. A SQL statement to insert a new record might read as follows:

    ```
    INSERT INTO employees
    (LastName, FirstName)
    VALUES ("Romero", "Jose Antonio")
    ```

    For more information, see the section titled "`INSERT` Statements in SQL."

18. **B, C**. A SQL statement that uses the `JOIN` clause to match records from two tables, showing all records from one of the tables regardless of whether they have matches in the other table might be called an *equi-join*. An *outer join* (answer A) shows all records from one of the two tables, and only matching records from the other. A *right join* (answer D) is an outer join that shows all records from the second named table, and only matching records from the first named table. For more information, see the section titled "Using `JOIN` Clauses to Connect Tables."

19. **A**. The SQL statement

    ```
    SELECT * FROM customers LEFT JOIN orders
    ON customers.custid = orders.custid
    ```

    will display all customers, regardless of whether they have any orders. For more information, see the section titled "Using `JOIN` Clauses to Connect Tables."

20. **C**. An optimistic locking strategy locks data when the `Update` method is called. Answers A and B describe two possible pessimistic locking strategies, and answer D is not a valid locking strategy. For more information, see the section titled "Using Locking Strategies to Ensure Data Integrity."

21. **A**. VB ADO's default locking strategy is `adLockReadOnly`. For more information, see the section titled "Using Locking Strategies to Ensure Data Integrity."

OBJECTIVE

This chapter helps you prepare for the exam by covering the following objective:

Instantiate and invoke a COM component (70-175 and 70-176).

- **Create a VB application that uses a COM component.**
- **Create a VB application that handles events from a COM component.**

▶ If you've used earlier versions of VB or other Windows programming environments, you are probably already familiar with COM components under other names, such as OLE servers (VB4 and earlier) or ActiveX servers (VB5). A COM component has the following characteristics:

- It provides one or more object classes that Windows programmers can use to build applications.
- It follows the COM (Component Object Model) specification for implementing objects in a computing environment and for communications between those objects. Microsoft implements the COM specification in its ActiveX standard. In most situations in VB, you will hear the terms COM and ActiveX used loosely to mean the same thing (though this is not strictly true).

▶ This chapter discusses how VB programmers can write VB programs that use existing COM components available in the target computing environment.

▶ In addition, some COM components expose events (just as built-in VB controls and other objects expose events). As the second subobjective listed above implies, we will also discuss how to program the events of such components.

CHAPTER 10

Instantiating and Invoking a COM Component

OBJECTIVE

- You also need to understand what this chapter *doesn't* cover about COM. Two objectives on the certification exam related to the one covered here include creating callback procedures for asynchronous COM component processing (see Chapter 12, "Creating a COM Component that Implements Business Rules or Logic", for coverage of this objective) and debugging VB code that uses COM objects (see Chapter 19, "Implementing Project Groups to Support the Development and Debugging Process").
- VB programmers can also create their own COM components for use in other applications. This is an increasingly important aspect of modern Windows programming, and the certification exam covers COM component creation in great detail. We discuss the creation of COM components in Chapter 12, Chapter 13, "Creating ActiveX Controls", and Chapter 14, "Creating an Active Document."

OUTLINE

COM, Automation, and ActiveX **449**

Creating a Visual Basic Client Application that Uses a COM Component **451**

Setting a Reference to a COM Component 452

Using the Object Browser to Find Out About a COM Component's Object Model 453

Using the `New` Keyword to Declare and Instantiate a Class Object from a COM Component 455

Late and Early Binding of Object Variables 456

Using the `CreateObject` and `GetObject` Functions to Instantiate Objects 458

Using a Component Server's Object Model 460

Manipulating the Component's Methods and Properties 461

Releasing an Instance of an Object 463

Detecting Whether a Variable Is Instantiated 463

Handling Events From a COM Component **463**

Chapter Summary **467**

Study Strategies

- Make sure that your computer has at least one major COM component application installed on it so you can actually do your own COM component programming. This would include any of the Microsoft Office applications, especially either Microsoft Word or Microsoft Excel (since the VB certification exams have traditionally concentrated on examples from these two applications). This chapter uses examples with Excel.
- Become at least slightly familiar with the component's object model by using the Object Browser and the object model's documentation (usually available from inside the Object Browser by pressing the F1 key).
- Experiment on your own using VB to program the component's object model, as discussed in this chapter.
- Make sure you know the difference between use of the `GetObject` and `CreateObject` functions. Pay special attention to `GetObject` since its parameters can be tricky (you should learn the rules for their use by heart). `GetObject` can cause runtime errors if not used properly.
- Know the different strategies for declaring and instantiating COM components, including the difference between early and late binding, the use of the `As New` keyword, and the `CreateObject` and `GetObject` functions.
- Experiment with programming a COM object's events, and in particular, make sure you know by heart the syntax for using the `WithEvents` keyword in object declarations.

INTRODUCTION

COM Automation lets you use functionality from a COM component (which may be a standalone application, an ActiveX DLL library, or a module inside an application) using objects.

Most Microsoft products and commercial vendors' applications provide a COM interface to the services that applications provide. Applications that provide services via a COM interface are called COM components. The client applications that use COM components are called COM clients. There are two basic types of COM clients. One type of client is the ActiveX Document container. This type of client application is used to host ActiveX Documents. The other type of COM client, and the type that we will discuss in this chapter, communicates with COM components through automation. Chapter 12, "Creating a COM Component that Implements Business Rules or Logic," provides more detail on creating COM components.

Automation enables you to script commands to applications. Therefore, your client applications can use a COM component that has exposed its objects to the outside world. This chapter discusses how to use COM components in a VB6 application.

This chapter covers the following topics:

- Understanding the meaning of COM and the ActiveX standard.
- Using `GetObject` and `CreateObject` to instantiate COM component object variables that contain references to COM components.
- Understanding and using early and late binding for object variables that are instances of COM components.
- Using automation to handle a COM component's methods and properties.
- Using automation to handle a COM component's events in a VB program.
- Debugging VB code that uses objects from a COM component.
- Releasing an instance of a COM component from memory.
- Using the Object Browser to get and interpret information about the objects made available by a COM component.

COM, AUTOMATION, AND ACTIVEX

The term ActiveX describes a set of services provided as part of the Windows operating system. The services provided with ActiveX are based on a set of services provided with the Component Object Model (COM), summarized as follows:

- Object Management
- Object Persistence
- Structured Storage
- Data Transfer
- Naming and Binding Services

COM provides Object Management a services by using *reference counting*. With reference counting, COM enables developers to control when object references and their objects are released from memory. In Visual Basic, the management of object references is provided by the Visual Basic runtime engine. In addition COM provides Object Persistence services that allow objects to be stored in a file. This is particularly important when you wish to extract an application-specific object from a document file.

When objects are stored in files, they are stored using the Structured Storage services provided in COM. With Structured Storage, you can store data in a hierarchical format within a file very much as files are stored in directories and sub-directories. Structured Storage allows applications to read and save items to files without restructuring the layout of the file.

COM provides Data Transfer a services that enable you to transfer and share data between applications. With Naming and Binding services (Monikers), applications can create, store, and manage objects that provide complex operations such as asynchronous downloading of files. These services are usually not directly accessed by the Visual Basic developer; however you will use these services via the objects and functions that Visual Basic provides.

ActiveX is a Microsoft standard based on COM. In earlier versions of VB and in other documentation before 1996, Microsoft referred to the object linking and embedding (OLE) standard. The ActiveX standard has replaced the OLE standard. ActiveX enables the programmer to visually link or embed objects in a program just as OLE 1.0 did.

ActiveX also gives the programmer all the possibilities provided by OLE 2.0 for application automation, and ActiveX reaches beyond OLE to provide a general standard for component development and manipulation under Windows and across the Internet.

Although you will now usually hear and read the term ActiveX, be prepared for slips of the tongue and pen (even perhaps in the Certification Exam, but hopefully not in this book!) that refer to OLE. Some older features of ActiveX, such as the OLE container control, actually still use the term OLE in their names.

COM provides the core services upon which ActiveX technology is based. The ActiveX technology provides the following services:

- Active Documents
- ActiveX Controls
- ActiveX Servers
- ActiveX Automation

ActiveX Documents enable developers to create applications that can host documents in their native format inside other ActiveX Documents. Chapter 14, "Creating an Active Document," provides more detail on ActiveX Documents.

Developers use ActiveX Controls to extend and enhance the services that an application provides. If you need word processing capabilities in your application, for example, you can purchase an ActiveX Control that provides the desired functionality. In addition, you can write your own controls that encapsulate whatever functions you need. Chapter 13, "Creating ActiveX Controls," provides more detail on ActiveX Controls.

ActiveX Servers are applications in the form of EXE or DLL files that expose a COM-compliant object model for use by programmers creating other applications. Chapter 12, "Creating a COM Component that Implements Business Rules or Logic," discusses how to create ActiveX Servers.

As a programmer, one of the most exciting things you can do with existing COM components is automation—the ability to declare an object within your application that uses the functionality of a different application (a COM component). This chapter focuses on how to use automation within your VB application to exploit the functionality of COM components.

A programmer might write a component specifically to be used in other applications (an ActiveX library or ActiveX control), but many standalone component applications can also expose objects.

Some examples of important standalone applications that also provide COM components would be the current versions of Microsoft Internet Explorer, Microsoft Excel, Microsoft Access, and Microsoft Word for Windows. In fact, every component of Microsoft's Office 97 Suite and later Office Suites is supposed to be a COM component.

A COM component contains an object structure that is visible to client applications. Objects belonging to the component but available outside the component are known as exposed objects. In order for an object to be an exposed object, it must be listed in the Windows Registry.

Exposed objects have their own methods and properties that a COM client, such as a VB application, can manipulate with standard object syntax.

Creating a Visual Basic Client Application that Uses a COM Component

Here are the main steps for using a COM component's exposed objects in your VB application:

1. Set a reference to the server in your application or know the server's name and its classnames.
2. Be familiar with the server's object members.
3. Declare an instance of the server object or objects that you want to use.
4. Manipulate the declared objects in your code through their members (properties, methods, and events).
5. Release the object instance from memory when your application is done with it.
6. Provide for error handling in case the COM component's code generates an error.

The following sections detail the steps to use a COM component's exposed objects.

Setting a Reference to a COM Component

Your application must contain a reference to a component's Object Library before it can use the component's object classes.

An Object Library is a collection of information about an application's objects. You can find this information in a .TLB (Type Library) file, an .OLB (Object Library) file, or in an ActiveX server's EXE or DLL file.

To locate the COM component's Object Library, check the component application's documentation to see where it stores its Object Library.

However if the component in which you're interested has been properly registered in the Windows Registry, you won't ever need to know the name or location of the Object Library. Instead, the Windows Registry will contain information about the component in which you're interested. In that case, all you need to do is refer to the Windows Registry entries for the component, as in the following description.

In order to make a COM component's object model available to your application, you should execute the following steps:

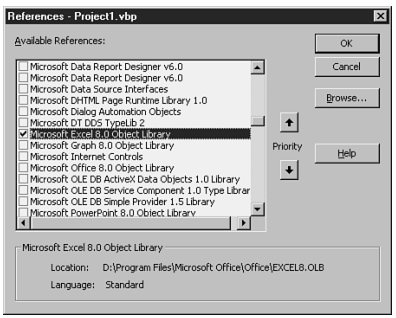


FIGURE 10.1
Use the References dialog box to set a reference in your project to a COM component.

STEP BY STEP

10.1 Making a COM Component's Object Model Available to Your Application

1. Choose Project, References from the main VB menu.
2. Scroll through the Available References list until you find the name of the COM component that you want (see Figure 10.1). The Object Library list you see here is not a VB-specific feature, but is generated from the Windows Registry.

3. Click on Browse to find and select the TLB, OLB, EXE, OCX, or DLL file containing the server's Object Library if the server you need isn't in the list (see Figure 10.2).
4. Select the desired COM component by clicking on the check box to the left of the server name.
5. Click on OK in the References dialog box.

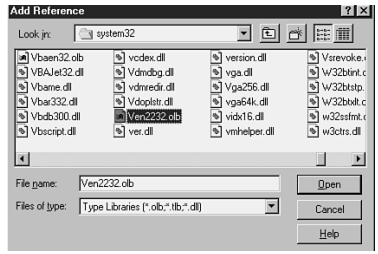


FIGURE 10.2
You can browse for the file containing a component's Object Library if the component isn't listed in the Available References list of the References dialog box.

Using the Object Browser to Find Out About a COM Component's Object Model

A COM component's class objects have characteristics similar to VB objects. You can manipulate their members (properties, events, and methods) as you would the members of VB controls and other objects.

You use the Object Browser to find out about a COM component's object model just as you would find out about the native objects provided by VB. To see the component's information in the Object Browser, you must first have set a reference to the component as described in the previous section, "Setting a Reference to a COM Component."

In order to use the Object Browser with a COM component, you must execute the following steps:

STEP BY STEP

10.2 Using the Object Browser with a COM Component

1. Access the Object Browser within VB by pressing F2 or the toolbar button (see Figure 10.3).
2. Select the COM component from the list of references in the Projects/Servers list.
3. Select a class from the Classes/Modules list.

4. Select a property or method from the Methods/Properties list. Notice the brief description of the method or property at the bottom of the Object Browser dialog to the right of the question mark (?) icon.

5. If the component's Object Library has a help file, you can click the question mark (?) icon on the Object Browser's Toolbar for more information about the selected method or property (see Figure 10.4).

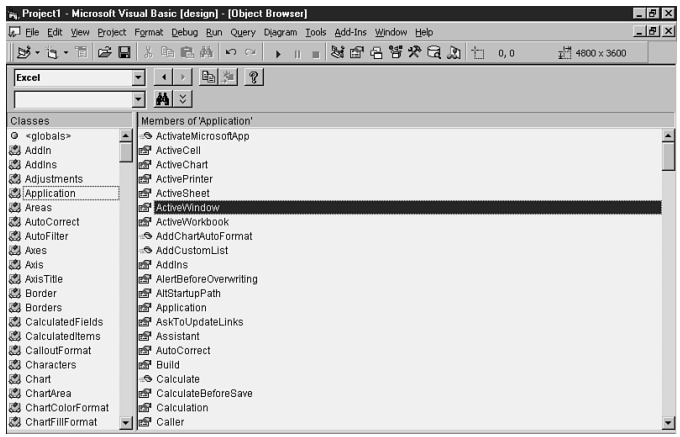


FIGURE 10.3▶
The Object Browser can give you detailed information about a COM component's objects and their methods and properties.

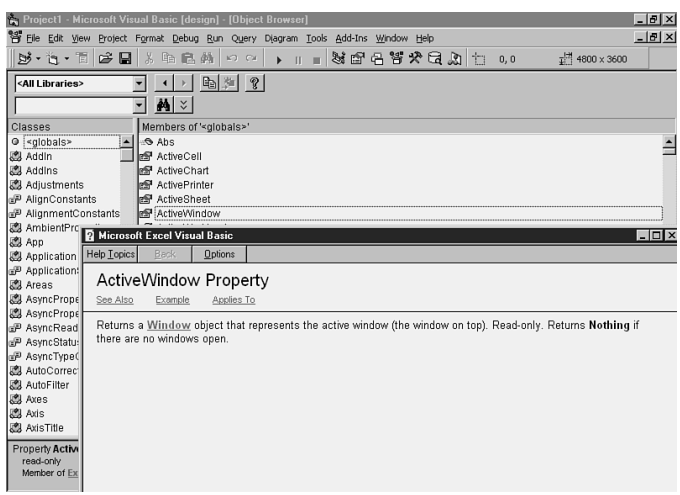


FIGURE 10.4▶
Clicking the ? icon in the Object Browser's Toolbar brings up an extended help screen for the selected item when you're browsing information about a COM component.

If you're not already familiar with the Object Browser, you will soon find it's a powerful tool that helps you intelligently and efficiently program with COM components.

Using the `New` Keyword to Declare and Instantiate a Class Object from a COM Component

After you've set a reference to a COM component in your application, you can instantiate objects from the component's classes in your application.

You may declare an object variable for a component class that you want to use. You can then use this object variable to point to an instance of the corresponding class.

The `New` keyword is Microsoft's preferred technique for instantiating object variables from server classes. You can use the `New` keyword in one of two ways, discussed in the following sections:

- Instantiate the object at the same time you declare it by using the `As New` keyword.
- Declare the object and instantiate it later with set / `New`.

Many COM components don't support the `New` keyword. Refer to each component's documentation to see whether it supports `New`.

If the component you want to use doesn't support `New`, you will need to use the `CreateObject` or `GetObject` function, as discussed in the section of this chapter entitled "Using the `CreateObject` and `GetObject` Functions to Instantiate Objects."

NOTE

VB Version Support for the `New` Keyword Components created in VB4, VB5, or VB6 always support the `New` keyword. See Chapters 12, 13, and 14 for a discussion of how to create ActiveX components in VB6.

Using `As New` to Instantiate an Object Variable When You Declare It

If the component whose class you want to instantiate supports the `New` keyword, all you need to do is declare a variable of appropriate scope (usually `Private` or `Public`), and the object is ready to use in your application. For example, if you want to use a class called `MyClass` from a component application called `MyComp`, your variable declaration in a General Declarations section might look like this:

```
Private objMyClass As New MyComp.MyClass
```

You could then manipulate the object's methods and properties and react to its events through your own application's code.

Using New to Instantiate an Object Variable After You Declare It

In the previous section, an object variable was instantiated at the same time it was declared. If you wanted to manage your object more tightly, you could wait until you actually needed to use the variable before instantiating it.

It's possible to declare an object variable without instantiating it and then instantiate it as needed with a combination of the Set statement and the New keyword, as illustrated in Listing 10.1.

LISTING 10.1

INSTANTIATING AN OBJECT VARIABLE WITH THE New KEYWORD AFTER YOU HAVE DECLARED IT

```
'General Declarations
Private objMyClass As MyComp.MyClass
.
.
.
'later in your code
Set objMyClass = New Mycomp.MyClass
```

This second method, as illustrated in Listing 10.1, is preferred over the method of the previous section (declaring the variable with As New). The As New declaration requires extra runtime checking of the variable type.

Late and Early Binding of Object Variables

You can implement class objects with object variables by declaring them as either generic object-type variables or variables of the specific type provided in the server's class library. For example,

```
Dim objMyClass As New MyComp.MyClass
```

or

```
Dim objMyClass As MyComp.MyClass
```

and

```
Dim objMyClass As Object
```

would all be valid declarations for an object that will later point to an instance of the `MyComp` component server's class named `MyClass`.

The first two declarations are preferable, however, as they provide *early binding* of the `MyClass` class in your application while the third, more generic declaration, is an example of *late binding*.

Binding refers to the point at which a system recognizes references to external objects in the compile-run cycle. For instance, you might misspell a declared object's property name in your code. Knowing whether the object is early- or late-bound will tell you when the system will detect the error.

If you've declared the object with early binding, the VB compiler can check the component's class library and catch any syntax errors before the application fully compiles and runs.

If, however, you use the `As Object` declaration to provide late binding, the VB compiler won't be able to check the component's class library and the compiler won't detect any syntax errors in the use of the object. Instead syntax errors with objects from the COM component will cause runtime errors in your application.

For instance, the Excel component server's most important object is the Application object. Application has a `Visible` property which, as you might expect, sets the object's visibility to a user. Suppose you have the following declaration in your code:

```
Dim objExcel As Excel.Application
```

or

```
Dim objExcel As New Excel.Application
```

And, later on, you have the following line:

```
objExcel.Visable = True
```

The compiler catches the misspelling of the `Visible` property's name as soon as you try to run the application during design mode or when you try to make an executable file.

If, however, you'd declared the object with the line

```
Dim objExcel As Object
```

the compiler wouldn't be able to check the syntax of

```
objExcel.Visable = True
```

NOTE

Additional Support for Early Binding in the IDE When you write code that uses early-bound object variables, you'll notice that the VB runtime environment is able to recognize the object model behind the variable by offering you a drop-down list of possible members whenever you type the variable's name followed by a period.

NOTE

When to Use Late Binding You don't need to set a reference to a component with the Project References dialog box if you're going to use late binding. For some components (those without an available Object Library, such as the versions of Microsoft Word through version 7.0), late binding is the only option because the application provides no Object Library to set a reference.

WARNING

Only Recent Versions of Excel Support Early Binding Early binding onlyworks with versions 7.0 and later of Excel. If your Excel component is a version earlier than 7.0, you must use late binding.

WARNING

Can't Use `As New` to Declare an Object With Late Binding If you use late binding with an `As Object` declaration, you can't use the `New` keyword when you instantiate the object variable. You must use `CreateObject` or `GetObject`, as discussed in the following section.

against Excel's class library. The error (`Visible` is misspelled) goes undetected until such time as VB attempts to execute this line, at which point the application would generate a runtime error.

You *must* use late binding if a component server's class library isn't available to you. For instance, Microsoft Word for Windows 7.0 and below doesn't provide a class library for its object classes. You must, therefore, always use the `As Object` syntax to declare a Word 7.0 object.

Using the `CreateObject` and `GetObject` Functions to Instantiate Objects

Because many COM components do not yet support the `As New` keyword, you will need to create and instantiate object variables for these components' classes with standard variable declarations and the `CreateObject` or `GetObject` functions.

The `CreateObject` and `GetObject` functions return a reference to a server class object. Use them with the `Set` keyword to assign their return values to a previously declared class object variable:

```
Set objExcel = GetObject(,"Excel.Application")
```

or

```
Set objExcel = CreateObject("Excel.Application")
```

The `CreateObject` function takes a single required argument, which is the name of the class you're instantiating. It always instantiates a new object in your application. It also takes a second optional argument which is a string representing the share name of the server where you can create a remote object.

`GetObject`

You can use `GetObject` to create an object from an already running instance of a server.

`GetObject` takes two possible parameters. You must always specify at least one of the two parameters:

- `GetObject`'s first parameter is a String giving the path and filename of a data file associated with the server application and its class.

- `GetObject`'s second parameter is the same as `CreateObject`'s single parameter: the name of the class you're instantiating.

There are several rules to keep in mind when using `GetObject`'s parameters:

- If you leave the first parameter completely blank (that is a single comma before the second argument), `GetObject` will always reference an existing object. If there is no existing object, a runtime error occurs.
- If you specify a valid filename in the first parameter and the file is of the type associated with the server application, you may leave the second parameter blank. `GetObject` will open the file with the associated server application. `GetObject` will use an existing reference to the object if it exists, or it will open a new copy of the object if none existed before in the application.
- If you specify a blank filename (" ") in `GetObject`'s first parameter, then you must specify the second parameter. `GetObject` will then always open a new copy of the object regardless of whether one already exists in the application.

The possible configurations of these two parameters are summarized in Table 10.1.

WARNING

Limitations on Support for `GetObject` The following discussion of `GetObject` applies to many Microsoft products and other COM components. However, the use of `GetObject` is application-specific. In fact, some ActiveX server applications don't support `GetObject` at all. You should refer to the application's documentation to find out whether `GetObject` is supported and, if so, what the proper syntax would be for using `GetObject`. If `GetObject` isn't available, you must always use `CreateObject`.

TABLE 10.1

POSSIBLE COMBINATIONS OF FILENAME AND SERVER.CLASS ARGUMENT SETTINGS FOR GETOBJECT FUNCTION

FileName	*Server.Class*	*Effect*
Blank	Blank	Not a possible combination.
Blank	Server.Class	Always uses an existing instance.
		Runtime error if there is no existing instance.
Empty String	Server.Class	Always opens a new instance of the server.
FileName	Blank	Opens server of type associated with FileName.
		Uses instance if Available otherwise opens new instance.
FileName	Server.Class	file specified with a new instance of the server.

Comparing `GetObject` and `CreateObject`

Since several exam questions usually rely on the confusion between `GetObject` and `CreateObject`, it is useful to emphasize the difference between these two functions:

- `CreateObject` always creates a new instance of the server object.
- `GetObject` can use a running instance of the server object but can also create a new instance depending on the syntax you use, as detailed in the following section.

Using a Component Server's Object Model

An object class belonging to a COM component may contain other classes or class collections. Subclasses or collection elements might repeat this nesting structure and could, in turn, contain other object classes and collections.

Only objects in the topmost level of the component's hierarchy can be created directly with `CreateObject`(). The subordinate objects in the hierarchy either already exist as subobjects of the higher objects or must be created with methods of the higher objects.

In ActiveX terminology, the highest-level objects are said to be `Public` and `Creatable`, whereas subobjects are `Dependent` or `Public` but `Not Creatable`. That is these subobjects can be seen by client applications, but the only way to reference them or create them is to go through the `Public` objects above them in the object model hierarchy.

For instance, one of the Excel component hierarchy's topmost objects is the `Application` object. An `Application` object can contain (among other objects) a collection object called `WorkBooks`, which in turn can contain individual `WorkBook` objects. Each `WorkBook` object of the collection can in turn contain a collection of `WorkSheet` objects, and each `WorkSheet` object can contain a `Range` object.

You would initialize the `Application` object as you've seen in the previous two sections:

```
Dim objExcel As Excel.Application
Set objExcel = CreateObject("Excel.Application")
```

You must always refer to the `WorkBooks` collection indirectly through the Application object, adding a new `WorkBook` to the `WorkBooks` collection with the Add method:

```
objExcel.Workbooks.Add
```

As an alternative, you can declare another variable to point to the newly added element of the `Workbook` collection. The Excel server contains a `Workbook` class. You can declare a variable of this class and use the `Set` keyword to assign the declared object variable to the results of the `Add` method:

```
Dim wb As Excel.Workbook
Set wb = objExcel.Workbooks.Add
```

You could add a `Worksheet` object to the `Workbook` object's Worksheet collection. A full example might look like Listing 10.2.

LISTING 10.2

USING THE EXCEL OBJECT MODEL TO MANIPULATE EXCEL OBJECTS

```
'General declarations section
Option Explicit
Private objExcel As Excel.Application
Private wb As Excel.Workbook
Private ws As Excel.Worksheet

Private Sub Form_Load()
    Set objExcel = CreateObject("Excel.Application")
    Set wb = objExcel.Workbooks.Add
    Set ws = wb.Worksheets.Add
End Sub
```

You can use the Object Browser or refer to a COM component application's documentation to find out about its object model hierarchy, as mentioned in the section of this chapter on the Object Browser.

Manipulating the Component's Methods and Properties

After you have instantiated an exposed object from a COM component's class in your application, you can programmatically manipulate it in the same way you would any other object in VB, such as a control or a form.

You must, of course, know how to use the object's methods and properties. If the component application has a class/type library, the Object Browser gives you a list and a brief description of each method or property and possibly more extended documentation if the component developer has provided a Help file along with the class library. You may also want to refer to any documentation on the component's classes as published by the component application's vendor.

The Excel component's various classes, for example, have methods and properties that enable you to manipulate an Excel application and every aspect of an Excel Spreadsheet or Chart. Assuming that `objExcel` is an instance of `Excel.Application` (see the previous three sections in this chapter), the following lines of code would terminate the running instance of Excel without prompting the user to save any changes:

```
Dim wb As Excel.Workbook
For Each wb In objExcel.Workbooks
    'consider workbook as saved - don't_prompt
    wb.Saved = True
Next wb
'End Excel application
objExcel.Quit
```

The Excel component's `Worksheet` class contains a subclass known as the `Range` object. The `Range` object uses a string argument that defines the range of cells to be manipulated. `Range`, in turn, has a `Value` property that you can either read or write. The effect of setting or getting the `Range` object's `Value` property is to read or set the cell value specified in the argument to the `Range` object. The following line of code assumes that the current VB form contains a `TextBox` control named `txtCellValue` and that there's a `Worksheet` object variable named `ws`:

```
txtCellValue.Text = ws.Range("A1").Value
```

This line of code would assign the contents of cell A1 to the `TextBox` control.

The following line of code would reverse the process, setting the contents of cell A1 to be the same as the contents of the `TextBox` control:

```
ws.Range("A1").Value = txtCellValue.Text
```

Releasing an Instance of an Object

It is a good practice to free the memory taken up by an object variable after you no longer need it. You can release the object by setting it to `Nothing`, as in the line

```
Set xl = Nothing
```

where `xl` is the name of the COM component's object variable.

Detecting Whether a Variable Is Instantiated

If an object variable holds a value of `Nothing`, then the variable doesn't currently hold an instance of any object. You should check an object variable to see if it holds `Nothing` before attempting to refer to it in your code with a line such as

```
If xl Is Nothing Then . . .
```

Although it might seem contorted, you will often see syntax such as the following in applications that use COM component objects:

```
If Not xl Is Nothing Then
    'Do some stuff with the xl object
End If
```

Handling Events From a COM Component

In order to handle the events of a COM component object instance, you must declare it using the `WithEvents` keyword. For example, you might declare an instance of an Access Report in a form's General Declarations with the line:

```
Private WithEvents RPT As Access.Report
```

In the Code Window for the form, you'd see RPT listed as one of the form's objects (see Figure 10.5). If you chose the RPT object from the left side drop-down list, then you'd see the list of procedures for the events supported by this object (see Figure 10.6).

FIGURE 10.5▶
The object you declared using `WithEvents` is visible in the list of objects for the module where you declared it.

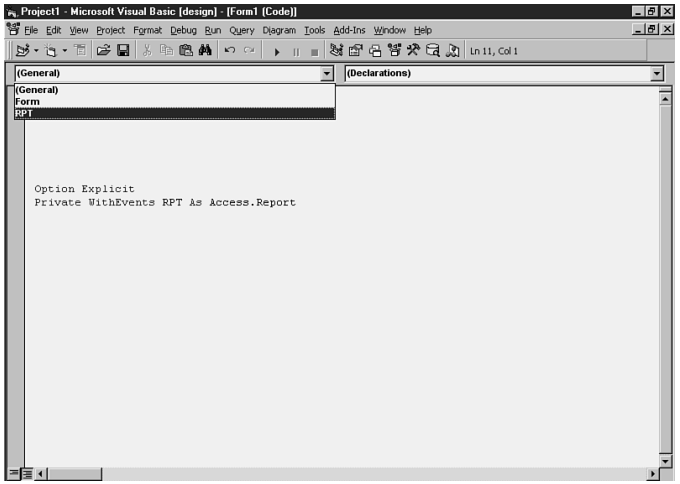
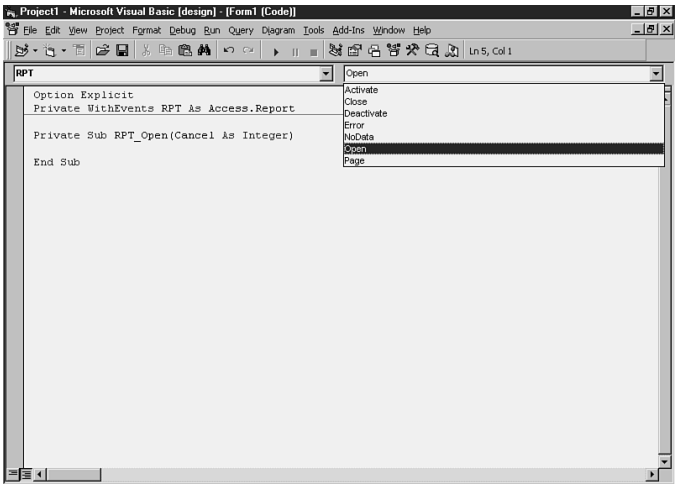


FIGURE 10.6▶
The object's event procedures are available to you when you declare the object using `WithEvents`.



There are several important limitations on programming using the `WithEvents` keyword:

- You can't use `WithEvents` keyword in a Standard Module (a .BAS file).
- You can only declare an Object variable using `WithEvents` in the General section of a module.

- You can't use the `As New` keyword in a declaration that uses `WithEvents`.
- Not all ActiveX component applications support the `WithEvents` keyword. In particular, only components whose object classes are registered in the Windows Registry will support `WithEvents` and then not even all of those components. Obviously, you can't handle events for those components that don't support `WithEvents`.

For a more detailed discussion of how to program with COM components' events, see the section "Handling a Class Event" in Chapter 12, "Creating a COM Component that Implements Business Rules or Logic."

CASE STUDY: A LOAN PROCESSING APP THAT USES EXCEL AND WORD

NEEDS

A loan department in a large bank has a complex set of operations it uses to process and track loans. Management would like to automate them more fully but wants to keep the processes flexible.

Existing processes are semi-automated with fairly standardized templates and management reports in Microsoft Office Suite applications, such as Excel and Word, and data stored in a SQL Server database.

Management would like to keep the flexibility of the Microsoft Office suite as a way of maintaining and enhancing their templates and reports, but training users to maintain them and other users to use them is becoming more difficult and costly as the system evolves.

REQUIREMENTS

From the scenario, we could identify the following requirements:

- Prevent accidental changes to template documents in Word and Excel to improve system integrity. Users should need as little awareness as possible of the management of these documents.
- Allow users to enter only standard, formatted information into Excel and Word documents. This would improve efficiency and data integrity.
- Provide an automated interface between the standard Excel/Word documents and the database to improve data integrity and system efficiency.

continues

CASE STUDY: A LOAN PROCESSING APP THAT USES EXCEL AND WORD

continued

- Automatically generate documents, such as routine client correspondence and loan papers, from the existing data and provide an automated interface to the user for doing this.

DESIGN SPECIFICATIONS

We might provide a solution based on automation of Word and Excel COM components coupled with SQL Server data access.

An outline of the solution could read as follows:

- Create a VB application with Excel and Word COM components.
- Create a VB user interface for each of the main functions to be automated.
- Use the respective COM component's object model to make a writeable copy of the template for the user, without user intervention, when a particular function calls for the use of one of the Excel or Word document templates.
- Gather input from the user in the VB interface (thus providing controlled input) and perform any necessary saving, routing, or printing in the VB application, once again using automation on the COM component.

CHAPTER SUMMARY

This chapter has covered the following key topics that you need to understand in order to master the certification exam objectives:

- Main steps to create a VB application that uses a COM component
- Setting a reference to a COM component
- Using the Object Browser to find out about a COM component
- Using the `New` keyword to declare and instantiate a COM component's class object
- Using `As New` to instantiate an object variable when you declare it
- Using `New` to instantiate an object variable after you declare it
- Late and early binding of object variables
- Using `CreateObject` and `GetObject`
- Using a COM component's object model
- Manipulating a COM component's methods and properties
- Releasing a COM object's instance and detecting whether an object variable is instantiated
- Using the `WithEvents` keyword
- Programming the events of COM components

KEY TERMS

- ActiveX
- Automation
- Class
- Component Object Model (COM)
- COM Client
- COM Component
- Early Binding
- Instance
- Instantiate
- Late Binding
- Object Browser
- Object Model
- Object Linking an Embedding (OLE)
- Reference Counting

APPLY YOUR KNOWLEDGE

Exercises

10.1 Defining and Initializing Object Variables

In this exercise, you look at how to define object variables and assign them to objects by using the `New` keyword.

Estimated Time: 10 minutes

You will create the following types of object variables:

- Define a variable called `objMyObject` of the type `Object`.
- Use the `New` Keyword to create a new `Form` Object and assign it to the object variable.
- Use the `Set` Keyword to create a new `Form` Object and assign it to the object variable.
- Add and remove the Data Access Object (DAO) Library reference from your project.

To perform these tasks, follow these steps:

1. Dim `objVar` as object. (The statement defines the variable that will hold the object reference.)
2. Dim `objVar` as New Form1. (This defines the variable and assigns a new object instance to the variable.)
3. Set `objVar =` New Form1. (Use the `Set` keyword to assign an instance of the object to the variable.)
4. Add the Data Access Object (DAO) Library reference to your project. (Choose the Project/References menu item. This menu opens the References dialog box, which enables you to add and remove the Object Library reference.)

10.2 Late Bound Variables Versus Early Bound Variables

In this exercise, you look at using an early and a late bound object variable.

Estimated Time: 20 minutes

The tasks to be performed are as follows:

- Use early binding to create and use Excel Application Objects.
- Use late binding to create and use Excel Application Objects.

To use early binding in your applications, a reference to the Object Library of the desired application is needed in your Visual Basic project. The Object Library to be used in this exercise is Microsoft Excel 8.0 Object Library. The same exercise can be performed with other Object Libraries using different Objects. To perform the first task, follow these steps:

1. Create a new Visual Basic application.
2. Add the Microsoft Excel 8.0 Object Library reference to the project.
3. Add a button (`btn1`) to the default form (called Form1). In the form, add the `Click` event code for the button for creating an early bound object as follows (note that the variable `objEarly` is defined as a module level variable):

```
Dim objEarly As Application
Private Sub btn1_Click()
'Create a new Application Object
Set objEarly = New Application

'Set the Object to be Visible
objEarly.Visible = True

'Create A new Workbook
objEarly.Workbooks.Add
```

APPLY YOUR KNOWLEDGE

```
    'Send some data
    objEarly.ActiveCell = "Hello World"
    End Sub
```

With the early binding approach, Visual Basic knows exactly what methods and properties an object supports because the object is defined as an `Application` object. With late binding, Visual Basic does not know what methods and properties an object supports; this information must be resolved using a slowing process. The `btn2 Click` event demonstrates the use of late binding in creating objects. The code for this event is as follows:

```
'General Declarations section
    Private objLate As Object

    Private Sub btn2_Click()

        'Create a new Application Object
        Set objLate = New Application

        'Show the Excel Window
        objLate.Visible = True

        'Create A new Workbook
        objLate.Workbooks.Add

        'Set Cell value
        objLate.ActiveCell = "Hello World 2"
    End Sub
```

Notice that with late binding, the object is defined as an `Object` data type. This allows the variable to have a pointer to any object type. With early binding, however, the object variable is defined as an `Application` object.

Therefore, the only types of objects that the object variable can refer to are `Application` objects. The use of early binding allows Visual Basic to perform data type, parameter, and syntax checking on methods and properties at compile time. With late binding, Visual Basic does not provide any syntax checking on methods and properties until runtime.

10.3 Creating Objects with the `CreateObject` Function

In this exercise, you look at creating objects with the `CreateObject` function.

Estimated Time: 15 minutes

The tasks to be performed are as follows:

- Use the `CreateObject` function to create early bound objects.
- Use the `CreateObject` function to create late bound objects.

The key to creating early or late bound objects is with the definition of the object variable. To create early bound objects with the `CreateObject` function, follow these steps:

1. Add an Object Library reference to the desired library reference using the References dialog box.
2. Define an object variable of the desired type, such as `Application`, `Workbook`, and so on. For example:

   ```
   Dim objVar as Excel.Application
   ```

> **NOTE**
> **Object Types and Object Libraries**
> Keep in mind that the object types will vary based on the type of object library you are using.

3. Use the `CreateObject` statement to create and return the object reference. For example:

   ```
   Set objVar = CreateObject
   ➥("Excel.Application")
   ```

To create a late bound object reference, no Object Library reference is needed. The late bound object reference can be created by following these steps:

APPLY YOUR KNOWLEDGE

1. Define the object variable as `Object` variable type. This can be done as follows:

   ```
   Dim objVar as Object
   ```

2. Use the `CreateObject` statement to create and return the object reference as follows:

   ```
   Set objVar = CreateObject("Excel.Application")
   ```

10.4 Obtaining Object References with the `GetObject` Function

In this exercise, you look at obtaining object references by using the `GetObject` function.

Estimated Time: 20 minutes

The tasks to be completed are as follows:

- Obtain an Excel 8.0 `Workbook` object reference by using the `GetObject` function using the file Ex0904.xls.
- Create an Excel `ChartObject` reference.
- Copy the chart to the Clipboard and paste it to an image control.

Figure 10.7 shows what happens when the application is executed.

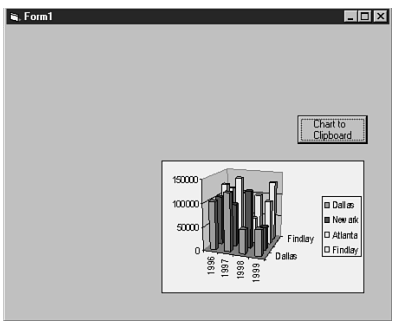


FIGURE 10.7
The sample application solution.

To add an Object Library reference to your project, follow these steps:

1. Select the Project/References menu. This will present the Object Library References dialog box. At this point you can add or remove Object Library references.
2. To open a file with the `GetObject` function, the following code can be used (assume that the variable has already been defined):

   ```
   Set objVar = GetObject(App.Path & "\Ex1104.xls")
   ```

3. After the object is returned, a `ChartObject` can be obtained from it by using the `ChartObjects` collection.
4. The Chart's graphic information can be copied using the `Copy` method of the `ChartObject`.

The code that creates the object variable and copies the `ChartObject`'s image to the image control is as follows:

```
Private Sub btn3_Click()
Dim objWkb As Workbook
Dim objWkSheet As Worksheet
Dim objChart As ChartObject

'Get the XL Workbook from the file
Set objWkb = GetObject(App.Path & "\Ex0904.xls")

'Get The first worksheet object
Set objWkSheet = objWkb.Worksheets(1)

'Get the first chart object
Set objChart = objWkSheet.ChartObjects(1)

'Check if Chart Object is valid
If objChart Is Nothing Then
    MsgBox "No ActiveChart Found"
Else
    'Copy Image to ClipBoard
    objChart.Copy

    'Past the Image Control
    img.Picture = Clipboard.GetData()
End If

End Sub
```

APPLY YOUR KNOWLEDGE

10.5 Using `WithEvents`

In this exercise, you demonstrate the use of `WithEvents` in a COM component's object declaration.

Estimated time: 30 minutes

The tasks to be completed are as follows:

1. Create a new EXE project with a standard form and add `CommandButtons`, `Labels`, and `TextBoxes` to the form, as described in Table 10.2 and illustrated in Figure 10.9.

TABLE 10.2

PROPERTIES OF CONTROLS TO PLACE ON FORM FOR EXERCISE 10.5

Control	*Property*	*Value*
Label	Name	lblSourceCell
	Caption	Source Cell
Label	Name	lblResultCell
	Caption	Result Cell
TextBox	Name	txtSourceCell
	Text	0
TextBox	Name	txtResultCell
	Text	<BLANK>
CommandButton	Name	cmdInitExcel
	Caption	Initialize Excel
CommandButton	Name	cmdRecalcResult
	Caption	Recalc Result
	Enabled	False

2. Set a reference to the Excel library as in the previous exercise.
3. Include the following code in the event procedures of the project's objects:

```
Option Explicit

Dim WithEvents xl As Excel.Application

Private Sub CmdInitExcel_Click()
    Dim Wb As Excel.Workbook
    Set Wb = xl.Workbooks.Add
    xl.Range("A1").Value = 0
    xl.Range("A2").Formula = "=A1/2"
    cmdRecalcResult.Enabled = True
    cmdInitExcel.Enabled = False
End Sub

Private Sub cmdRecalcResult_Click()
    xl.Range("A1").Value = txtNewValue.Text
End Sub

Private Sub Form_Load()
    Set xl = New Excel.Application
End Sub

Private Sub Form_Unload(Cancel As Integer)
    Dim Wb As Excel.Workbook
    For Each Wb In xl.Workbooks
        Wb.Saved = True
    Next Wb
    xl.Quit
    Set xl = Nothing
End Sub
```

4. In a Code Window, activate the drop-down list for Objects and note that the `xl` object appears in the list. Then select the object.
5. In the drop-down list for events, choose the `SheetCalculate` event procedure, as in Figure 10.8, and enter the following code:

```
Private Sub xl_SheetCalculate(ByVal Sh As
➥Object)
    MsgBox Sh.Name & " recalculated"
    txtResult.Text = xl.Range("A2").Value
End Sub
```

6. Now run the application and click the command button to initialize Excel. This button becomes disabled, and the second `CommandButton` becomes enabled.

APPLY YOUR KNOWLEDGE

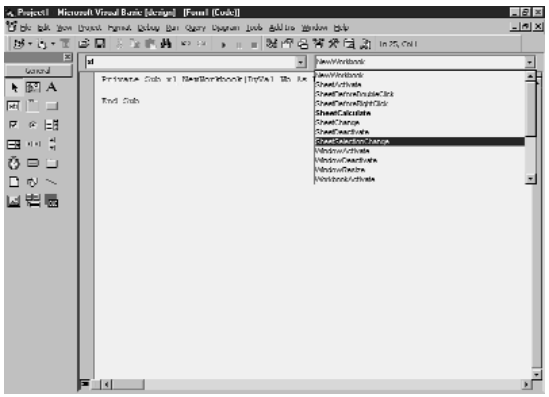


FIGURE 10.8▲
The `New` object and its event procedures.

7. Every time you change the contents of the source cell's `TextBox` and click the `Recalc` button, you will also see the message box, as shown in Figure 10.9, followed by a change to the resulting `TextBox`.

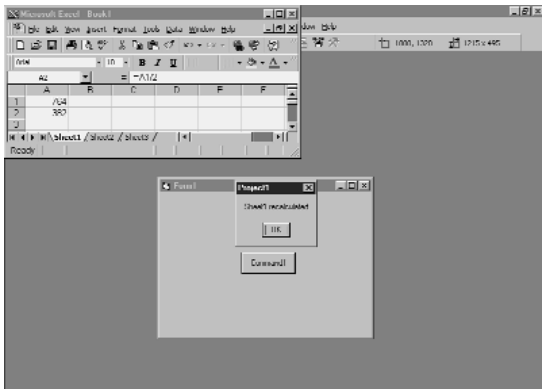


FIGURE 10.9▲
The form for Exercise 10.5.

Review Questions

1. What Visual Basic statement can be used to define an `Excel.Application` object using the `Dim` statement?
2. Write the statement(s) that will define an object variable and return a valid object reference of type `Excel.Application`.
3. Assume that you have a COM component with an application name of `MyApp` and an object called `Spreadsheet`. Write the statement that will create an instance of the `Spreadsheet` object.
4. If the following code is executed and the component is not running in memory, what occurs?

```
Set objInfo = GetObject(,"Word.Application")
```

5. Assume that you have a COM component with an application name of `MyApp` and an object called `Spreadsheet`. Write the declaration for a `Spreadsheet` object that will allow you to program with the object's events.

Exam Questions

1. Which types of files can contain COM components?

   A. (.EXE) Executables

   B. (.FRM) Visual Basic files

   C. (.DLL) Dynamic Linked Libraries

   D. (.BAT) Bat files

2. Which of the following statements will instantiate a new `WordDoc` object?

   A. `Dim objMyVar As WordDoc`

   B. `Dim objMyVar As New WordDoc`

APPLY YOUR KNOWLEDGE

C.
```
Dim objMyVar As Object
Set objMyVar = New WordDoc
```

D.
```
Dim objMyVar As Object
Set objMyVar = GetObject("App.WordDoc")
```

E. All these statements

3. The reference to Object Libraries provides Visual Basic with which of the following information?

   A. It is not used.

   B. A list of objects.

   C. Object types supported by the ActiveX Server and their properties and methods.

   D. Functions for creating object libraries.

4. Which of the following statements will define an object variable and return a valid object reference of type `Application`?

   A.
   ```
   Dim x as Object
   Set x = New Application
   ```

   B.
   ```
   Dim x as new Application
   ```

   C.
   ```
   Dim x as Application
   ```

   D.
   ```
   Dim X as Object
   Let X = new Application
   ```

5. Assume that you have a COM component with an application name of `MyApp` and an object called `Spreadsheet`. Which one of the following statements enables you to access that object?

   A.
   ```
   Dim objX as new
   CreateObject("SpreadSheet")
   ```

   B.
   ```
   Dim objX as Object
   Set objX = CreateObject("MyApp")
   ```

   C.
   ```
   Dim objX as Object
   Set objX =
   CreateObject("MyApp.SpreadSheet")
   ```

   D.
   ```
   Dim objX as Object
   Set objX = CreateObject("SpreadSheet")
   ```

   E. This cannot be done because the object reference library is not added to the project.

6. If the following code is executed and the server is not running in memory, what occurs?

   ```
   Set objInfo = GetObject("Word.Application")
   ```

   A. The application is loaded into memory and an object reference is returned.

   B. A runtime error occurs.

   C. The application will not be loaded because the path is not correct.

   D. The application is loaded into memory and nothing is returned.

7. Assume that you have a COM component named `MyApp` and an object class `MyClass` that supports events. The proper way to declare an instance of `MyApp.MyClass` so that you can program with its events is

   A.
   ```
   Private WithEvents objMine As
   MyApp.MyClass
   ```

   B.
   ```
   Private WithEvents objMine As New
   MyApp.Myclass
   ```

   C.
   ```
   Private objMine WithEvents As New
   MyApp.MyClass
   ```

   D.
   ```
   Private objMine WithEvents As
   MyApp.MyClass
   ```

8. When you program with an object class using `WithEvents` (pick all that apply):

   A. You must place the object's declaration in the General section of the module.

   B. You must write the event procedures "from scratch" in the General section of the module.

APPLY YOUR KNOWLEDGE

C. You cannot place the declaration in a Standard Code Module.

D. You must place code in all the event procedures that the object provides.

Answers to Review Questions

1. A variable of the type `Application` can be defined as follows:

```
Dim objApp as Application
```

If necessary you can define and create the object as follows:

```
Dim objApp as new Application
```

See "Late and Early Binding of Object Variables."

2. Two methods can be used to define and create the application object. These methods are as follows:

```
Dim x as Object
Set x = New Application
```

or

```
Dim x as New Application
```

See "Using `New` to Instantiate an Object Variable After You Declare It."

3. The `CreateObject` statement is used to create the instance of the object. This is done as follows:

```
Dim objX as Object
Set objX = CreateObject("MyApp.SpreadSheet")
```

See "Using the `CreateObject` and `GetObject` Functions to Instantiate Objects and Its Sub-Sections."

4. A runtime error will occur because an instance of the object is not running in memory. See "Using the `CreateObject` and `GetObject` Functions to Instantiate Objects and Its Sub-Sections."

5. You can declare an object for programming with its events as follows:

```
Private WithEvents xl as Excel.Application
```

See "Handling Events from a COM Component."

Answers to Exam Questions

1. **A, C**. Type Library references are usually found in Type Library files, Executables, or Dynamically Linked Libraries. For more information, see the section titled, "Setting a Reference to a COM Component."

2. **B, C**. Both B and C assume that there is a Type Library reference in the project. If the Type Library reference is absent, the application will not compile. The example in answer A is syntactically correct, but this does not instantiate an object. D is syntactically incorrect and would not work under any circumstance (`GetObject` requires a blank argument, blank string, or filename as its first argument). For more information, see the section titled, "Setting a Reference to a COM Component."

3. **C**. The Type Library will provide the information needed for early binding such as the object types, properties, and methods supported. For more information, see the section titled, "Manipulating the Component's Methods and Properties."

APPLY YOUR KNOWLEDGE

4. **A, B**. Answer A uses late binding to create the object. Answer B uses early binding to create the object. For more information, see the section titled, "Late and Early Binding of Object Variables."

5. **C**. The `CreateObject` statement requires a `ProgID` that is made up of the application name and the object name. For more information see the section titled, "Using the `CreateObject` and `GetObject` Functions to Instantiate Objects and Its Sub-Sections."

6. **B**. A runtime error occurs because the `GetObject` requires that an instance of the requested ActiveX Server is running on the machine. For more information, see the section titled, "`GetObject`."

7. **A**. Private `WithEvents objMine As MyObj.MyClass` gives the correct syntax for declaring an object variable to use its event procedures. The other choices are all syntactically incorrect and would cause compiler errors. In particular, you can't use the `New` keyword in a `WithEvents` declaration. For more information, see the section titled, "Handling Events from a COM Component."

8. **A, C**. `WithEvents` declarations must go into the General section of a module, and they cannot be placed in a Standard Code Module. B is incorrect because you can find an object's pre-loaded event procedures by using the drop-down boxes in the Code Window (just as you would for any control or Form). D is incorrect—although if it sounds familiar, that's because this is one of the rules for programming with a class object that uses the `Implements` keyword. For more information, see the section titled, "Handling Events from a COM Component."

OBJECTIVE

This chapter covers the following objective and its subobjectives:

Implement error handling for the user interface in distributed applications (70-175 and 70-176).

- **Identify and trap runtime errors.**
- **Handle inline errors.**
- **Determine how to send error information from a COM component to a client computer. (This subobjective is covered in Chapter 12, "Creating a COM Component that Implements Business Rules or Logic" in the section called "Handling Errors in the Server and the Client."**

▶ The subobjectives discussed in this chapter can be expanded as follows:

- To identify and trap runtime errors in an application, you must know how to set up an error trap within a procedure using a combination of the On Error statement, labeled locations, and the Resume statement. In addition you must know how to recognize and handle different types of errors. You must also know the consequences of not using an error trap in every procedure in the call stack.
- Inline error handling refers to a specific error-trapping technique that does not use a labeled location. With inline error handling, you write error-handling code in place immediately after the lines you expect to cause errors; you do not use the Resume statement.

CHAPTER 11

Implementing Error-Handling Features in an Application

Outline

Setting Error-Handling Options **479**

Setting Break on All Errors 480
Setting Break in Class Modules 480
Setting Break on Unhandled Errors 481

Using the `Err` Object **481**

Properties of the `Err` Object 482
Methods of the `Err` Object 485
Using the `vbObjectError` Constant 486

Handling Errors in Code **487**

Using the `On Error` Statement 487
Inline Error Handling 489
Error-Handling Routines 490
Trappable Errors 492

Using the Error-Handling Hierarchy **494**

Common Error-Handling Routines **496**

Using the `Error` Function **499**

Using the `Error` Statement **499**

Inline Error Handling **500**

Chapter Summary **502**

Study Strategies

- Create an application with at least one procedure that calls a second procedure. Put an error-handling routine in both procedures, as described in this chapter, and experiment with them.
- Experiment with various combinations of disabling error handling in the two procedures to familiarize yourself with the way errors are handled in the call stack.
- Write a routine that uses inline error handling and experiment with it.
- Check out the VB sample applications. There is a sample VB application included with the VB 6 installation that demonstrates error-handling techniques. It is called Errors.vbp and is in the Errors folder under the folder where your sample VB applications are installed. Experiment with it.

INTRODUCTION

This chapter discusses error handling in Visual Basic applications. Being able to trap, identify, and handle errors is a vital part of developing a user-friendly application. Error trapping and error handling can provide important feedback to the user about why an error occurred. The information provided through error handling can also help the developer find, correct, and prevent errors.

Trapping and handling errors in an Executable is covered in this chapter as well as dealing with errors at design time while debugging a project. You will learn different ways to trap errors when they occur and to handle different error situations. In this chapter, recommendations for creating common error handling procedures are provided along with examples of how error information should be presented to the user.

This chapter covers the following topics:

- Setting error handling options
- The `Err` object
- Handling errors in code
- The error handling hierarchy
- Common error handling routines
- The `Error` function
- The `Error` statement

SETTING ERROR-HANDLING OPTIONS

The first time you will encounter errors in your application is when you are still testing your application in the VB IDE.

No matter how careful you are in coding, you will always find something that you did incorrectly—or a condition you did not anticipate—the first time you run your application. *How* you want to handle these errors when you run in the development environment depends on several factors. You might want to stop your application and go into Debug mode every time your application generates a runtime error. You may also want to bypass runtime errors or allow any error handling that you have already coded to take control.

Visual Basic provides several different options for you to use in the development environment. These choices for handling errors impact your application only while you are in the IDE and do not affect an Executable created with Visual Basic. The choice you make depends on how complete your code is and the type of application you are developing.

Setting Break on All Errors

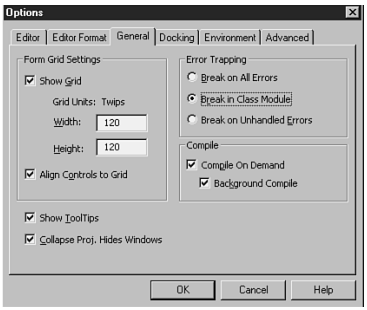


FIGURE 11.1
The error-handling options available.

Error-handling options are set in the development environment with the Options dialog box. From the Visual Basic menu, choose Tools and then Options. The Options dialog box appears. The error-handling options are on the General tab, as shown in Figure 11.1.

The first option for error handling in the development environment is to Break on All Errors. If this option is selected and a runtime error occurs in your application, the program will stop immediately. Execution will stop regardless of any error-handling procedures you have in your code (discussed later in "Handling Errors in Code"). The program stops regardless of whether the code that caused the error is in a class module that is part of your Visual Basic environment.

Remember that these options are in effect only when you are running your code in the Visual Basic development environment, not for Executables. Breaking on All Errors is usually used when you have very little error handling in your procedures and you expect some runtime errors to occur. It enables you to find and correct the errors while you run your project interactively.

Setting Break in Class Modules

The second option for handling errors at runtime is to Break in Class Modules when there is no error handling present. This option is used for projects that contain class modules, especially projects that are using ActiveX components created as other Visual Basic projects.

When debugging ActiveX components, it will be important for you to see the code that generates an error in the ActiveX project. With this option you need to be able to see the line of code that generated an error in a class module. If you did not use this option, errors generated in a class module would return back to the client program instead of breaking in the class module.

Breaking in class modules is different than breaking on all errors when error handling is in place. If Break in Class Modules is selected and error handling is in place for the code that generated a runtime error, the error handler would be executed and the continued program execution would depend on that error handler.

Setting Break on Unhandled Errors

The final option for error handling in the IDE is to Break on All Unhandled Errors. For this option if an error occurs and an error-handling routine is in place, that error handler will be run and execution will continue without entering Break mode. If no error-handling routine is active (see the section "Using the Error-Handling Hierarchy") when a runtime error occurs, the program will go into Break mode.

The difference between breaking in class modules and breaking on unhandled errors comes into play when the code that generated a runtime error is in a class module. In Break in Class Module option, if an unhandled error occurred in a class module, Visual Basic would enter Break mode at that line of code in the class module. In the Break on All Unhandled Errors option, if the unhandled error occurs in a class module, Visual Basic enters Break mode at the line of code that referenced the procedure in the class module.

Using the Err Object

Visual Basic uses the `Err` object to provide information to your application for error handling. The `Err` object can be used to retrieve information about the type of runtime error that has occurred. The information contained in the `Err` object can be used to determine how and whether your application should continue. It can also be used to present information to the user, if either the user can take some action to correct the problem or if error information has to get back to the developer.

This section defines and describes the properties and methods of the `Err` object. You can use the properties to retrieve information about an error that has occurred. They can also be set with other information that can be used by other procedures in an application. The `Err` object can be manipulated with the methods of the object. In some cases it is necessary to use these methods to pass information between modules.

Properties of the `Err` Object

The properties of the `Err` object identify an error by number, description, and source. These properties are set by whatever generated the error. Usually this is the Visual Basic application or an object used by the Visual Basic application. They can also be set by a Visual Basic developer in class modules to identify errors to external users of those class objects (see the section "Sending Information from a COM Component" in Chapter 12, "Creating a COM Component that Implements Business Rules or Logic").

After an error occurs, the properties of the `Err` object can be used to identify the problem and to determine what action, if any, should be taken to correct the error.

`Number` Property

The `Number` property of the `Err` object identifies an error by number. It is set by Visual Basic when a runtime error occurs. It can also be set in code by a developer to identify a user-defined error. The use of the `Err` object for user-defined errors are described later in this chapter (see "Sending Information from a COM Component" in Chapter 12).

When a runtime error occurs, the `Err.Number` property can be used to determine what that error was and how it should be handled. Some error numbers that Windows generates are described later in this chapter in the section titled "Handling Errors in Code."

The `Number` property contains a long integer value. Visual Basic error numbers and "user-defined" numbers (that is error numbers implemented by the programmer) range from 0 to 65,535. If you define any of your own errors in your application, use numbers below the 65,535 limit specified by Visual Basic. Also, be careful not to use a number already defined by Visual Basic. VB reserves numbers up through 512 for itself, so the practical range available to you for your own "user-defined" errors is 513-65,535 (see "Trappable Errors" later in this chapter).

VB documentation further recommends that you add the constant `vbObjectError` to your error code so that calling routines can definitely identify your error as other than a standard VB error. Microsoft is preparing us for the day when they will need to use errors between 513 and 65,535.

Description Property

A brief description of an error is available in the Description property of the Err object. The text corresponds to the error identified by Err.Number.

When a runtime error occurs in a Visual Basic application such as a division by 0 or an overflow, the Err.Number and Err.Description properties get set and can be used to handle the error or display information to the user. The easiest way to display error information is with a message box, as follows:

```
Msgbox Err.Number & "-" & Err.Description
```

Where you code this message box depends on how you have implemented error handling for your application (see the section "Handling Errors in Code").

As with the Err.Number, the Err.Description can be set in code. If you set the Err.Number property in your code, you should also set the Err.Description property to provide a description of the error. If you do not set the Description property and you use an error number recognized by Visual Basic, the description will be set for you automatically. If Visual Basic does not recognize the error, it sets the Description property to Application-Defined or Object-Defined Error. This is discussed in more detail later in this section, under the section titled "Raise Method."

Source Property

The Source property of the Err object is a string expression of the location at which the error occurred in your application. If an unexpected runtime error occurs, Visual Basic sets the Err.Source for you. If the error occurred in a standard module, the source will be set to the project name. If the error occurred in a class module, Visual Basic uses the project name and the class module name for the source. For errors in class modules, the source will have the form project.class.

Like the Number and Description properties, the Source property can be set in code. Even if Visual Basic generates the error and populates the properties of the Err object, you might want to overlay the source with a more descriptive text. Because Visual Basic uses only the project, class, and form names to create the source description, you may want to add to the source to specify a more exact location for the error. You could add the name of the procedure in which the error occurred, for example.

WARNING

Numeric Range For Your Own Error Numbers Implementing user-defined errors with numbers in the range between 0 and 512 (the range reserved for VB-defined errors) will cause confusion for other developers working with your projects. If you accidentally use an error number that already has meaning to Visual Basic, another developer will not know whether the error should be treated as a user-defined error or as a standard error. Even if you use a number in the range that is not defined in the current release of Visual Basic (513-65,535), it might be implemented in subsequent releases. To avoid confusion, it is best to implement user-defined error numbers in the 513 through 65,535 range, adding the constant offset amount vbObjectError to your error number as discussed above and in the section in this chapter devoted to vbObjectError.

The `Err.Source` property is typically used in an error-handling routine within a procedure or a common error-handling procedure for the application (see the section "Handling Errors in Code").

`HelpFile` Property

If you are generating errors within your code and you want to provide additional help—beyond the `Err.Description`—for the user, you could have a help file associated with the error information. The `HelpFile` property is a string expression that contains the path and filename of the help file.

The `HelpFile` property is used in conjunction with the `HelpContext` property described in the following section. Together they can be used to provide the user with optional help when an error message is displayed in a message box.

`HelpContext` Property

The `HelpContext` property of the `Err` object defines a help topic within the file identified by the `HelpFile` property. Together the two can be used to add additional help to a message box when an error is displayed to the user. An example of using a help file in conjunction with an error might look like this:

```
Dim Msg As String

Err.Number = 61 'Disk Full
Err.Description = "Your disk is full. Try another drive."
Err.HelpFile = "project1.hlp"
Err.HelpContext = 32  'context ID for a topic within the
➥help file

Msg = Err.Number & "-" & Err.Description & vbCrLf & _
       "To see more information, press F1."
MsgBox Msg, , "Error in Project1", Err.HelpFile,
➥Err.HelpContext
```

This displays a message box containing the error number, description, and the option to get help from the help file identified by the `HelpFile` and `HelpContext` properties, as shown in Figure 11.2.

FIGURE 11.2
An error message with a Help option.

`LastDLLError` Property

The `LastDLLError` property is a *read-only* property used by DLLs to return error information to Visual Basic applications. For versions of Visual Basic prior to 5.0, it is important to know that `LastDLLError` is only available in 32-bit Windows operating systems.

If an error occurs in a DLL, the function or sub-routine will signify the error through a return value or an argument. Check the documentation for that DLL to determine how errors will be identified. If an error does occur, you can check the `LastDLLError` property of the `Err` object to get additional information for that error. When the `LastDLLError` property is set, an error exception is not raised and error handling in your Visual Basic application will not be invoked.

WARNING

`LastDLLError` Only Available in 32-Bit Code Because the `LastDLLError` property is only available in 32-bit Windows operating systems, you must be cautious when developing code that will be conditionally compiled for both 16-bit and 32-bit environments. Make sure that you do not include references to `LastDLLError` in code that will be compiled for a 16-bit executable.

Methods of the `Err` Object

The `Err` object has two methods that you can invoke in your applications. These methods are also invoked automatically in Visual Basic applications as described in the following sections.

`Clear` Method

The `Clear` method of the `Err` object reinitializes all the `Err` properties. You can use the `Clear` method at any time to explicitly reset the `Err` object. Visual Basic also invokes `Clear` automatically in the following three situations:

- When either a `Resume` or `Resume Next` statement is encountered.
- At an `Exit Sub`, `Exit Function`, or `Exit Property` statement.
- Each time an `On Error` statement is executed.

`Raise` Method

The `Raise` method of the `Err` object is used to generate errors within code. You can use `Raise` to create your own runtime errors that will be used elsewhere in your application. The `Raise` method can also be used to pass error information from a class module to another application that uses objects of that class.

The arguments of the `Raise` method correspond to the properties of the `Err` object. They are as follows:

- **`Number`.** This is a required argument. It is a long integer that contains the error number. Remember that Visual Basic errors fall between 0 and 65,535, inclusive, and VB reserves error numbers 0-512 for itself. If you are defining any of your own errors, use numbers within the range 513-65,535.

Microsoft documentation also recommends adding the constant `vbObjectError` to your error code, as discussed below in the section on `vbObjectError` and in "Sending Errors from a COM Component" in Chapter 12.

- **`Source`.** An optional argument identifying where an error occurred. `Source` is a string property that can contain any information that will help point to the exact location of the problem. It may contain the class module name, form name, and procedure. The standard is to set the `Source` to `project.class`.
- **`Description`.** An optional argument describing the error that has occurred. If the description is not set, Visual Basic examines the `Number` argument to determine whether the error number is recognized (between 0 and 65,535). If `Number` does map to a Visual Basic error, the `Description` property is set automatically. If `Number` does not correspond to a Visual Basic error, the `Description` is set to Application-Defined or Object-Defined Error.
- **`HelpFile`.** Identifies a help file and a path to the file. This optional argument sets the `Err.HelpFile` property, which can be used with the `HelpContext` property, to provide help to the user. If `HelpFile` is not specified, the path and filename for the Visual Basic Help file is used.
- **`HelpContext`.** Used with the `HelpFile` argument, the optional `HelpContext` argument identifies a topic within the `HelpFile`. If the `HelpContext` is not specified, Visual Basic uses the help topic of the Visual Basic Help file corresponding to the `Number` argument (if a topic is available).

The `Raise` method is usually used within class modules. It allows you to generate your own runtime errors to pass information to another application using your application as a server. The `Raise` method is discussed further in the section titled "Sending Errors from a COM Component" in Chapter 12.

Using the `vbObjectError` Constant

When you return error information from a class module, you must be aware that some special handling is required. If you want to generate a

certain numbered error in a class module, you should add a constant number, `vbObjectError` (a number in the range of negative two billion on most systems), to your error number before you invoke the `Raise` method. You should do this to signal the error is generated in the class object but raised to the calling routine or application.

If you wanted to raise error number 500 in your class module, for example, you would invoke the `Err.Raise` method and pass `vbObjectError + 500` as the error number. The calling program or module interpreting the runtime error generated by the `Raise` method should then strip off the `vbObjectError` constant to interpret your error.

NOTE

Error Numbers That You Can Use
Currently not all the numbers between 0 and 66,535 are being used by Visual Basic. Some of the numbers are used; others are being reserved for future use. Using the `vbObjectError` enables you to define your own error numbers. It also prevents rewrite in the future—when later versions of Visual Basic that use more error numbers are released.

Handling Errors in Code

There are several ways to handle errors in your Visual Basic applications. The simplest but least effective way is not to do anything. Let the errors occur. Windows will generate error messages to the user, and the application will shut down. A better way to handle errors is to trap the errors with the Visual Basic statement `On Error`. By coding `On Error`, you are handling errors in a more graceful manner. You have the option of trying to correct the problem through code, to bypass the code that caused the error, or to shut down the application if the error is unrecoverable.

This section discusses the `On Error` statement and the different ways of using it in procedures. The discussion examines inline error handling and error-handling routines found at the end of procedures. Some trappable errors will also be covered with recommendations on how to use them and what to look for in code.

Using the `On Error` Statement

The `On Error` statement in Visual Basic identifies how errors will be handled for a particular routine. It can be used to turn on and turn off error handling for a procedure and, in some cases, sub-routines and functions called from the procedure in which it is coded (see “Using the Error-Handling Hierarchy”). An `On Error` statement instructs Visual Basic on what should be done if a runtime error occurs.

It is good practice to place error handling in every procedure. Generally it is especially important in any routine prone to errors. These include routines that process database information, routines that read from and write to files, and procedures that perform calculations. If an application contains code that relies on some outside events—such as a network connection being available or a disk being ready in a drive—there are always situations that are beyond the control of the developer. For these instances good error-handling routines are very important.

Different routines require different types of error handling. The syntax of the `On Error` statement can be coded several ways, depending on each situation.

Goto <line>

The first way to code the `On Error` statement is as follows:

```
On Error GoTo Main_Error
```

where `Main_Error` is a line label in a procedure. This is the most common use of the `On Error` statement and gives the developer the most control over error handling. A procedure using this format would look something like this:

```
Private Sub Main()
     On Error GoTo Main_Error
     ' ... some processing ...
     Exit Sub
Main_Error
     ' error handling code
End Sub
```

As in the preceding procedure, it is generally best to put the `On Error` statement as the first executable statement in a procedure so that any other lines of code will fall under the control of the `On Error` statement. In this example when an error occurs anywhere after the `On Error` statement, execution will continue in the error-handling code.

Error-handling code can contain any Visual Basic statements that can be coded elsewhere. It is important, however, to keep error-handling code simple to prevent additional errors from occurring. If a runtime error occurs in the error-handling code, then the new error will not be handled in the error handler. Instead, the error will be passed up the call stack until either an error handler is encountered, or until the error becomes a runtime error.

Resume Next

The second way of coding an `On Error` statement is with a `Resume Next` clause. With a `Resume Next` clause, the `On Error` would look like this:

```
On Error Resume Next
```

`Resume Next` tells Visual Basic that when a runtime error occurs, ignore the statement that caused the error and continue execution with the next statement.

GoTo 0

The last way to code the `On Error` statement is with the `GoTo 0` clause, as in the following:

```
On Error GoTo 0
```

This is different from the other `On Error` statements in that it disables rather than enables error handling for the current routine. Even if there is a line labeled "0", error handling will be disabled.

Inline Error Handling

Not all error handling occurs at the end of a procedure, and you are not limited to one `On Error` statement in each sub-routine or function you code. Sometimes the error-handling requirements change from the beginning of a routine to the end of a routine. There may be some sections of your code where you expect errors to occur—and don't care—and there are other lines where you do not expect errors and want to be warned when they occur.

You may have a procedure that reads information from a file, for example, does some processing, and exits. If the file is not found, you want to exit the routine. If any other error occurs, you want to display a message box. Your code could look something like this:

```
Private SubA()
   ' set the initial error handling to go to line SubA_Exit
   On Error Goto SubA_Exit

   Dim sFile as String
   sFile = "filename.dat"

   ' do some additional processing
```

```
    ' Reset the error handling to go to line SubA_Exit
    ' for the Open statement.
    On Error Goto SubA_Exit
    Open sFile For Input As #1

    ' Reset to original error handling
    On Error Goto SubA_Error
    ' continue processing...

SubA_Exit
    Exit Sub

SubA_Error
    ' error handling

End Sub
```

In this example, the `On Error` statement is used three times. First at the beginning of the sub-routine, the `On Error` statement is used to tell Visual Basic to go to the error-handling code at `SubA_Error` when an error occurs. Then before the `Open` statement, `On Error` is used again to say that if any error occurs to go to line `SubA_Exit` and then to leave the routine. Finally the `On Error` statement is used again to send errors to the `SubA_Error` line after the `Open` statement has run successfully.

The number of `On Error` statements in a procedure is only constrained by the limit on total lines and bytes of code for Visual Basic procedures. In addition, you can use any of the `On Error` types in the same procedure. You can use the `Resume`, `Resume Next`, `Goto <line>`, and `Goto 0` all in the same procedure, and each can be used several times to toggle between different types of error handling.

Error-Handling Routines

This chapter has already discussed the `Err` object and has also examined the options for the `On Error` statement in Visual Basic. In addition, the error-handling routines within sub-routines and functions have been mentioned. This section continues the discussion of error-handling routines in procedures and recommends some standards for their creation.

Error-handling routines are sections of code that are executed in the event of an error. They can range from being very simple with only a few lines of code to being very lengthy. Unlike inline error handling, using an error-handling routine enables you to localize all your exception processing in one place within a procedure.

Some things to remember when you code error routines within your Visual Basic procedures include the following:

- *Keep your error-handling routines simple*. If another error occurs in your error-handling routine, your application will generate a fatal error and shut down.
- *Error-handling routines are generally found at the end of a procedure*. Make sure that you put an `Exit Sub` or `Exit Function` statement before the error-handling routine so that during normal execution, your program does not fall through to the error-handling code.
- *Use the* `On Error Goto <line>` *statement to control program flow in the event of an error*.
- *Anticipate the types of errors that are common to the functionality of your code*. This will enable you to handle different errors in different ways. It could also prevent fatal errors from occurring (see "Trappable Errors" later in this section).
- *If you need to display error messages to users, provide information useful to them*. A user typically does not understand errors such as `Invalid File Format`. Instead provide a message that states `The file you selected is not an Excel spreadsheet.`
- *When you display error messages to the user, also include information that will help you—as the developer—track down errors.* Include the form, class, or standard module name. Identify the procedure in which the error occurred. Add any other additional information that will help you find and correct problems.
- *Consider logging errors.* You might wish to implement some sort of persistent record of errors (writing to a text file or database, for example). This will take the burden for error reporting off the user who could forget or misreport error messages. If you decide to log errors, however, be aware that logging routines themselves might cause runtime errors (since database and text file access are prime sources of runtime errors).
- *Whenever possible, give the user the chance to correct problems*. If they try to read a file from a floppy drive and forget to insert a disk, for example, provide an opportunity for them to retry the operation.

No matter how hard you try, conditions beyond your control will sometimes cause errors to occur in your application. Using some or all of the preceding tips will provide a more user-friendly environment for your users. These tips help provide more information to the user when an error occurs, and will also make it easier to track down and debug problems in an application.

Trappable Errors

In parts of your applications, you can anticipate the types of errors that can occur. If your application is reading and writing to disk files, for example, you know that you may encounter some file errors. Disks can be full, the user may not have placed a disk or CD-ROM in a drive, or a disk could be unformatted or corrupt.

Errors that you can anticipate and code for in your applications can be found in Visual Basic's Help file. If you search Help for "Trappable Errors," you will get a list of errors for which you can trap in your application (see Figure 11.3).

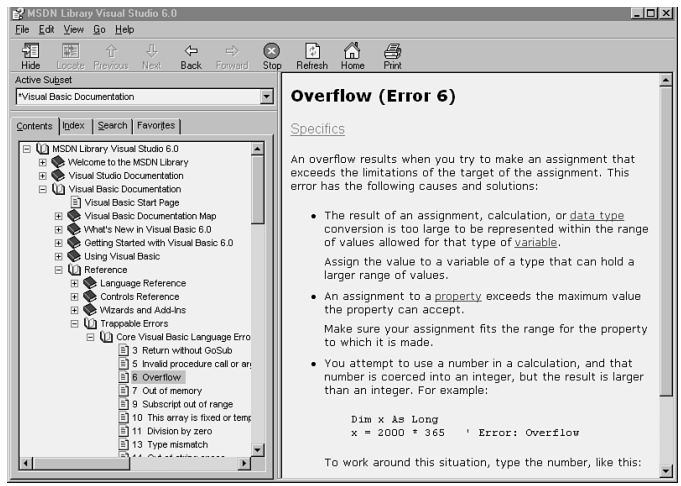


FIGURE 11.3
Visual Basic's Help for Trappable Errors.

Help will provide you with a description of the error and the number corresponding to that error. These will be the same number and description found in the `Err.Number` and `Err.Description` properties after the runtime error has occurred. If you select an error message from Help—as shown in Figure 11.3—you will get further information for that error, as shown in Figure 11.4.

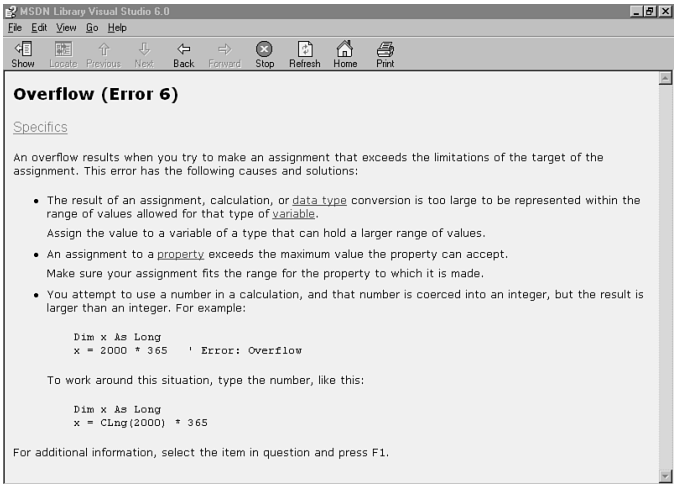


FIGURE 11.4
Detailed Help for Visual Basic errors.

It makes sense to anticipate and code for errors that could occur frequently in your application. If you know where errors may occur and what those errors are, you can handle them gracefully.

Instead of displaying an error message and ending your application when the user has not placed a disk in a drive, you may want to give the user a chance to use a disk or cancel what he or she was trying to do. You can trap for error number 71, `Disk not ready`, and prompt the user for his or her input. As part of your error-handling routine, you might include the following:

```
If Err.Number = 71 Then  ' Disk Not Ready
   If Msgbox("Disk not ready. Retry?", vbYesNo) = vbYes
➥Then
       Resume
   Else
       Exit Sub
   Endif
Endif
```

Now the user has a choice. If he or she places a disk in the drive and clicks on the Yes button, the process can continue. If the No button is clicked, Visual Basic will exit the sub-routine.

If a disk has not been formatted, you may want to give the user the option to format that disk. If a disk is full, you can let the user insert a new disk. Always be aware of the problems that can occur, especially because of user interaction, and code accordingly.

Using the Error-Handling Hierarchy

Unless you code error-handling routines in every procedure you write, Visual Basic needs a way of finding that error handler when a runtime error occurs. This section discusses the methods by which Visual Basic finds that error-handling procedure and what happens if one is not found.

Developers of Visual Basic programs control error handling with the `On Error` statement. The `On Error` statement dictates how a runtime error will be handled when encountered. If a procedure in your code is executing and an error occurs, Visual Basic looks for `On Error` within that procedure. If the `On Error` statement is found, the commands following `On Error` are executed. If an error handling does not exist in the procedure in which the error occurred, Visual Basic goes up in the calling chain to the code that called the current procedure to look for an `On Error` statement. Visual Basic continues going up the calling chain until some kind of error handling is found. If there is no procedure in the calling chain with error handling, the application ends with a fatal error.

The following example shows how errors will be handled in a simple code example:

```
Private Sub Main()
     Call SubroutineA
End Sub
Private Sub SubroutineA()
     Call SubroutineB
End Sub
Private Sub SubroutineB()
     Dim I as Integer
     Dim J as Integer
     I = 0
     J = 10 / I
End Sub
```

If you have these three procedures in your application, you can see that this code will generate a runtime error in `SubroutineB`. The statement `J = 10 / I` will generate a "divide by zero" error. When this application runs, sub-routine `Main` starts, and then it calls `SubroutineA`, which in turn calls `SubroutineB`. After the runtime error occurs in `SubroutineB`, Visual Basic must determine how to handle the error.

In this case, when the error occurs in `SubroutineB`, Visual Basic looks in `SubroutineB` for error handling specified by an `On Error` statement. Because it is not coded, Visual Basic goes up the calling chain to `SubroutineA` and checks there for error handling. Again because none is found, Visual Basic goes up the chain to `Main`. Because `Main` does not have any error handling either, a fatal error occurs and the application shuts down.

Now consider the case where sub-routine `Main` is modified to look like this:

```
Private Sub Main()
     On Error GoTo Main_Error

     Call SubroutineA
     Exit Sub

Main_Error:
     Msgbox "An error occurred."
     Resume Next

End Sub
```

When this code runs, a division by 0 still occurs in `SubroutineB`. Visual Basic still checks for error handling in `SubroutineB` and then in `SubroutineA`. When it is not found, Visual Basic checks `Main` for an `On Error` statement. This time it finds one and does whatever the error-handling statement instructs. In this case there is a statement instructing that execution continue at the label `Main_Error`. A message box is displayed, and then a `Resume Next` statement is encountered.

You must be aware of the point that a `Resume Next` or a `Resume` statement will send you to after the error occurs. A `Resume Next` statement will not send you back into `SubroutineB` to the statement after the divide by 0. It sends you instead to the statement in `Sub Main` after the line that made the call to the procedure that generated the error. In this case the `Exit Sub` statement would be executed next.

It is important to remember two important things about the error-handling hierarchy. The first is that Visual Basic will search up the procedure in the calling chain until an error handler is found. As an example look at a case where there are three procedures in the calling chain: `ProcA`, `ProcB`, and `ProcC`. `ProcA` calls `ProcB`, which in turn calls `ProcC`. If an error occurs in `ProcC`, Visual Basic will look in that procedure for an error handler. If one is not found, Visual Basic will go up the calling chain to `ProcB` and look for an error handler there.

Again if one is not found, Visual Basic will check `ProcA`. The second thing to remember is that if there is no error handler in the calling chain, a fatal error occurs and the application shuts down.

Common Error-Handling Routines

Sometimes while you are creating an application with error-handling routines in procedures, you may find yourself writing the same code over and over again. All your error-handling routines may be anticipating the same error numbers. Instead of duplicating the same error-handling code in many procedures, you can create a common error-handling function and call it from the individual error-handling routines within your other procedures.

Assume, for example, that you are developing an application that does extensive file processing. You find that many of your procedures check for errors 53 (`File Not Found`), 58 (`File Already Exists`), and 61 (`Disk Full`). Without a common error-handling routine, you might find yourself coding the following in every procedure that references files:

```
fErrorHandler:

    Dim msg As String

    Select Case Err.Number
        Case 53  ' file not found
            msg = "File not found. Do you want to try
➥again?"
            If MsgBox(msg, vbYesNo) = vbYes Then
                Resume
            Else
                Exit Function
            End If
        Case 58  ' file already exists
            msg = "The file already exists. Do you wish to
➥overlay?"
            If MsgBox(msg, vbYesNo) = vbYes Then
                Resume Next
            Else
                Exit Function
            End If
        Case 61  ' disk full
            MsgBox "The disk you specified is full. The
➥operation cannot continue."
            End
    End Select
```

Your error handlers can be even longer, trapping additional errors that your users might encounter.

Instead of repeating the same code many times in different procedures, you should move your error-handling code to a common error-handling function. You may find yourself not only using the common function many times within a project, but also find yourself including the common error handling in many projects you write. For this reason it is best to put your common error-handling function in a separate standard module so it can be easily ported to other projects.

When you create a common error-handling function, consider the following:

- *A common error-handling routine needs to know what error occurred*. The easiest way to pass this information is by passing the `Err` object to the function.
- *You need to know the results of the error handling*. Will execution continue? Should the line of code that caused the exception be bypassed or run again? This information can be passed back as the return value or as an argument of the function.
- *The error-handling function or module may be used in other projects*. Try not to reference forms or procedures specific to one project. This will prevent rewrites later.
- *Code the function to handle as many trappable errors as you can*. The more types of errors you trap for the more specific and user-friendly your error handling can be. Even if you do not code for all the trappable errors, you can add to the function at a later time.
- *Include handling of any errors for which you do not have specific error handling*. For example, your error handler may trap for errors 53, 58, and 61. Include some code to handle any other errors. Unexpected errors always occur in applications. In addition future releases of Visual Basic may include more trappable errors than the current release.

When you create an error-handling function, you must decide how to tell a calling procedure what should be done after the error handling has completed. Do you want the code to retry (`Resume`), bypass the code (`Resume Next`), or shut down the application?

One way to do this is to pass back the information through the return code of the function by using constants:

```
Public Const iRESUME = 1
Public Const iRESUME_NEXT = 2
Public Const iEXIT_PROCEDURE = 3
Public Const iEXIT_PROGRAM = 4
```

Remember to include the constant definitions in the BAS module that contains the error-handling function.

The error code that you previously had in many procedures throughout your application can now be combined into one error-handling function:

```
Public Function fErrorHandler(objError As Object) As
➥Integer

    Dim msg As String

    Select Case objError.Number
        Case 53  ' file not found
            msg = "File not found. Do you want to try
➥again?"
            If MsgBox(msg, vbYesNo) = vbYes Then
                fErrorHandler = iRESUME
            Else
                fErrorHandler = iEXIT_PROCEDURE
            End If
        Case 58  ' file already exists
            msg = "The file already exists. Do you wish to
➥overlay?"
            If MsgBox(msg, vbYesNo) = vbYes Then
                fErrorHandler = iRESUME_NEXT
            Else
                fErrorHandler = iEXIT_PROCEDURE
            End If
        Case 61  ' disk full
            MsgBox "The disk you specified is full. The
➥operation cannot continue."
            fErrorHandler = iEXIT_PROGRAM
        Case Else  ' covers all other errors
            msg = objError.Number & "=" &
➥objError.Description & _
                  vbCrLf & "Source = " & objError.Source
            MsgBox msg
            fErrorHandler = iEXIT_PROGRAM
    End Select

End Function
```

Now that your error handling has been centralized in a common function, the error handling in individual procedures can be simplified to something that looks like this:

```
sub1_error:

    Select Case fErrorHandler(Err)
        Case iRESUME
            Resume
        Case iRESUME_NEXT
            Resume Next
        Case iEXIT_PROCEDURE
            Exit Sub
        Case iEXIT_PROGRAM
            End
    End Select
```

For most of your procedures, the preceding code will be sufficient for handling errors. The common error function will be called, and the way of handling any given error will be returned. There are always times, however, where you will customize the error handling in different sub-routines and functions to handle unique situations. For these cases it is best not to alter a common error-handling function.

Using the Error Function

A function related to errors and error processing is the `Error` function. The `Error` function has one argument—an error number. It returns a string description of the error corresponding to that number. For example, if you coded the following:

```
Msgbox Error(61)
```

the user would see a message box with the error description `Disk full`, as shown in Figure 11.5.

FIGURE 11.5
An error message from the `Error` function.

If the error number is not specified, the description of the most recent runtime error is displayed (the same value as `Err.Description`). If there have been no runtime errors, the result is a 0 length string. If the error number is valid but has no Visual Basic definition, the return value will be Application-Defined or Object-Defined Error. An error will occur if the error number is not valid.

Using the Error Statement

The `Error` statement can be used to simulate or force errors to occur. The syntax is as follows:

```
Error <error number>
```

For example,

```
Error 51
```

generates an Internal Error.

The `Error` statement is available in Visual Basic 6 for compatibility with older versions. With Version 6, you should use the `Raise` method of the `Err` object.

INLINE ERROR HANDLING

Another form of the `On Error` statement is `On Error Resume Next`. This alternative is used to handle an error immediately after the line causing the error rather than branching to a specified error handler. Another use of `On Error Resume Next` is to simply allow your application to ignore errors that you don't need to handle at all.

In Listing 11.1, you can assume that the `GetObject` function might cause a runtime error. Notice that this code attempts to handle the error on the following line by first evaluating the current value of `Err.Number`.

LISTING 11.1

THE ON ERROR RESUME NEXT STATEMENT ALLOWS US TO HANDLE ERRORS AS THEY OCCUR

```
Private Sub XLInLine()
    Dim errTemp As Long
    On Error Resume Next
    Set xl = GetObject(, "Excel.Application")
    Select Case Err.Number
        Case 0
            MsgBox "Successfully Opened Excel"
        Case 429
            Set xl = CreateObject("Excel.Application")
        Case Else
            MsgBox "Fatal error #" & Err.Number _
            & " (" & Err.Description & ")" & _
            ": Passing the buck"
            End
    End Select
    Err.Clear
End Sub
```

The example also checks to see if the value of `Err.Number` is `0`, which indicates that no error has occurred.

Notice that the `Err` object's `Clear` method is called after the Select Case structure handles the error. You need to call `Err.Clear` because inline error handling doesn't use the `Resume` statement, and `Resume` is what you normally rely on to reset the `Err` object. If an error had occurred in the code in Listing 11.1 and `Err.Clear` hadn't been used, the `Err` object would continue to store information about this error. This could mislead a called procedure, which might have its own error handlers and would depend on the value of `Err.Number` to function properly.

Remember to call `Err.Clear` after each place where, in a routine where you handle inline errors, an error could occur in any of these locations, and you must handle each one individually.

Generally, you'll want to be careful about using inline error handling. Your procedures can get quite long and more difficult to construct and maintain since an error handler must follow each line that could cause an error.

As we mentioned previously, the `On Error Resume Next` statement is used for inline error handling. There are other times, however, when the `On Error Resume Next` statement comes in handy.

There are basically two uses for `On Error Resume Next`:

- Ignore errors and keep on processing. This is useful when you might expect an error to occur occasionally, but the error does not affect the remaining code. For example, the code in Listing 11.2 will process all the controls on the current form.

LISTING 11.2

USING ON ERROR RESUME NEXT TO IGNORE UNIMPORTANT ERRORS

```
On Error Resume Next
For each ctrlCurr In Controls
    ctrlCurr.Text = UCase(ctrlCurr.Text)
Next ctrlCurr
```

- Whenever the loop hits a control without a `Text` property, such as a `Label`, VB generates a runtime error. In Listing 11.2, the error is not relevant and does not require handling because you want VB to simply ignore any control that does not apply (controls without a `text` property). The `On Error Resume Next` statement tells the system to ignore the error and keep going.
- Process errors immediately after they occur. This is possible because VB sets the values of `Err.Number` and `Err.Description` when an error occurs even if `On Error Resume Next` is in effect. This enables you to use a second style of local error trapping—inline error handling—as previously discussed in this section.

Inline error handling therefore offers you an alternative, and often less cumbersome, way to process errors in a routine.

CHAPTER SUMMARY

KEY TERMS

- Compile error
- Error handling
- Error number
- Fatal error
- Syntax
- Syntax error
- Trappable error

This chapter covered the following topics:

- Setting error-handling options including Break on All Errors, Break in Class Modules, and Break on Unhandled Errors.
- Using the `Err` object including its properties (`Number`, `Description`, `Source`, `Helpfile`, `HelpFileContext`, and `LastDLLError`) and methods (`Clear` and `Raise`).
- Using the `vbObjectError` constant when returning error information from a class module.
- Constructing an error trap with various forms of the `On Error` statement.
- Constructing inline error handlers with `On Error Resume Next`.
- Writing error-handling routines.
- Using the error-handling hierarchy.
- Using the `Error` function to return descriptions of errors.

APPLY YOUR KNOWLEDGE

Exercises

11.1 The Error Processing Hierarchy

In this exercise, you create a simple application that illustrates the error hierarchy. You will create a project with multiple layers of sub-routine calls to see how Visual Basic finds error-handling routines. You will also code several `On Error` statements to handle errors instead of allowing Visual Basic to generate fatal errors and shut down your application.

Estimated Time: 15 minutes

1. Open a new project with a standard module and no forms. Make sure that you set the project options so that execution starts in `Sub Main`.
2. Create three sub-routines: `Main`, `SubA`, and `SubB`. In `Main` add a call to `SubA`. In `SubA` add a call to `SubB`.
3. In `SubB` you will want to force a runtime error. Dimension a variable as an integer. Then divide the variable by 0. For example:

```
Dim I as Integer
I = 1 / 0
```

4. Run the application. You should get a dialog box indicating division by 0. It will give you the option to end the application or debug. If this happened in an Executable, the application would shut down.
5. Stop the program. Edit sub-routine `Main` and add error trapping. Create an error-processing routine at the end of the sub-routine. Use `Main_Error` as the line label. Then add a message box that displays `"Error trapping in sub Main."`. Remember to put an `Exit Sub` before the `Main_Error` label. Your `Main` should look like this:

```
Sub Main()

    On Error GoTo Main_Error

    Call SubA

    Exit Sub

Main_Error:

    MsgBox "Error handling in sub Main."
    End

End Sub
```

6. Run the application again. This time you should get your own error message, not the fatal error displayed last time.
7. Now add error handling to `SubA` similar to that of `Main`.
8. Run the application again. Did you get the error message from `SubA` or from `Main`? Because Visual Basic searches up the calling chain for error handling, you will get the error message from `SubA`.

11.2 Trapping Specific Errors

In this exercise, you build on the code you created in Exercise 11.1. You will add error handling to trap for specific errors and code for each accordingly. You will also use message boxes to allow the users to make determinations on how execution will continue.

Estimated Time: 15 minutes

1. Using the project you started in Exercise 11.1, edit `SubB`. Add an error-handling routine at the end of the procedure:

```
Private Sub SubB()

    On Error GoTo SubB_Error

    Dim i As Integer
```

APPLY YOUR KNOWLEDGE

```
    i = 1 / 0

    Exit Sub

SubB_Error:

    ' error processing for SubB

End Sub
```

2. Now add a `Select Case` statement to the error handling. The first case you will test for is division by zero, error number 11 (see "Trappable Errors" in Visual Basic help).

```
' error processing for SubB

    Select Case Err.Number
        'If div by 0 error, handle here
        Case 11
            'Ask use to continue
            Dim iAnswer As Integer
            iAnswer = MsgBox("Division by
➥zero. Continue?", vbYesNo)
            If iAnswer = vbYes Then
                Resume Next
            Else 'user doesn't want to
➥continue
                End
            End If
        'If some other error, raise error
➥here
        'and it will be handled by calling
➥routine's
        'error handler
        Case Else
            Err.Raise Err.Number
    End Select
```

3. Run the application again. Was the error trapped correctly?

4. Edit `SubB` again. Add a statement after the division by zero to move a long integer value (that is 99,999) into the `I` variable. This will cause an overflow.

5. Add code to the error handler to trap for an overflow error. Use Visual Basic help to find the error number for an overflow.

6. Save and run the application again. When you get the error message for division by zero, make sure you continue processing to get to the line of code that creates the overflow condition.

11.3 Handling Errors Inline

In this exercise, you adapt the code created in Exercises 11.1 and 11.2 to implement inline error handling.

Estimated Time: 15 minutes

1. Using the project, that you started in Exercise 11.1 and modified in 11.2, edit `SubB` to perform inline error handling, as follows:

```
Sub SubB()
    On Error Resume Next
    Dim I As Integer
    'Enabled one of the following statements:
    'I = 1 / 0
    'I = 999999999
    If Err.Number <> 0 Then
        Select Case Err.Number
            Case 11
                MsgBox "Division by Zero"
            Case 6
                MsgBox "Overflow"
        End Select
        Err.Clear
    End If
End Sub
```

2. Notice the changes made to the routine from Exercise 11.2.

3. You've changed the `On Error GoTo` statement to `On Error Resume Next`.

4. Instead of creating a labeled error handler, you simply insert error-handling code after the line where you expect an error to occur.

5. You also have removed the `Resume Next` statements from the error-handling code of Exercise 11.2. This is because `On Error Resume Next` would not support `Resume` statements.

APPLY YOUR KNOWLEDGE

6. You also had to make a call to `Err.Clear` in order to clear the error condition. This was not required in the previous exercises because the `Resume` statements automatically clear the error condition.
7. Notice that there are two statements (both commented on in the example) that can cause an error. Run the application several times, first enabling one statement (division by 0) and then the other (integer overflow). You should see different error messages in the two cases because the error handler will recognize the difference in the error numbers.

Review Questions

1. What Visual Basic object is used to identify a runtime error that has occurred in an application? What property of that object describes the error?
2. If you are coding a class module and want to return error information through the `Err` object, what method of the `Err` object do you use to do this?
3. You code an application that starts in `Sub Main`. `Main` calls sub-routine `Sub1`. A runtime error occurs in `Sub1`. Neither `Main` nor `Sub1` has error handling. What happens to your application?
4. Early in the development stage of an application, you want to run and debug your code interactively. What is the best error option to choose so that execution stops every time a runtime error is generated?
5. What statement would you place in VB code to enable inline error handling?

Exam Questions

1. In your application procedure, `SubA` calls procedure `SubB`. A runtime error occurs in `SubB`, but that procedure does not have any error handling. What does Visual Basic do?
   - A. Generates a fatal error and shuts down the application
   - B. Checks `SubA` for an error handler
   - C. Displays a Windows error message and continues execution
   - D. Adds error handling to `SubB`
2. An error occurs in your application, and you want to display information in a message box. You will display the description of the error for your user, and you also want to display the error number so that you as the developer can identify the exact error that occurred. What two properties of the `Err` object do you want to use?
   - A. `Err.ErrDescription` and `Err.ErrNumber`
   - B. `Err.Desc` and `Err.Number`
   - C. `Err.Message` and `Err.Num`
   - D. `Err.Description` and `Err.Number`
3. What does the statement `On Error Goto 0` do?
   - A. Disables error handling
   - B. Causes a runtime error
   - C. Generates a syntax error
   - D. Causes execution to continue with the first statement in a procedure when an error occurs

APPLY YOUR KNOWLEDGE

4. Which three are options for handling errors at design time?

   A. Break On All Errors

   B. Break On Fatal Errors

   C. Break On Unhandled Errors

   D. Break In Class Modules

5. What Visual Basic constant should be added to custom error numbers returned from a server application with the `Raise` method?

   A. `vbCustomError`

   B. `vbErrorConstant`

   C. `vbUserError`

   D. `vbObjectError`

6. In Visual Basic 6, what is the best way to generate a runtime error?

   A. The `Error` statement

   B. The `Error` function

   C. `Err.Raise`

   D. Divide by zero

7. What will happen if an error occurs in a compiled VB executable application and Visual Basic does not find an error-handling routine?

   A. A message will be displayed, and execution will continue with the next line of code.

   B. A message will be displayed, and the application will end.

   C. No message will be displayed, and execution will continue with the next line of code.

   D. No message will be displayed, and the application will end.

8. What is the best error-handling option to use when debugging ActiveX components?

   A. Break On All Errors

   B. Break On Unhandled Errors

   C. Break In Class Modules

   D. None of these

9. In which environment can the `LastDLLError` property of the `Err` object be used?

   A. 16-bit only

   B. 32-bit only

   C. 16-bit and 32-bit

   D. Neither 16-bit nor 32-bit

10. Which of the following are valid options for the `On Error` statement?

    A. `Resume`

    B. `Resume Next`

    C. `Goto 0`

    D. `Goto <line number>`

11. If you are raising an error in your code and want to pass the location—such as the procedure name—at which the error occurred, which property would you use?

    A. `Err.Source`

    B. `Err.Context`

    C. `Err.Location`

    D. `Err.Procedure`

APPLY YOUR KNOWLEDGE

Answers to Review Questions

1. The `Err` object, instantiated by Visual Basic, provides information about runtime errors. The `Description` property gives a brief description of the error. See "Using the `Err` Object."
2. The `Raise` method of the `Err` objects enables you to return error information from a class module. The arguments of the `Raise` method specify the error number, description, source, and help information. See "Using the `Err` Object."
3. A fatal error occurs and your application shuts down. When an error occurs, Visual Basic searches up through the calling chain for an error handler. If none is found, a fatal error occurs. See "Using the Error-Handling Hierarchy."
4. The Break On All Errors option will stop execution in the IDE every time an error occurs, regardless of any error handling in place. This option is set on the General tab of the environment's Options dialog box. See "Setting Error-Handling Options."
5. To enable inline error handling, place the statement

   ```
   On Error Resume Next
   ```

   in your code. This will prevent VB's runtime error-handling system from taking over when an error occurs and will make you, the programmer, responsible for reacting to any error conditions that are generated. See "`Resume Next`" and "Inline Error Handling."

Answers to Exam Questions

1. **B.** Visual Basic will search up a calling chain to find an error handler. If `SubB` does not have error handling, Visual Basic will check `SubA` for an error handler. If no error handling exists in the calling chain, a fatal error occurs and your application ends. For more information, see the section titled "Using the Error-Handling Hierarchy."
2. **D.** `Err.Description` is a brief description of the error. `Err.Number` is the Visual Basic number corresponding to the error. For more information see the section titled "Using the `Err` Object."
3. **A.** `On Error Goto 0` disables error handling. For more information see the section titled "Handling Errors in Code."
4. **A, C, D.** The option that is not valid is Break On Fatal Errors. For more information see the section titled "Setting Error-Handling Options."
5. **D.** Use of the `vbObjectError` differentiates Visual Basic errors from user-defined errors.
6. **C.** The recommended way to generate runtime errors is with the `Err.Raise` method. The `Error` statement can also be used, but it is available in Visual Basic 6 only for the purpose of backward compatibility. For more information see the sections titled "Sending Information from a COM Component" in Chapter 12, and "Creating a COM Component" and "Using the `Error` Statement" in this chapter.

APPLY YOUR KNOWLEDGE

7. **B.** The user will see a message corresponding to the error that occurred and the application will stop executing. For more information see the section titled "Handling Errors in Code."

8. **C.** Because most ActiveX components consist of class modules, you usually want to use the Break In Class Modules option in the event an error does occur. For more information see the section titled "Setting Error-Handling Options."

9. **B.** The `LastDLLError` property of the `Err` object is only available in 32-bit environments. For more information see the section titled "Using the `Err` Object."

10. **B, C, D.** Only `Resume` is not a valid option for the `On Error` statement although it can be used within an error-handling routine. For more information see the section titled "Handling Errors in Code."

11. **A.** The `Err.Source` property is used to identify the location of an error. For more information see the section titled "Using the `Err` Object."

OBJECTIVES

This chapter helps you prepare for the exam by covering the following objectives and subobjectives:

Create a COM component that implements business rules or logic. Components include DLLs, ActiveX controls, and Active documents.

- A COM component provides reusable, encapsulated functionality to client programs running in a Windows environment. You can use VB to create one of the several types of COM components.

Choose the appropriate threading model for a COM component.

- A COM component's threading model affects the way that the COM component handles requests from clients and manages memory.

Compile a project with class modules into a COM component.

- **Implement an object model within a COM component.**
- **Set properties to control the instancing of a class within a COM component.**
- **Determine how to send error information from a COM component to a client computer.**

- Class modules are the backbone of COM object programming. You create a class module within a COM object and then implement its methods, properties, and events for clients to manipulate. You need to take special considerations into account to pass error information between COM components and clients.

CHAPTER 12

Creating a COM Component that Implements Business Rules or Logic

OBJECTIVES

Use Visual Component Manager to manage components.

- Visual Component Manager is an add-in provided through Visual Studio. You can use Visual Component Manager to manage source code or compiled components to make your components more available for reuse in future projects.

Create callback procedures to enable asynchronous processing between COM components and Visual Basic client applications.

- You can create special classes in COM components that COM clients can instantiate and pass to COM component instances. The special class instances can then be used to notify the client when particular events happen.

Register and unregister a COM component.

- If a COM component is registered in a machine's Windows Registry, client programs can instantiate objects from the component and manipulate them. You need to know how to manually register and unregister COM components.

Implement messages from a server component to a user interface.

- This objective is a subobjective of the objective "Implement online user assistance in a distributed application." The other subobjectives for online user assistance are discussed in Chapter 7, "Implementing Online User Assistance in a Distributed Application." This single subobjective is discussed here for better logical continuity.

> **NOTE**
> **MTS Objective in Chapters 14 and 15**
> This chapter does not cover one of the objectives listed under the COM component heading in Microsoft's published certification exam objectives list. That objective, "Design and create components that will be used with MTS," is covered in Chapter 16.

OUTLINE

Overview of COM Component Programming 513

The COM Specification and the ActiveX Standard 514

Comparing In-Process and Out-of-Process Server Components 515

Steps in Creating a COM Component 517

Implementing Business Rules with COM Components 518

Implementing an Object Model with a COM Component 519

Implementing COM Components Through Class Modules 520

The Uses of Class Modules 520

Starting a Class Module in a Standard EXE Project 520

The Class Module `Name` Property 520

Implementing Custom Methods in Class Modules 523

Implementing Custom Properties in Class Modules 524

Implementing Custom Events in Class Modules 528

Built-In Events of Class Modules 530

Using `Public`, `Private`, and `Friend` 532

Storing Multiple Instances of an Object in a Collection 533

Declaring and Using a Class Module Object in Your Application 538

Managing Threads in a COM Component 541

OUTLINE

Managing Threads in ActiveX Controls and In-Process Components **542**

Managing Threading in Out-of-Process Components 542

The `Instancing` Property of COM Component Classes **544**

Using Private `Instancing` for Service Classes **545**

Using `PublicNotCreatable` Instancing for Dependent Classes 545

`Instancing` Property Settings for Externally Creatable Classes 546

Deciding Between `SingleUse` and `MultiUse` Server Classes 549

Handling Errors in the Server and the Client **549**

Passing a Result Code to the Client 550

Raising an Error to Pass Back to the Client 551

Managing Components with Visual Component Manager 552

Storing VCM Information in Repository Databases 553

Making VCM Available in the VB IDE 553

Publishing Components with VCM 555

Finding and Reusing Components with VCM 559

Using Interfaces to Implement Polymorphism **561**

Steps to Implement an `Interface` Class 563

Providing Asynchronous Callbacks **572**

Providing an Interface for the `Callback` Object 574

Implementing the `Callback` Object in the Client 575

Manipulating the `Callback` Object in the Server 577

Registering and Unregistering a COM Component **579**

Registering/Unregistering an Out-of-Process Component 579

Registering/Unregistering an In-Process Component 580

Sending Messages to the User from a COM Component **581**

Managing Forms in an Out-Of-Process Server Component **581**

Managing Forms in an In-Process Server Component **582**

Choosing the Right COM Component Type **583**

Implementing Scalability Through Instancing and Threading Models **584**

Under-the-Hood Information About COM Components **585**

Chapter Summary **587**

STUDY STRATEGIES

- Create a COM component that implements business rules or logic. Components include DLLs, ActiveX controls, and active documents.
- For the first general objective ("Create a COM component that implements business rules or logic"), become familiar with the material discussed in this chapter's major sections: "Steps in Creating a COM Component," "Implementing Business Rules with COM Components," "Implementing an Object Model with a COM Component," and "Implementing COM Components Through Class Modules." Do Exercise 12.1.
- For the objective that refers to threading models, see the section titled "Managing Threads in a COM Component" and Exercise 12.2.
- Choose the appropriate threading model for a COM component.
- Compile a project with class modules into a COM component.
- Implement an object model within a COM component.
- Set properties to control the instancing of a class within a COM component.
- Determine how to send error information from a COM component to a client computer.
- For the objective and subobjectives listed under "Compile a project with class modules," see the sections titled "Declaring and Using a Class Module Object in Your Application," "The Instancing Property of COM Component Classes," and "Handling Errors in the Server and the Client." Review Exercise 12.1 and do Exercises 12.3 and 12.4.
- For the objective "Use Visual Component Manager to manage components," see the section and subsections under "Managing Components with Visual Component Manager." Complete Exercise 12.5.
- For the objective that refers to callback procedures, see the section titled "Providing Asynchronous Callbacks." Complete Exercise 12.6
- For the objective that refers to registering and unregistering a COM component, see the section titled "Registering and Unregistering a COM Component." Complete Exercise 12.7.
- For the objective that refers to implementing messages from a server component to a user interface, see the section titled "Sending Messages to the User from a COM Component." Complete Exercise 12.8.

INTRODUCTION

An object is *externally creatable* when an application that can serve as an ActiveX client can declare an instance of (that is, *instantiate*) that object as a variable and then manipulate the methods and properties of that object.

You can register your server component in the Windows Registry in one of several ways. These ways are discussed in more detail in the section titled "Registering and Unregistering a COM Component."

A Visual Basic project provides an externally creatable object through certain project settings and by possessing a class module whose properties you have set appropriately.

An externally creatable object also typically provides a gateway to other objects that can't be directly created by clients. Every COM component provides at least one externally creatable object. The server component's externally creatable objects, together with the other objects indirectly exposed by the externally creatable objects, is known as an *object hierarchy* or *object model*.

You can implement your own COM component's object model with custom object classes. Although you can create custom object classes specific to a single application, Microsoft's published documentation and courseware are full of examples of classes used to implement COM components.

NOTE

Getting Information About a COM Component Developers of potential client applications can find out about your server component by looking at the Windows Registry, or by using utilities such as VB's Object Browser.

OVERVIEW OF COM COMPONENT PROGRAMMING

Recall the syntax you saw in the preceding chapter for instantiating ActiveX objects in a client application:

```
Dim xl as Excel.Application
```

and

```
set xl = CreateObject("Excel.Application")
```

or

```
set xl = GetObject(,"Excel.Application")
```

Some server components (including any server components that you create with VB) support Microsoft's recommended standard, `Dim As New`. Assuming you had created a server component `MyServer` with a class `MyClass`, you could set up an object variable reference to the server component class with the line:

```
Dim objSvr As New MyServer.MyClass
```

This is the only line of code you would need, because the `As New` keyword declares and instantiates the object variable in a single line of code.

A somewhat more resource-efficient way to use the `New` keyword would be to declare the object variable without the `New` keyword, only using `New` when you're ready to use the object in code:

```
Dim objSvr As MyServer.MyClass
'Later in code:
Set objSvr = New MyServer.MyClass
```

This means that the resources to instantiate the object would not be required until the time that the object was actually needed.

The common thread in each of these sample lines is the *`servername.classname`* syntax you use when referring to your server component class.

The preceding statements that reference the system name of the server component and its class (`Excel.Application` or `MyServer.MyClass`) assume that you have already set a reference to the server component in the client project.

In this chapter, you see how to create, test, and maintain server components that can behave as `Excel.Application` did in the examples from Chapter 13.

The COM Specification and the ActiveX Standard

Microsoft now uses the term *ActiveX* as the name for its standard for enabling objects to be instantiated and communicate with each other in a Windows environment. ActiveX is an extension of the older OLE standard. The main difference between ActiveX and OLE standards is that Microsoft markets ActiveX as a technology for providing Internet-enabled, docucentric computing solutions.

You will still see the term *OLE* used in Microsoft documentation and in the names of many of its internal object types and events. This is because the OLE standard is still one part of ActiveX, the part that provides object linking and embedding. In fact, OLE stands for "object linking and embedding."

ActiveX and OLE, in turn, are based on a more generalized standard or specification known as the Component Object Model, or COM. The COM specification lays out the general blueprint for any object standard and defines how separate object components can communicate with and manipulate each other.

Comparing In-Process and Out-of-Process Server Components

VB6 enables you to create two types of COM components:

- **Out-of-process server components**. Out-of-process server components are always EXE files.
- **In-process server components**. In-process server components are always DLL files.

The difference between these server component types is with respect to the client application.

When a server component runs out-of-process with respect to its client, it has the following characteristics:

- It can run as a standalone application completely apart from the client.
- It does not share the same address space under the operating system.
- Its public creatable classes can be instantiated either as `SingleUse` or `MultiUse` objects. (See "The Instancing Property of COM Component Classes" later in this chapter for a discussion of these concepts.)

When a server component runs in-process with its client, it has these important features:

- The server component and client share the same executable space under the operating system.
- The server component and client share some of the same memory.
- An in-process server component's public creatable classes can be instantiated only as `MultiUse` objects. (See "The Instancing Property of COM Component Classes" later in this chapter.)

As you might imagine, there are pros and cons to both in-process and out-of-process server components.

Performance: In-Process Server Components Win the Contest

Because an in-process server component shares the same process space with its client at runtime, communication between the in-process server component and its client can avoid the ActiveX interface. The client can therefore call the in-process server component very efficiently.

Flexibility and Availability: Out-of-Process Server Components Win the Contest

Here is a list of things only an out-of-process server component can do:

- Provide classes that are either `SingleUse` or `MultiUse`. (See "The Instancing Property of COM Component Classes" later in this chapter.)
- Display modeless forms.
- Use the `End` statement (although not recommended in every application because it prevents the firing of an `Unload` or `Terminate` event).

Steps in Creating a COM Component

To implement a server component application, you must define classes that will become the server component's exposed objects.

In this section, you learn the basic steps for creating a server component and implementing classes in the server component. Later in this chapter, you also see how to test, fine-tune, maintain, and distribute your server component.

To create a server component application, follow these basic steps:

1. Start a new project for the server component class. Use the ActiveX DLL template for in-process component or EXE template for out-of-process component.
2. Change the project's name to match the name you want to give your server component application. This choice of name is important, because it is the name that the Windows Registry will use to identify the server component and the name that clients will use when they instantiate objects from the server component with the *`servername.classname`* syntax.

For each object class that you want to implement in the server component, follow these steps:

1. Add a class module to the server component project.
2. Set the class module's `Name` property to match the name you want to give the class.
3. If the class is to be externally creatable, make sure that the class module's `Instancing` property is set to `SingleUse`, `MultiUse`, `GlobalSingleUse`, or `GlobalMultiUse`. (The ActiveX DLL or ActiveX EXE automatically sets this property, but you should check it anyway.) If the class should not be directly created by clients, choose one of the other `Instancing` property's options (see the following section titled "The Instancing Property of COM Component Classes").

4. Implement the class object's members (methods, properties, and events) according to the guidelines given in the sections under "Implementing COM Components Through Class Modules" later in this chapter.
5. Create the other classes you need to round out your object's functionality.

Implementing Business Rules with COM Components

▶ Create a COM component that implements business rules or logic. Components include DLLs, ActiveX controls, and active documents.

In a multitier business solution, the most common three tiers are as follows:

- User-interface tier
- Business-rules or business-logic tier
- Data-access tier

The user-interface tier typically resides on each user's workstation; the business-rules or business-logic and data-access tiers more often reside on a network server.

You can use ActiveX with VB to create COM components that implement business logic in the solutions that you develop.

The objects provided by COM components to implement business logic are often known as *business objects*. You might, for example, provide a business object to the system in the form of a COM component that gives credit information and performs credit validation for customers.

When designing your system, you need to determine where components for the business-logic tier should reside and what the best form of implementation should be.

The first question to ask when designing a business-logic tier is this: "How should the tier be implemented?" Some possibilities might include the following:

- Implement as part of the user interface (compiled into that tier's executable). Although this might be the easiest to program in the beginning, it would quickly become a maintenance nightmare. Every time that your company changes the way it does business, you would need to make a change to the application and then distribute it to all the users. An email with a note to please rerun the install would probably not be sufficient—if a business rule changes, it should change for all users at the same time; it would be hard to count on all users to implement the change at, say, 12:00 a.m. on a particular date.
- Implement as part of the system's data structure (a trigger or a stored procedure). This would eliminate the problem of having to update every user's workstation. Such a choice will almost certainly confuse the functions of any data tiers and the business-logic tier, however. The business-logic tier would not be a separate unit and so couldn't be easily separated from the data when needed. This solution also impedes scalability. What would happen, for instance, if the data itself were someday split into more than one database? Business rules would have to be distributed among those databases. If any business rules were duplicated between the databases, they would have to be maintained in two places.
- Implement as a COM component on a server. This solution would provide the most long-term stability for maintainability and scalability. It would guarantee that future changes to the business rules would only require maintenance in one place. It would also guarantee that upward scaling of the system to accommodate more users or increased resources for reasons unrelated to business rule changes would not require changes in the component or its implementation.

Implementing an Object Model with a COM Component

▶ Implement an object model within a COM component.

To implement an object model in a COM component, you typically would use one or more classes of objects.

- Define the major real-world objects that the component needs to model. These major objects will become the classes of the COM component.
- Define the attributes and behaviors of these real-world objects. The attributes will become properties of the classes and the behaviors will become methods or events.
- Identify any repeating groups of objects from among the attributes of the class—that is, groups of subobjects that can belong to the class and whose number might vary. An employee can have one or more previous jobs, for example, so `PrevJobs` could be an attribute of an `Employee` class. Such repeating groups of objects can be implemented in `Collection` objects, which in turn contain zero or more individual members known as `Dependent` objects.

An `Employee` class in a `Human Resources` COM component might have properties such as `HireDate`, `Salary`, and `VacationTimeAccrued`; methods such as `Promote`, `TakeVacationTime`, `Hire`, and `Fire`; and events such as `VacationAccrualChange`.

Implementing COM Components Through Class Modules

▶ Compile a project with class modules into a COM component.

To create COM components in VB, you must first know how to create and manipulate a class module. This section and the following subsections discuss the basics of VB class modules and show how to use objects created from classes in a standalone (non-ActiveX) project. The information in these sections is referenced again during the VB ActiveX programming discussion in later sections.

The Uses of Class Modules

You use class modules to implement your own custom object variable types in VB. Just as VB provides the programmer with `TextBox`, `Form`,

`CommandButton`, or `RecordSet` object types, so you can provide other programmers (or yourself) with, say, `Employee`, `FileLocation`, or other object types. And just as the predefined object types have their particular properties, events, and methods, so can you define custom properties, events, and methods for your custom objects.

Each custom object type that you wish to define must have a class module behind it. You can use class modules in two different environments:

- **As part of an ActiveX component server**. As part of a server, your class gets registered in the Windows Registry and other applications can create object instances of it.
- **As part of a standard EXE**. Code in the rest of the application outside the class module can declare and manipulate object variables based on the class modules you provide within the application.

The following subsections concentrate on the basics of creating and manipulating a class in a standard EXE. Later sections extend the use of classes to the world of COM components.

NOTE

Class Builder Utility Not Used in This Chapter Most of the techniques in this chapter assume that you are manually manipulating the class module without benefit of the Class Builder utility.

Add-in modules are beyond the scope of the VB certification exam.

Starting a Class Module in a Standard EXE Project

You can add a Class module to your project by choosing the Project, Add Class Module VB menu option. From the resulting dialog box (see Figure 12.1), you can choose Class Module, VB Class Builder, Complex Data Consumer, Data Source, or OLE DB Provider.

If you choose the Class Module icon, you will find yourself in the new class module's code window.

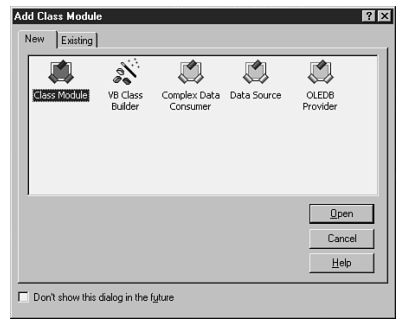


FIGURE 12.1
Dialog box for adding a new class module to your project.

The Class Module `Name` Property

Your class module's `Name` property is important because it is the name that controller code will use when it instantiates a copy of your class as an object. You can change the class name from the Project Explorer by following these steps:

WARNING

Changing the Name of a Class That Is Already in Use Be careful when you change the name of a class that is already in use, because controller code depends on the name of the class to instantiate objects.

- In Project Explorer, right-click the name of the class module to bring up the pop-up menu for this class module (see Figure 12.2).
- Choose the Properties option from the pop-up menu. The only property listed is the `Name` property, and you can change it here (see Figure 12.3).

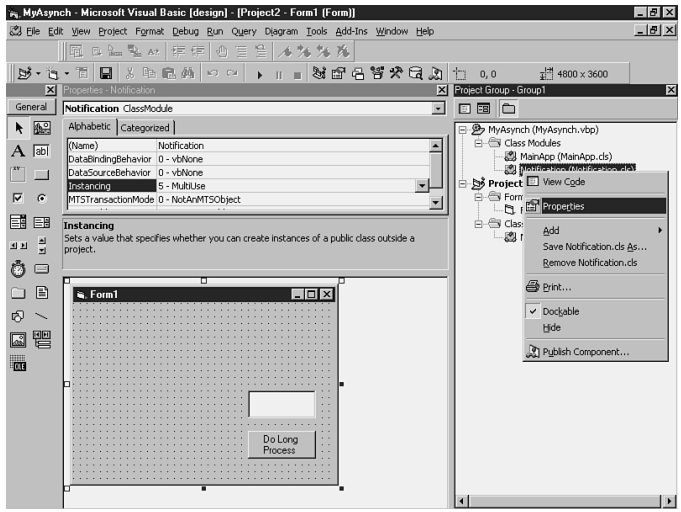


FIGURE 12.2▶
The pop-up menu of the class module

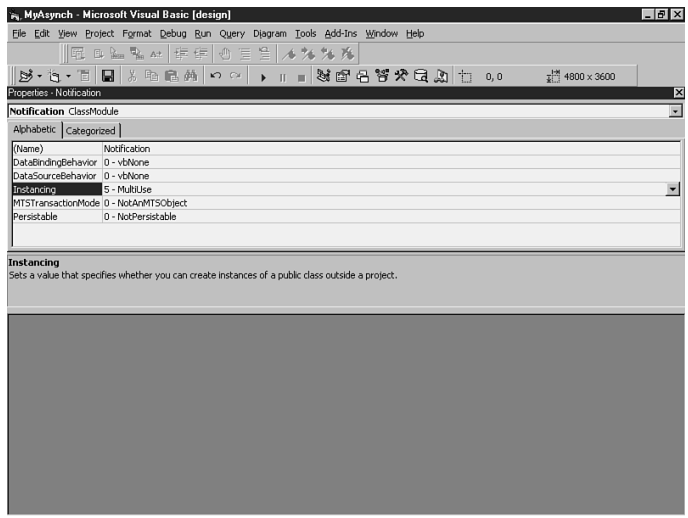


FIGURE 12.3▶
The properties dialog box of a class module in a standard EXE project.

Implementing Custom Methods in Class Modules

When you create methods in a class, you can call them in the rest of your code just as you would call methods for any standard VB class.

To provide methods for your class, just create `Public` procedures in the class module (both `Sub` and `Function` procedures will work). Code from other parts of the application or in ActiveX controllers can then call these procedures directly as methods of the object `Class`.

The example in Listing 12.1 defines a `Public Sub` procedure `SearchPath` in the class module `FileFind`. Code residing elsewhere in the application can call the `SearchPath` method of the object `Class`. The calling code passes to the method the name of the directory to search and a file specification to search for.

LISTING 12.1

IMPLEMENTING A METHOD WITH A `Public` PROCEDURE

```
[In the Class module]
Public Sub SearchPath(strDirToSearch As String, _
        strFileSpec As String)

    'Add a backslash to the directory
    'name if one's not there already
    If Right(strDirToSearch, 1) <> "\" Then
        strDirToSearch = strDirToSearch & "\"
    End If

    'Get the name of the first file in the
    'directory fitting the specified pattern
    Dim strName As String
    strName = Dir$(strDirToSearch & strFileSpec)

    'Loop through all file names in the directory
    'fitting the specified pattern
    Do Until strName = ""
        'Add this file to our list
        frmFiles.lstFiles.AddItem strName
        'get the next file name in the list
        strName = Dir$
    Loop
End Sub
```

continues

LISTING 12.1 *continued*

IMPLEMENTING A METHOD WITH A `Public` PROCEDURE

```
[In the General Declarations of a Form]
Private FF As FileFind

[In the calling routine]
Private Sub cmdFindFiles_Click()
    lstFiles.Clear
    Set FF = New FileFind
    FF.SearchPath _
      dirFind.Path, _
      txtFileSpec.Text
End Sub
```

Implementing Custom Properties in Class Modules

You can create custom properties as part of the functionality that your `Class` object provides to calling code. You can implement these custom properties either as Public variables of the Class or by using special `Property Let/Set` and `Property Get` procedures.

Implementing Properties as `Public` Variables

A `Public` variable in a class module automatically implements a property of the class. You can declare `Public` variables in a class module in the same way you declare `Public` variables for other modules such as forms and standard (BAS) modules.

To add a property called "Name" to a class, you just write the following lines in the General Declarations section of a class module:

```
Public Name as String
```

This is, of course, the same way that forms use `Public` variables. Therefore, when calling code accesses your class by instantiating an object, the caller can manipulate the properties of the object it has declared with the standard `object.property` syntax, as in the following line:

```
objMine.Name = "Geryon"
```

and later

```
MsgBox objMine.Name
```

The basic advantage of implementing an object property with a `Public` variable is that it is very easy to do. The downside is that

- You can't create a read-only property, because `Public` variables are always writable.
- You have no way of triggering an automatic action when the property is manipulated (such as hiding an object when a `Visible` property is set to `False`).
- Different simultaneous instances of a class share the same `Public` variables, and thus cause all sorts of possible confusion in your calling code.

To get around these disadvantages, you must give up the idea of using `Public` variables to implement class properties and instead use `Property Let/Set` and `Property Get` procedures as described in the following sections.

Implementing Properties with Property Procedures

With some additional work, you can overcome the disadvantages of properties implemented as `Public` variables.

You can use the special `Property` procedures instead. To implement the `Name` property of a class with `Get` and `Let` procedures, you put the code shown in Listing 12.2 in your class module rather than a `Public` variable declaration.

LISTING 12.2

IMPLEMENTING A CLASS PROPERTY WITH `Property Get` AND `Let` PROCEDURES

```
[General Declarations section of Class]
    Private gstrName As String

[General Code section of Class]
    Public Property Let Name(strVal As String)
        gstrName = strVal
    End Property
    Public Property Get Name() As String
        Name = gstrName
    End Property
```

Using a `Private` Variable of the Class to Store a Property

Notice in Listing 12.2 that you still declare a variable in the class's General Declarations section as you would when implementing properties with `Public` variables, but this time the variable is a `Private` variable. This means the variable won't be visible outside the class. You use it to store the value of the property you are implementing, but code won't access this variable directly. Instead, `Property Let` and `Get` procedures will be the gateway to the property.

Allowing Writes to a Property with `Property Let`

The `Property Let` procedure is a `Sub` procedure that will run whenever the controlling code attempts to assign a value to the property, as when, say, it runs code such as:

```
objMine.Name = "Ciacco"
```

Notice that the `Property Let` procedure in Listing 12.2 takes a single parameter. The parameter holds the value that the controlling code is trying to assign to the property. If your controlling code had run the single line listed here, the value of the parameter `strVal` in the `Property Let` procedure would be `"Ciacco"`. The `Property Let` in Listing 12.2 then does what `Property Let` procedures almost always do: It stores the incoming parameter value in the `Private` variable (`gstrName` in this case) where the value of the `Name` property is being held.

Allowing Reads of a Property with `Property Get`

`Property Get` is a `Function` procedure that will run whenever controlling code attempts to read the value of the property named by the procedure, as in either of the two lines in Listing 12.3. Notice that the `Property Get` in Listing 12.2 takes no parameters, but has a return type and sets a return value equal to the name of the `Property Get` procedure, just as any normal `Function` procedure does. That is because the `Property Get`'s job is to pass a value back when controlling code requests it. Typically, the `Property Get` will do just as the example in Listing 12.2 does: It will get the value of the property from the value stored in a `Private` variable of the class.

LISTING 12.3

CONTROLLER CODE THAT READS THE VALUE OF AN OBJECT'S PROPERTY

```
    ThisName = objMine.Name
    MsgBox "And then we met " & objMine.Name
```

Implementing a Read-Only Property

If you would like a property to be read-only for controller code, all you have to do is leave out the `Property Let` procedure from your class.

That way, the controller has no way of writing the property. You can still store the value of the property in a `Private` variable of the class module. Other parts of your server class can manipulate the property by changing the `Private` variable, but the controller application can't change it directly.

Listing 12.4 implements an Integer-type property to track the number of times the `Name` property changes. The tracking property is called `Accesses`, and is stored in a class `Private` variable called `giAccesses`. The `Accesses` property would be updated indirectly through `giAccesses` every time the `Name` property gets changed by a controller (the `Property Let` Name procedure increments `giAccesses`). Because there is no written `Property Let Accesses` procedure, however, there is no way for controlling code to change the `Accesses` property directly.

WARNING

Don't Use `Public` Variables to Store Intermediate Property Values
Don't use class `Public` variables to store `Property Let/Get`'s intermediate values, because then controller code can read and write them directly, which will confuse the issue of how to access the property's value. Always use class `Private` variables to store the values of properties that you implement with `Property` procedures.

LISTING 12.4

IMPLEMENTING A READ-ONLY CLASS PROPERTY BY OMITTING ITS PROPERTY LET PROCEDURE

```
[General Declarations of Class]
    Private giAccesses As Integer
    Private gstrName As String

[Code section of Class]
    Property Get Accesses() As Integer
        Accesses = giAccesses
    End Property
```

continues

LISTING 12.4 *continued*

IMPLEMENTING A READ-ONLY CLASS PROPERTY BY OMITTING ITS PROPERTY LET PROCEDURE

```
Public Property Let Name(strVal As String)
    gstrName = strVal
    giAccesses = giAccesses + 1
End Property

Public Property Get Name() As String
    Name = gstrName
End Property
```

Implementing Custom Events in Class Modules

You can define and fire your own custom-defined events in a class module. The act of firing an event from within the class module's code is known as *raising* the event. When an object instantiated from your class raises an event, the controller code can handle that event in an event procedure.

Declaring the Event

You must first define the event by declaring it in the General Declarations section of the class. The syntax for an event declaration is

```
Public Event EventName (argument list)
```

where *argument list* is a list of arguments declared by name and type similar to the parameter list you would declare in a regular procedure. You might declare an event named `FileFound`, for example, with the following line in a class's General Declarations section:

```
Public Event FileFound  _
    (FileName as String, _
     FilePath As String)
```

The controlling code would then be able to write an event procedure that received two String-type parameters. As the following section shows, it is the responsibility of the code in the class that raises the event to determine which values the arguments will hold.

Unless you specify otherwise with the `ByVal` keyword, the arguments are passed by reference. This means that the event procedure in the controller code could change the value of the parameters. Modifiable arguments provide a way for an object and its controller to communicate with each other during processing, because you could specify arguments whose purpose is to carry a message back from the controller. In the following example, a third Boolean parameter is added: `Cancel`. The controller might change this to tell our class object to stop processing:

```
Public Event FileFound _
    (FileName as String, _
     FilePath As String, _
     Cancel as Boolean)
```

Raising the Event and Implementing Callbacks in a Class Object

The only other thing you need to do in the class to implement an event is to fire, or raise, the event in a single line of code, typically in a method of the class, as shown in Listing 12.5 with the `RaiseEvent` keyword.

Although, as just stated, you raise the event with a single line of code, you will probably need to write several more lines to fully implement the firing of the event, especially if the event provides arguments to the controller: Before firing the event, you will want to prepare the value of the arguments to be passed to the controller, and after the event fires, you will want to check the arguments' values to see whether the controller has changed any of them.

Notice that the code checks the value of the `Cancel` argument in Listing 12.5 after the event fires to see whether the controller wants this code in our class to continue processing. Because controllers can modify an event's arguments, this effectively implements callback functionality between controller and object.

LISTING 12.5

Raising a Custom Event from the Code in Your Class

```
'. . .code to assign values of strName and strDir
'initialize value for
'Cancel argument
blnCancel = False
```

continues

LISTING 12.5 *continued*

RAISING A CUSTOM EVENT FROM THE CODE IN YOUR CLASS

```
'fire the event
RaiseEvent FileFound(strName, strDir, blnCancel)

'check to see whether controller
'changed the Cancel argument
If blnCancel Then Exit Sub
```

Built-In Events of Class Modules

Every Class module comes with two predefined events: `Initialize` and `Terminate`.

You can find the event procedure stubs for these two events in the class module's code window by clicking the Objects drop-down list (left-hand list box at the top of the code window) and choosing the Class object.

The Procedures drop-down list (right-hand list box at the top of the code window) then displays the `Initialize` and `Terminate` event procedure stubs, which you may navigate to with the mouse (see Figure 12.4).

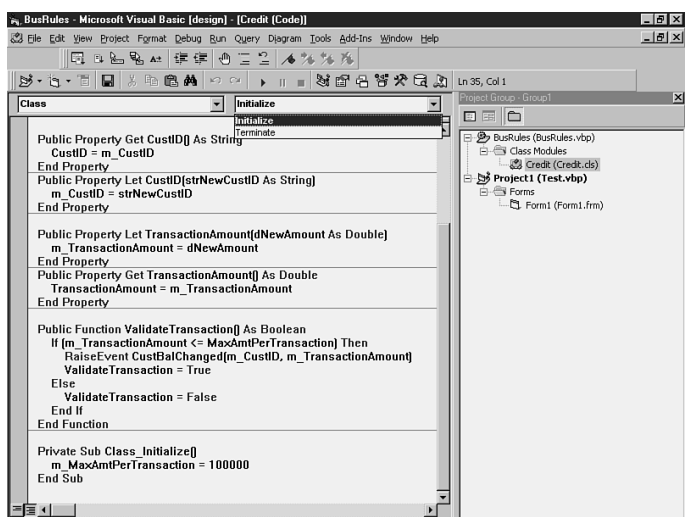


FIGURE 12.4
Navigating to a class module's `Initialize` and `Terminate` event procedures.

The `Initialize` Event

The `Initialize` event happens whenever a routine in another part of the application or in an ActiveX client uses your class module to instantiate a new copy of an object.

You therefore want to put code in the `Initialize` event procedure that is appropriate for—well—initialization.

You might set default initial values of properties from the Windows Registry or Private constants, for instance. If your class accesses data, you might open database files here. The example in Listing 12.6 initializes a classwide Private variable that is used as the storage for a pair of `Property Let/Property Get` procedures.

LISTING 12.6

INITIALIZING A PROPERTY'S VALUE IN THE CLASS MODULE'S INITIALIZE EVENT PROCEDURE

```
Private Sub Class_Initialize()
    iNumberOfAccesses = 0
End Sub
```

You should remember that the `Initialize` event takes place before your object is completely instantiated. Therefore, it is not a good place to do things such as instantiating another object variable of the same class: This would cause an infinite recursive loop, as successive copies of the object would try to instantiate each other.

The `Terminate` Event

The `Terminate` event happens whenever a routine in another part of the application or in an ActiveX client destroys the object instance of your class module. An object gets destroyed in one of two ways:

- The variable that was used to hold your class module's object goes out of scope in the calling code.
- The calling code explicitly destroys the object with the syntax *`Set Object = Nothing`*.

You might use the `Terminate` event procedure to close data files or write information to the Windows Registry or INI files, or just to give a farewell message to your user.

NOTE

Define Properties and Methods as `Public` Explicitly defining properties and methods as `Public` when creating them will avoid confusion later because you or another programmer may assume that the scoping is not `Public`. Also, there is the slight (actually, very unlikely) but possible chance that Microsoft will change the default at some later date. This could lead to large sections of your code breaking. Avoid these possibilities and explicitly define properties by using the appropriate scope when you create them.

Using `Public`, `Private`, and `Friend`

If a property or method is added to a class module, it becomes part of the class. This property or method is usable by other modules in the same project as the class module. It is also available to other projects, as a property or method of objects derived from that class. This is the default setting for the scope of a property or method. There are three other ways of defining the scope of a property or method:

- `Public`
- `Private`
- `Friend`

Using the `Public` Keyword

If the `Public` keyword is used when defining a property or method, the property behaves as though the `Public` keyword is not used. This is because the default scope of a property or method defined in a class module is `Public`.

`Public` properties and methods are available to other modules in the same project as the class module. They are also available to other projects as properties and methods of an object derived from that class module. Most properties and methods will be defined as `Public`.

Consider the following, for example:

```
Public Property Get HireDate() as Date
Public Sub Save()
```

Using the `Private` Keyword

Using the `Private` keyword when defining a property or method creates a property or method private to that class module. Only code in that class module can access the property or method.

`Private` methods are typically support routines used by the properties of a class. For example, you may need some helper routines that perform date calculations.

Consider the following, for example:

```
Private Function NextWorkDay(FromDate as Date) As Date
Private Sub PrintForm(FormName As String)
```

Using the `Friend` Keyword

Using the `Friend` keyword when defining a property or method creates a property or method private to the project containing the class module. Code in the same class module as the procedure can access the property or method. Code in other modules in the same project can also access the property or method. This is useful when creating helper classes that should not be used by other programs.

`Friend` properties and methods are often used when a number of classes assist the main class that other programs will use. A class used to write another object to a file, for example, might have two methods: `Load` and `Save`. These methods, declared as `Friend`, would allow the calling object to access them. They would not be available to other projects.

Consider the following, for example:

```
Friend Sub Load(FileName As String)
Friend Function Save(FileName As String) As Boolean
```

By explicitly setting the scope for each of the properties and methods in a class, you can better control how other modules and programs access them. Most properties and methods will be `Public`; however, having access to the `Private` and `Friend` keywords allows for greater control of access.

NOTE

Don't Use `As Object` to Declare `Friend` Objects When declaring an object that has properties or methods declared as `Friend`, you must declare the object explicitly. If the object is declared `As Object`, the `Friend` properties will not be available for use.

Storing Multiple Instances of an Object in a Collection

A controller may need to create an indefinite number of copies of a single object class. The `FileFind` method might instantiate a special `File` object for every file it finds, for instance. You don't know, however, exactly how many `Files` might get created. You have already seen a concept in VB that enables you to implement something like this: the *collection*.

To implement your own custom collection in a project, you need to add two classes:

- A *dependent class*, which implements a single object of the collection
- A *dependent collection class*, which implements the collection of dependent `Class` objects

These classes are in addition to at least one other class that probably already exists in your project: the *parent class*, which is considered to "own" the dependent collection class.

By convention, a to dependent class Name is a singular noun and a dependent collection class Name is the plural form of the same noun. You might name a dependent collection class `Files` and a dependent class `File`, for example.

Setting Up the Dependent Class

To start a dependent class in your server project, just insert a class module into the project and make sure that the new module is named appropriately. As mentioned earlier, the name of the dependent class should be a singular noun.

You can then give the dependent class any features you need individual elements of the collection to have, such as custom properties, methods, or events.

Setting Up the Dependent Collection Class

Once again, insert a class module into the project and make sure that it is named appropriately: Typically, the name will be the plural form of the dependent class name.

Now, in the General Declarations section of the newly inserted module, declare a `Private Collection` variable, for example:

```
Private colFiles As New Collection
```

The `Collection` type has special methods and properties discussed in the next section that enable you to implement a collection of objects.

You declare the `Collection` in the General Declarations section so that the collection gets initialized as soon as an object is instantiated from this class. You declare it as `Private` so that controller code can't directly manipulate the `Collection` object. Instead, the controller code will have to go through wrapper methods (`Public` procedures) that you create in this class.

Implementing Built-In Collection Features in the Dependent Collection Class

`Collection` objects have three built-in methods and one built-in property:

- The `Count` property, which returns the number of items in the collection
- The `Add` method, which adds a new item to the collection and returns a pointer to the new item
- The `Remove` method, which removes an item, given its index or unique key
- The `Item` method, which returns a pointer to an existing item, given its index or unique key

As mentioned in the preceding section, you don't call these features directly from outside the dependent collection class. Instead, you need to write `Public` wrapper functions that, of course, become methods of the dependent collection class. These methods then mediate between controller calls to the collection and the collection itself.

The following sections list the wrapper methods you must write for each of these features.

The Collection's `Count` Property and `Class Wrapper` Method

You need to write a `Count` method in the dependent collection class to wrap the collection's `Count` property. This method just calls the `Private Collection` object's `Count` property and returns its value, as in Listing 12.7.

LISTING 12.7

A Wrapper Method for the Collection's Count Property

```
Public Function Count() As Long
    Count = colFiles.Count
End Function
```

The Collection's `Add` Method and `Class Wrapper` Method

You need to write an `Add` method in the dependent collection class to wrap the collection's `Add` method. You will implement this as a `Function` returning an object variable pointing to the newly added item, as in Listing 12.8. Notice that you pass the wrapper `Add` method any properties that you want to assign to the new object you will create. You initialize a new object variable whose type is an object of the dependent class. You then assign the properties that were passed into the wrapper and call the built-in `Add` method of the `Collection` object to add the newly created object to the `Collection` and give it a key value. Finally, you set the return value of your wrapper method to point to the newly added item.

LISTING 12.8

A `Class` ADD METHOD TO WRAP THE COLLECTION'S ADD METHOD

```
Public Function Add (ByVal Name As String, _
                     ByVal Path As String) _
      As File

    'Initialize new object variable
    Dim filNew As New File

    'assign parameters to its
    'properties
    FilNew.Name = Name
    FilNew.Path = Path

    'Add it to the collection
    colFiles.Add filNew, Name

    'Let the return value of this
    'function point to it
    Set Add = filNew
End Function
```

The Collection's `Delete` Method and `Class Wrapper` Method

You need to write a `Remove` method in the dependent collection class to wrap the collection's `Remove` method. You will implement this as a `Sub` procedure that takes an argument indicating the index in the

collection of the object to be deleted, as in Listing 12.9. Notice that the parameter is declared as a `Variant` rather than as an `Integer` or a `Long` as you might expect for an `Index` value. The reason for this is that controlling code can use either a numeric index or a String-type key value to look up an item in the collection.

LISTING 12.9

A Wrapper Function for the Collection's Remove Method

```
Public Sub Remove (ByVal Index As Variant)
    colFiles.Remove Index
End Sub
```

The Collection's `Item` Method and `Class Wrapper` Method

The `Item` method you will write wraps around the collection's built-in `Item` method, as you can see in Listing 12.10. The method will return a pointer to a particular item in the collection. To do so, it must take an `Index` parameter, which can be either an Integer array index of a unique key of type String. As with the `Remove` method, this parameter must be declared as a `Variant` to accommodate both these possibilities.

LISTING 12.10

An Item Wrapper Method for the Collection's Built-In Item Method

```
Public Function Item _
      (ByVal Index As Variant) _
      As Variant
    Set Item = colFiles.Item(Index)
End Function
```

Initializing the Collection in the Parent Class

The `Parent` class shows us just the tip of the collection's iceberg. It doesn't have to do much to implement this custom object class collection, but it does need to refer to the collection, because it provides the gateway to the collection by being the only externally creatable class.

You need to put a statement in the `Parent` class's General Declarations that initializes a pointer to the dependent collection class. If the dependent collection class were named `Files`, for example, you would put a line in the `Parent` class's General Declarations that would read as follows:

```
Public Files As New Files
```

Declaring and Using a Class Module Object in Your Application

The previous sections of this chapter have discussed how you can implement a custom object's properties, methods, events, and collections in a class module.

Now it is time to see how to use a custom object in controlling code. Such code might reside in another module in the same application or in a separate ActiveX client.

Declaring and Instantiating a Class Object

You have two basic options for actually instantiating a Class object: Declare it like any other variable and assign its contents later, or instantiate the object when you declare it.

Declare Now, Instantiate Later

You can declare the custom object somewhere in your code just as you would declare a VB variable of a standard variable type. Use the class name as the variable, as in the following example:

```
Private FF As FileFind
```

Later in your code, you need to use the `Set` and `New` keywords to instantiate the object:

```
Set FF = New FileFind
```

Instantiate When You Declare

You can declare a `Class` object and instantiate it with a single statement by using the `New` keyword in the declaration statement:

```
Private FF As New FileFind
```

Declaring `WithEvents`

If you plan to program the Class object's event procedures, you must use the `WithEvents` keyword in the declaration, as in the following example:

```
Private WithEvents FF As FileFind
```

You can't use the `New` keyword in a `WithEvents` declaration, so you always have to instantiate these Class objects later, as described earlier in the section titled, "Instantiate When You Declare".

Manipulating Class Object Properties and Methods

After you have instantiated your `Class` object, manipulating the object's properties and methods becomes quite straightforward: You use exactly the same techniques and syntax as you would use to manipulate VB's standard objects.

You manipulate a `Class` object's properties with the *`Object.Property`* syntax, as in the following example:

```
objMy.Name = "Vergilio"
MsgBox objMy.Path
```

You manipulate a `Class` object's methods with the `Object.Method` syntax, as in the examples in Listing 12.11.

LISTING 12.11

CALLING CUSTOM METHODS IN CONTROLLING CODE

```
'Method without argument or return value
objName.ShowMsg

'Method with argument
objName.KillFile "MY.BAK"

'Method with return value
strDir = objName.FindFile("VB.EXE")
```

Handling a Class Event

You handle a Class object's event just as you would handle events for a VB control: by writing code in the predefined stub of its event procedure. To make the object's event procedure available in a form or another class module, you must declare the object variable with the special `WithEvents` keyword in the General Declarations section, as in the following example:

```
Private WithEvents FF As FileFind
```

where `FF` will be the name of the object variable, and `FileFind` is the name of its class.

After you insert this declaration in your code, you will be able to see the object variable in the Objects list box at the upper-left corner of the code window of the file (Form or Class) where you put the declaration. When you choose the object from this list, you will then be able to see the object's events in the Events list box at the upper-right corner of the code window. If you use the Events list box to navigate to the event you are interested in, you will find your cursor blinking inside the predefined event procedure stub for that event, just as you would for any control you had placed on a VB form (see Figure 12.5).

At this point, all you need to do is to write code in the event procedure, as in Listing 12.12. If the event procedure furnishes parameters, you can use them and, if appropriate, change them. The event procedure in Listing 12.12 takes three parameters. The first two are informational, and the third (`Cancel`) could be changed to `True` to tell the object to stop the process that is generating these events (in this case, a file search process).

LISTING 12.12

EVENT-HANDLING CODE FOR A CUSTOM OBJECT NAMED FF

```
Private Sub FF_FileFound _
  (strFileName As String, _
   strFilePath As String, _
   Cancel As Boolean)
    iFilesFound = iFilesFound + 1
   If iFilesFound [me] 20 Then
    'tell object to stop processing if
    'we found the maximum number of files
        Cancel = True
    End If
End Sub
```

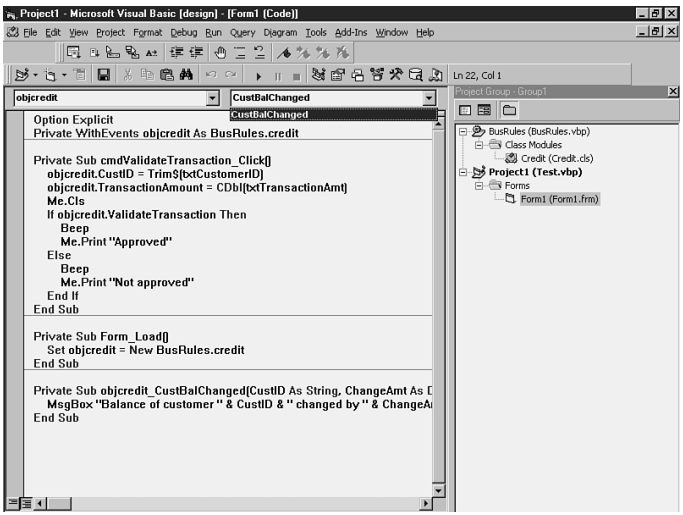


FIGURE 12.5
Navigating to the event procedure for a custom object's event.

WARNING

Can't Use `As New` in `WithEvents` Declaration When you declare an object using the `WithEvents` keyword, you can't instantiate it at the same time with the `New` keyword. You must instantiate the object later in your code.

MANAGING THREADS IN A COM COMPONENT

- Choose the appropriate threading model for a COM component.

VB6 gives programmers some capabilities for managing *threads* in COM components. A thread, simply defined, is a single path of execution in a computer.

In Win16 (Windows for Workgroups and Windows 3.*x*), every application had one and only one separate thread.

In Win32 (Windows 95, Windows NT, and beyond), a single application may use more than one thread. This might be advantageous if the application instantiates multiple objects at the same time. Each object could have its own thread, or the application might have a fixed number of threads to use and could manage its objects across these threads.

All VB6 applications use a type of threading known as *apartment-model threading*. When threads run under the apartment model, the object in each thread is unaware of objects in other threads and has its own separate copy of global data.

WARNING

Can't Use `WithEvents` in a Standard Module You can'tuse the `WithEvents` keyword in a standard module.

The options for threading are different for COM components, depending on the type of component that you are developing. The following sections discuss these options.

Managing Threads in ActiveX Controls and In-Process Components

If your application is an in-process component (ActiveX DLL) or an ActiveX control, the client application will control the creation and management of threads. You may still, however, specify whether your component will allow a single thread or multiple apartment threads.

You can specify whether your component supports multiple apartment threading by choosing either Apartment Threaded or Single Threaded from the drop-down box in the Threading Model section under the General tab of the Project, Properties dialog box, as shown in Figure 12.6.

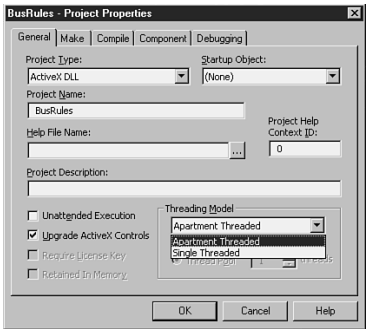


FIGURE 12.6
Specifying the threading model for an in-process component (ActiveX DLL) or an ActiveX control.

Note that the default setting for in-process and control components is Apartment Threaded.

Finally, you can check the Unattended Execution box on the General tab of the Project, Properties dialog box (refer again to Figure 12.6) to ensure that your in-process component is *thread-safe*.

Managing Threading in Out-of-Process Components

If your application is an out-of-process component (ActiveX EXE), you have three choices for its threading model (refer to Figure 12.6):

- **A Single thread**. When a client makes a request to the component before previous requests have finished processing, the additional request is *serialized*. A serialized request is placed in a queue behind other requests on the same thread.

  To make your out-of-process component single-threaded, select the Thread Pool option on the General tab of the Project, Properties dialog box and make sure that the number of threads is one.

NOTE

Single-Threaded ActiveX Controls Cause Problems in Multithreaded Clients If a client application is multithreaded, a single-threaded ActiveX control can cause problems for the multithreaded client. In fact, if a multithreaded client application is being written in VB, VB will not even allow programmers to use a single-threaded ActiveX control in the project.

Typically, developers choose this option when compiling projects that were created in earlier versions of VB.

- **Round-robin thread assignment to a fixed number of threads**. There is an upper limit on the number of threads that the component will support. As additional requests pile up beyond the maximum number of threads, the system moves to the next thread (or cycles back to the first thread if it was already on the least thread), serializing the new request behind other requests on the next thread.

  To implement round-robin thread assignment, select the Thread Pool option on the General tab of the Project, Properties dialog box, setting the number of threads to a number greater than one. You must also make sure that the project is marked for Unattended Execution.

  Round-robin threading can actually manage system resources and performance more efficiently than can unlimited threading (discussed next), because round-robin threading limits the amount of server overhead that will be dedicated to a COM component's objects.

  The main disadvantage of round-robin threading is that it is *nondeterministic* regarding the exact allocation of objects to threads. In other words, you can neither predict nor discover which objects are assigned to which threads, nor whether threads are *load balanced*. Load balancing is the art of allocating roughly similar amounts of work to available threads. In other words, it is possible with round-robin threading that one thread might have half a dozen objects serialized, and another thread may have only one object waiting in its queue, or in some cases no object at all.

- **Unlimited threads**. A separate new thread will begin for every new object.

  To implement unlimited thread assignment, select the Thread per Object option on the General tab of the Project, Properties dialog box. You must also make sure that the project is marked for Unattended Execution.

  Unlimited threads will, of course, guarantee that every new connection to the out-of-process component gets its very own thread and therefore suffers no blockage.

The performance of the system as a whole might degrade with a lot of unlimited thread connections, however, because the server will have to provide resources for each thread.

The section titled "Implementing Scalability Through Instancing and Threading Models" discusses these threading options in the light of solution scalability.

The Instancing Property of COM Component Classes

▶ Set properties to control the instancing of a class within a COM component.

Class modules in a COM component can be classified according to their relationship to client applications:

- **Service classes**. These classes are for internal use by the server itself and are invisible and unusable to clients.
- **Dependent classes**. These classes are visible to clients, but clients can only instantiate and manipulate them indirectly through the third type of class, externally creatable classes. (For a further discussion of dependent classes, see the subsections under "Storing Multiple Instances of an Object in a Collection" in this chapter).
- **Externally creatable classes**. These classes are visible to clients, and clients can instantiate and manipulate their objects directly.

When you create a class module in a COM component, you should set its `Instancing` property to reflect how the class module will behave with respect to clients.

To set the class's `Instancing` property, follow these steps:

- In Project Explorer, right-click the name of the class module to bring up the pop-up menu for this class module.
- Choose the Properties option from the pop-up menu. On the Properties window, you can choose the drop-down list for the `Instancing` property's settings (see Figure 12.7).

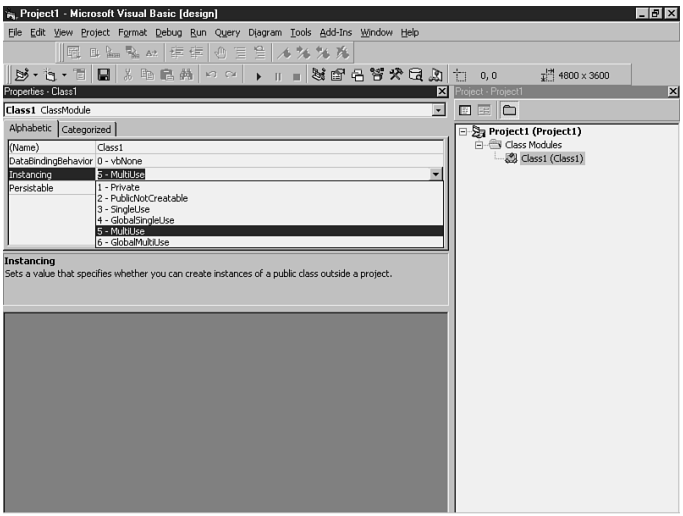


FIGURE 12.7
Choices for the `Instancing` property in the Properties dialog box of a class module in an ActiveX EXE project.

Note that the `Instancing` property is unavailable for the class modules of nonserver projects. In such projects, the `Instancing` property is always implicitly `Private`.

Using `Private` Instancing for Service Classes

A *service class* is a class that is only used internally by the server. In other words, you don't want to make this class available in any way to your server's clients, either as an externally creatable class or as a dependent class.

If you set the class's `Instancing` property to `Private`, no applications outside your server can see this class or instantiate objects from it.

Using `PublicNotCreatable` Instancing for Dependent Classes

If you want to set up a dependent class, your class must be visible to clients. A dependent class object is, by definition, not directly managed by clients, however. Instead, clients can only instantiate and manipulate dependent classes through externally created classes (see the following sections).

NOTE

`SingleUse` Instancing Not Available in DLLs The `SingleUse` `Instancing` setting is not available in ActiveX DLL projects.

For dependent classes, you will set the `Class Instancing` property to `PublicNotCreatable`. This signifies that an object of this class is visible to a client, but that the client can't create such an object directly.

`Instancing` Property Settings for Externally Creatable Classes

As you will recall from the section titled "Overview of COM Component Programming," an externally creatable class provides objects that a client application can instantiate directly.

You might guess that, because there is a single `Instancing` property setting for service classes and one for dependent classes, there would be a single setting as well for externally creatable classes.

The situation is a bit more complex than one `Instancing` setting can handle, however, because external objects have several additional features:

- External objects can be single use or multiuse. A single-use object can't be shared by more than one client, whereas a multi-use object can (see the following sections for further details).
- External objects can be global or they can require explicit instantiation (see the following sections for further details).

The combination of these two features yields four possible `Instancing` properties for externally creatable objects, as discussed in the following four sections.

Using `SingleUse` Instancing for Separate Instances of Every Object

A `SingleUse` class can only supply one instance of itself per copy of its component.

Assume that your COM component has two `SingleUse` object classes, `ClassA` and `ClassB`.

If two clients want to instantiate copies of `ClassA` from your server at the same time, ActiveX runs a second instance of the server each time an instance of `ClassA` is created.

If one client wants to instantiate a copy of `ClassA` and another client wants to instantiate a copy of `ClassB`, however, ActiveX will allow both of them to use the same copy of your server, as long as each object instance is the first requested instance for each class.

NOTE

`GlobalSingleUse` Instancing Not Available for DLLs The `GlobalSingleUse` Instancing setting is not available in ActiveX DLL projects.

Using `GlobalSingleUse` to Avoid Explicit Object Creation

A `Global` class does not require a client to explicitly instantiate it before the client tries to use it.

In other words, if a `Global` class exists in a server application that a client is referencing, the client can manipulate the `Global` class methods and properties as if they were just `Public` variables and procedures of the client.

Suppose, for example, that your server, named `MyServices`, has a `SingleUse` server class named `FileSvc`, with a property named `fsSize` and methods named `fsFind` and `fsDelete`. If client programmers wanted to use the `FileSvc` class to manipulate a file, they might write code that looks like Listing 12.13. This code declares and instantiates an object from the server/class combination and then calls the object's methods and properties, using the *`object.member`* syntax.

NOTE

Note: Internal Code Must Instantiate `GlobalSingleUse` Classes Explicitly Although *client* projects can access members of `GlobalSingleUse` classes as if they were `Public` variables, code *within the same server project* as the `GlobalSingleUse` class cannot do so. In other words, if you want to access the members of a `GlobalSingleUse` class from within the same server project, you must still instantiate an object from the `GlobalSingleUse` class to do so.

LISTING 12.13

MANIPULATING AN OBJECT FROM A CLASS THAT'S NOT GLOBAL

```
Dim filCurr As New MyServices.FileSvc
filCurr.fsFind "VB6.dep"
If filCurr.fsSize = 0 Then
    filCurr.fsDelete
End If
```

You could save client programmers some work, however, by setting the `Instancing` property of the `FileSvc` class to `GlobalSingleUse`. If you did so, client programmers could accomplish the same thing by writing the code in Listing 12.14. Notice that it is not necessary to declare or instantiate an object variable, nor is it necessary to use any sort of object reference at all when you want to manipulate the class's properties and methods.

As mentioned in the note accompanying this section, however, you *must* fully declare the object variable when programming with a `GlobalSingleUse` class *within* the component itself.

LISTING 12.14

MANIPULATING A GLOBAL CLASS OBJECT

```
fsFind "VB6.dep"
If fsSize = 0 Then
    fsDelete
End If
```

Before you decide to make all your externally creatable classes `GlobalSingleUse` or `GlobalMultiUse`, however, consider the drawbacks:

- Client-side code is now more ambiguous and therefore less maintainable.
- It's easier to confuse multiple instances of global information.

Typically, you will only use `GlobalSingleUse` when the function a class provides is very, very generic.

Using `MultiUse` Instancing for Object Sharing

A `MultiUse` class can supply more than one instance of itself per copy of its component.

A single instance of a COM component can create multiple instances of a `MultiUse` class. The first request for a class instance starts the COM component. All subsequent requests are serviced by the already running server. If a client application requests several instances of a `ClassA`, for example, one server creates all instances of `ClassA`.

Using `GlobalMultiUse` Instancing to Avoid Explicit Object Creation

All that has been said about `GlobalSingleUse Instancing` applies to `GlobalMultiUse` as well. The difference, of course, is that the latter provides a `MultiUse` object rather than a `SingleUse` object.

Deciding Between `SingleUse` and `MultiUse` Server Classes

If your server is out-of-process (an ActiveX EXE), you have to make the choice between implementing each of its externally creatable classes as a `SingleUse` or `MultiUse` object.

The main advantage of a `MultiUse` server is that it uses less system resources because the ActiveX Manager needs to run fewer copies of it.

The main reason to use a `SingleUse` server is to reduce "blocking" of one client's use of a method by another client.

Blocking can happen when a server is `MultiUse` and more than one client is trying to call the same code at the same time. The single copy of the server will attempt to satisfy requests for the same service from more than one client at a time. Because all the instances of a `MultiUse` server run in a single thread, one client's requests for a service from the server will effectively block all other requests for that service until the request is completed.

If you have time-intensivecode in your server and expect several instances of the server to be active at the same time, you can avoid end-user frustration by making the server `SingleUse` so that clients don't experience delays while they queue up for their turn to use a `MultiUse` server.

NOTE

DLLs Are Only `MultiUse` In-process (ActiveX DLL) servers can only be `MultiUse`.

Handling Errors in the Server and the Client

- Determine how to send error information from a COM component to a client computer

Two applications will be running when the system invokes a COM component: the client application and the server. When an error happens in your server, you want to make sure the error is handled adequately.

Because your server should run as transparently as possible to the client, your first line of defense for error handling in the server should be full-blown error handling within the server itself.

At times, however, it is more appropriate to let the client handle an error that has occurred. When you need to pass an error back to the client, you can use one of two methods for letting your client application know that an error has occurred:

- Let the server's methods return a result code to the client showing success or failure.
- Raise an error code in the server that the system can pass back to the client.

The following sections discuss these two methods.

Passing a Result Code to the Client

This "soft" method of error handling will not generate an error condition in the client. It will let the controlling application decide whether it is worth checking for an error after calling a method of your server. If the controlling application does check your server method's return value, it can determine whether an error occurred and decide what to do next.

To implement this technique, you should write your server methods as functions with an Integer return type. The return type will be, say, zero or some other encouraging-sounding number if no error happens in the function. You need to write and enable an error-trapping routine in the function that will cause the function to return a negative integer or some other equally dire value when an error does occur. The code in Listing 12.15 illustrates both client- and server-side code to implement this solution.

LISTING 12.15

Return an Error Status Code to the Client

[Method in the server class]

```
Public Function MyMethod () as Integer
    'Initialize return value to OK
    MyMethod = 0
    On Error GoTo MyMethod_Error
    .
    . 'do stuff that could possibly cause an error
    .
    Exit Function
```

```
MyMethod_Error:
    'return error code
    MyMethod = Err.Number
    Resume Next
End Function
```

[Code in the client application]

```
Dim iResult as Integer

iResult = MyServer.MyMethod()
'Check method's result code for an error
If iResult <> 0 then
    ' do something to handle the error
End If

'Or - Following line ignores the method's result code
MyServer.MyMethod
```

NOTE

Programmer-Defined Error Codes
With this type of error handler, you might consider implementing your own system of error codes. You must use integer values higher than 512, because lower values can conflict with existing ActiveX error codes.

The client can then check the return value of the server's method to see whether all has gone well. If it gets an error code back from the method, it can then decide what to do.

Raising an Error to Pass Back to the Client

This method has a more "in-your-face" attitude toward the client. It generates a hard error condition in the client and forces the client to do its own error handling to deal with the server's error.

As you can see in Listing 12.16, the method in the server does not have to be a function, because we're not using its return code to signal a success-or-failure status. The method still uses an error handler as it did in the previous technique, but now instead of setting a return value, it uses `Err.Raise` to cause another error to happen in the error handler.

You append the Visual Basic predefined constant `vbObjectError` to the error code you are raising. This tells the system to pass the error through to the client. When the client receives the error, the value of `vbObjectError` will be stripped off the error code it sees, and the client will see only the error code's original value.

Because an error now occurs in the client, the client must have its own error-handling routine in place.

LISTING 12.16

GENERATING AN ERROR IN THE CLIENT

[Method in the server class]

```
Public Sub MyMethod2 ()
    On Error GoTo MyMethod2_Error
    .
    . 'do stuff that could possibly cause an error
    .
Exit Sub
MyMethod2_Error:
    'client will see Err.Number as 1000
    Err.Raise 1000 + vbObjectErrorlient
End Sub
```

[Code in the client application]

```
    On Error GoTo MyCall_Error
    MyServer.MyMethod2
    Exit Sub
MyCall_Error:
    ' do something to handle the error
```

Note that Visual Basic's default error-handling strategy at design time is to always break in the class of the server application. To be able to test code that uses the strategy discussed here, you must set the general option known as Error Trapping to Break on Unhandled Errors. See Exercise 12.4 for more details.

MANAGING COMPONENTS WITH VISUAL COMPONENT MANAGER

▶ Use Visual Component Manager to manage components.

This section and the following subsections discuss how to use Visual Component Manager (VCM). You can use VCM to make it easier to reuse the COM components that you and others create. VCM helps you to perform three basic tasks for component reuse:

- **Publishing components**. When you publish a COM component, you make it available to others.
- **Finding components**. When you are developing a solution, you investigate whether components that fit your needs already exist.
- **Reusing components**. After you have found an existing component, you incorporate it into your project.

Storing VCM Information in Repository Databases

VCM implements the publishing, finding, and reuse of components by maintaining information about them in one or more *repository databases*.

Because VCM enables you to use more than one repository database, you can separate groups of components into different repository databases based on component functionality.

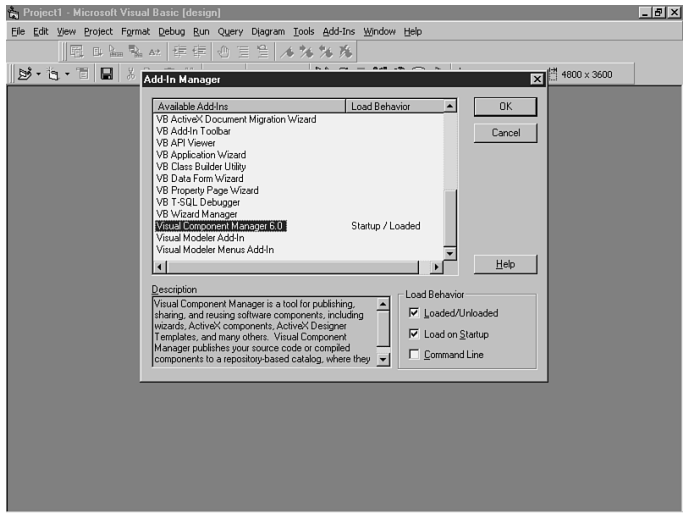
VCM also supports repository databases in MS Access format or in SQL Server format. You could use MS Access for local repository databases and SQL Server for repository databases residing on a server. VCM therefore offers a scalable solution to component management.

Making VCM Available in the VB IDE

If you have installed VB6 or any of Visual Studio's other development tools on your system, you have also automatically installed VCM.

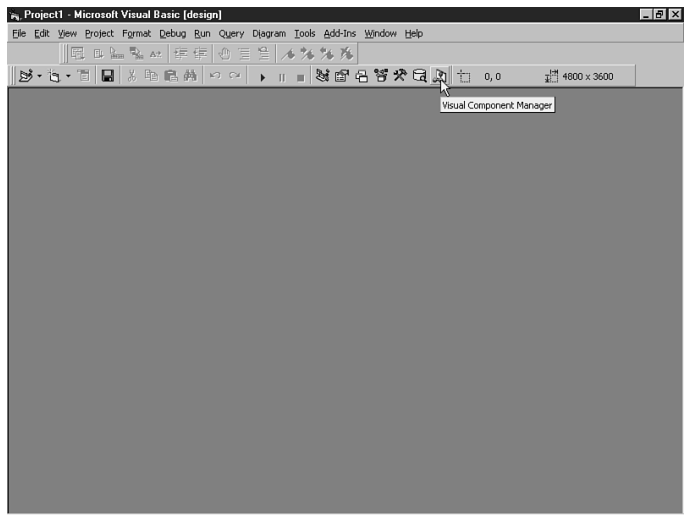
To make VCM available from the VB IDE, you must also add VCM to the VB toolbar. To add VCM to the VB toolbar, open the Add-Ins menu and make sure that the Visual Component Manager 6.0 item has its Loaded/Unloaded and Load on Startup options checked (see Figure 12.8).

FIGURE 12.8▶
Using the Add-Ins dialog box to add Visual Component Manager to the VB IDE.



After VCM has been enabled, you will see the VCM icon on the VB toolbar (see Figure 12.9).

FIGURE 12.9▶
The Visual Component Manager icon on the VB toolbar.



Publishing Components with VCM

To make a component available for reuse through VCM, you must first publish that component in a VCM repository database.

To publish a component through VCM from within the current project, you should take some preliminary steps to prepare the component for publication and to invoke VCM, as discussed in Step by Step 12.1.

STEP BY STEP

12.1 Preparing to Publish a Component with Visual Component Manager

If you intend to publish the component's source code, make sure that you have saved the project before publishing it. Otherwise, VCM will give you the warning `Project must be saved before it can be published` when you attempt to publish the component.

1. Click the VCM icon on the VB toolbar to invoke the VCM window (see Figure 12.10).

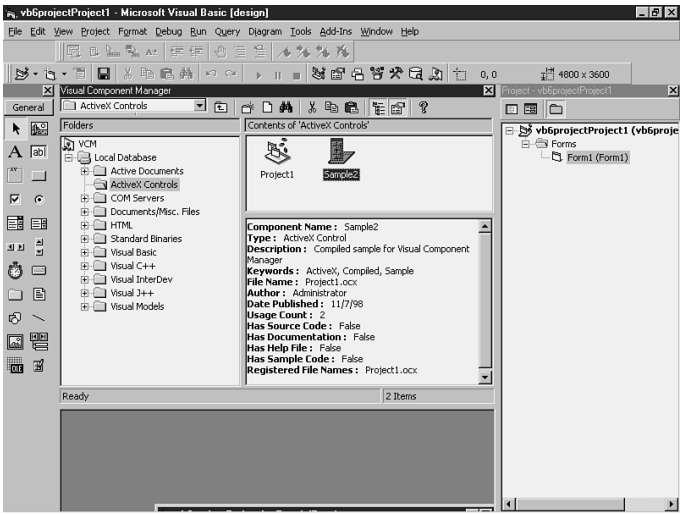


FIGURE 12.10
The main window for VCM.

2. Choose a folder within the VCM repository by double-clicking one on the tree in the left-hand window.

3. Invoke the VCM Publish wizard through one of three methods:

   - Drag a compiled component (*.DLL, *.OCX, or *.EXE) from Windows Explorer to a folder in the Visual Component Manager window.
   - Make sure that you have opened a folder in the repository (double-click it in the VCM window). Then, right-click in the folder's contents (the upper center and right pane of the VCM window). Click New on the resulting Visual Component Manager shortcut menu.
   - Select the component's project window in Project Explorer and right-click the project name to invoke the project's shortcut menu (see Figure 12.11). Choose Publish on the project's shortcut menu. When you click Source Files, you will publish the project's source-code files; when you click Build Outputs, you will compile and publish its outputs. This third technique does not require you to have the VCM window open.

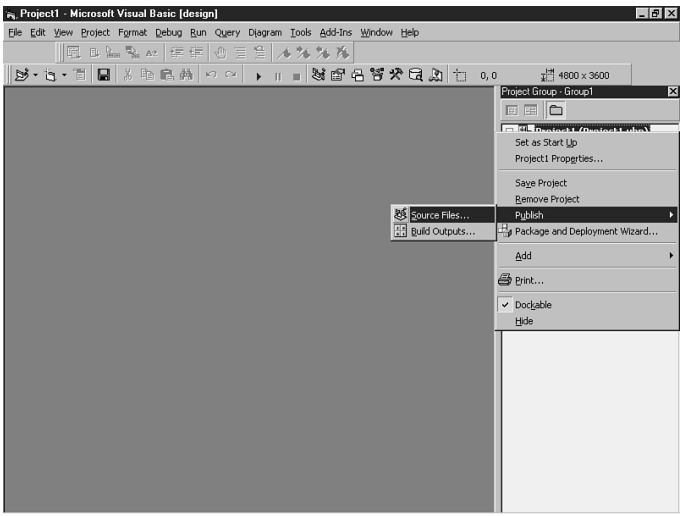
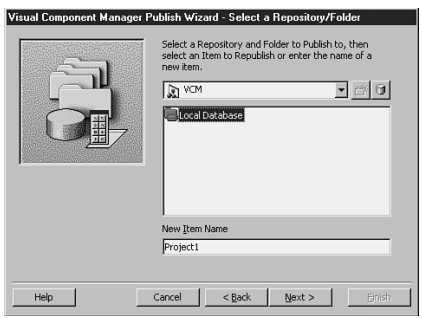


FIGURE 12.11
Visual Component Manager's Publish menu. In this figure, the developer has accessed the menu by right-clicking the Project's name in the Project Explorer.

4. Publish Wizard will display an introductory screen (not shown here). Click the Next button to proceed.

5. On the next screen (Repository/Folder), you can choose a VCM repository database (see Figure 12.12) and then a folder within that database (see Figure 12.13).



◀**FIGURE 12.12**
Initial view of VCM's Repository/Folder dialog box.

6. Complete the Repository/Folder dialog box by editing, if necessary, the name that you want to give to the component. (This name is for informational purposes only and will not affect project or Registry names.) The completed dialog box should look like Figure 12.14. Click the Next button to continue to the Title and Properties dialog box.

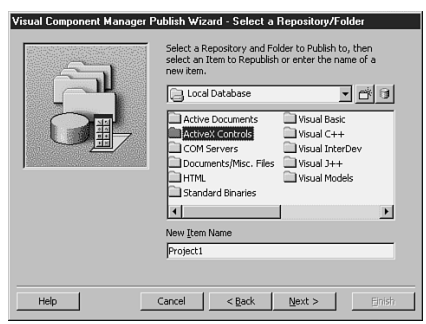
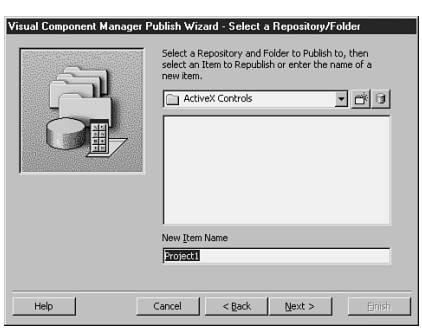


FIGURE 12.13▲
VCM's Repository/Folder dialog box, showing available folders.



◀**FIGURE 12.14**
VCM's Repository/Folder dialog box, showing a selected folder and the name of a component to add to VCM's repository database.

7. You typically don't change anything on the Title and Properties dialog box, although of course you can (see Figure 12.15). Click Next to proceed to the More Properties screen.

FIGURE 12.15▶
VCM's Title and Properties dialog box.

FIGURE 12.16▲
A second Properties screen in VCM enables you to add a component description and keywords.

8. From the More Properties screen (Figure 12.16), you can add a component description and keywords. Other developers can use the description and keywords to find your component later.

9. You can add a new keyword to the component by clicking the large plus button (+) on the More Properties screen and filling in the keyword text on the resulting dialog box (see Figure 12.17).

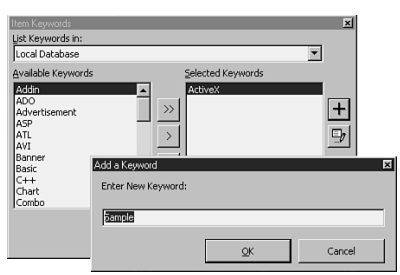
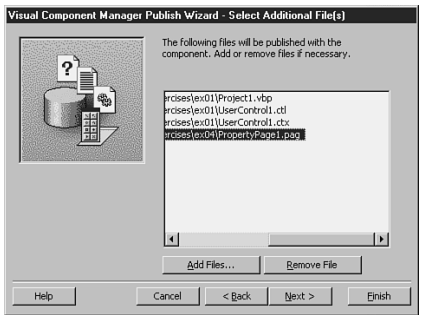


FIGURE 12.17▶
Adding a new keyword to a component in VCM.

10. Click the Next button to proceed to the next Publish Wizard screen. A list of files that will be published as part of the component appears, as shown in Figure 12.18. Depending on whether you chose to publish source code or the compiled component (see step 3), you will either see the source-code files for the project or the compiled file (and any known files that the compile file depends on to run). At this point, you can add or take away files. After you have finished, click the Next button to proceed to the next screen.



◀**FIGURE 12.18**
You can add or delete files from a published component's source code with the VCM Select Additional File(s) dialog box.

11. On the next screen, you can indicate whether any of the files that you are publishing with the COM component require an entry in the Windows Registry (see Figure 12.19). Usually you need to do nothing on this screen, because VCM is pretty good at guessing which files need registration. In fact, if you try to check a file that obviously doesn't need registration (such as a source-code file), VCM will warn you that the file doesn't need registration and will refuse to check the box next to the file's name.

12. After you have made any adjustments to your component's registration information, you can click the Next button. This brings up the Publish wizard's final screen (the Finish screen). Click the Finish button on this screen to finalize your component's entry in the VCM repository database.

FIGURE 12.19▲
You can indicate the files that belong to a published component and that must be registered in the Windows Registry.

Finding and Reusing Components with VCM

When information about components resides in VCM, developers can find the components and reuse them in their projects. Depending on whether you published the compiled component or its source code, developers can get the compiled component or a copy of the source code.

To find and reuse a component with VCM, complete the following steps:

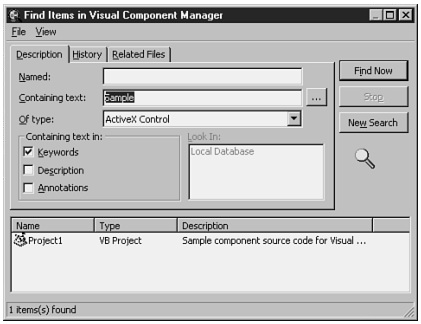


FIGURE 12.20▲
VCM's Find dialog box.

STEP BY STEP

12.2 Finding and Reusing a Component with Visual Component Manager

1. Open VCM (refer back to Figure 12.10) from the VB toolbar by clicking on the VCM icon.
2. Click the Find (binoculars) icon on the VCM toolbar.
3. On the resulting Find dialog box (see Figure 12.20), enter either a component name or text to locate in the component's keywords, description, or annotations. You can also limit your search to just one type of component by making a choice in the Of Type drop-down menu.
4. When the desired component appears in the Find window, double-click its name to select it.
5. The Select Folder dialog box appears. You can choose a folder to place the component in the current project (see Figure 12.21).

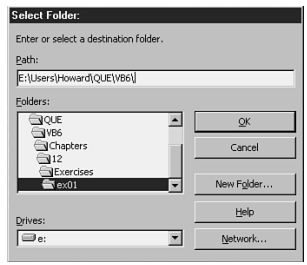


FIGURE 12.21▶
Selecting a project in the VCM Find dialog box.

6. After you have clicked the OK button, the component appears in your project. If the component included source code, the source-code files (typically for a VB project) will appear in the indicated folder. If the component was compiled, an ActiveX control component's icon will now appear in your toolbox. Other COM component types will be added to your project's References list.

f you can see the component in the VCM project window, you can forego steps 2 through 6. All you need do is right-click the component's name in the VCM window to bring up the component's shortcut menu (see Figure 12.22). If the component is an ActiveX control, choose the Add to Toolbox option. If it's another type of component, you can choose the Add to Project option or the Add to Project Group option.

FIGURE 12.22
Using the VCM component shortcut menu to add a component to a VB project.

USING INTERFACES TO IMPLEMENT POLYMORPHISM

Polymorphism, as you may know, is a feature of object-oriented design and programming that enables you to use a single object class in more than one way.

You typically apply this concept to general object types, such as `Person`, `Animal`, `File`, or `Vehicle`. Such general types could represent very disparate objects. A `Vehicle` object, for example, might represent either an automobile, a spaceship, an airplane, or an ox cart, to name just a few possibilities.

Each specific object type then usually has at least two subsets of features: features that apply to all objects instantiated from that class (all `Vehicles`, for instance, have properties such as `Speed` and `MaximumSpeed`, and methods such as `Travel`); and, on the other hand, features that apply only to certain types of objects in that class (such as `NumberOfWings`, `MaximumAltitude`, `MinAnimalsNeededToMove`, and so forth). An additional consideration for the common features would be the fact that the same methods and properties might need to behave differently for different types of objects (the `Travel` method would behave differently for different types of objects).

If you want to provide general object classes along with more specific implementations of those classes in your COM component, you have several choices in VB:

- Add all the properties and methods to a single class. By doing this, any object instantiated from the `Vehicle` class would have a `NumberOfWings` property, even if the particular object in question were a school bus. In addition, a `Startup` method might have to behave very differently, depending on whether the object in question were a spaceship or an ox cart. You would have to put special logic in that method to determine which `Startup` actions to take, based on the type of object.
- Create a completely separate class specific to each type of object. These object classes would share a lot of members in common, but they would be completely separate from each other. You would have a `SchoolBus` class, for example, that had the generic object members such as the `Speed` property and the `Travel` method. Likewise, all your other `Vehicle`-type classes would have these generic object members. The `Schoolbus` class, however, would have a `MaximumChildren` property and a `DisplayStopSign` method. You would write separate routines in each class for all common properties and methods.
- Use the concept of an `Interface` class that will *define* (but will *not implement*) common properties and methods for the more specific classes. Create separate classes that then implement each specific type of object that you will need and that use the `Interface` class to specify common functionality.

The `Interface` class provides an *abstract definition* for the properties and methods of a generic type of object. It does *not*, however, provide that *behavior*. Each specific object class that implements the `Interface` class must actually provide code for each of the methods and properties of the `Interface` class.

The advantage to an `Interface` is that it provides a common specification for all classes that use it, while allowing each class the flexibility of implementing the details of the common specification in its own way, as well as the flexibility of adding other functionality to the base specification. The classes that implement the `Interface` *must* implement all members of the `Interface`: In fact, the VB Compiler will enforce this requirement, giving you error messages if you forget to implement all the members of an `Interface` in an object class that uses the `Interface`.

Continuing with the example of `Vehicles`, imagine an `Interface` class called `IVehicle`. (The *I* at the beginning of the class name is a standard naming convention for `Interfaces`.) `IVehicle` contains properties such as `Speed` and `MaxSpeed` and methods such as `Travel`. You could create other classes such as `Automobile` and `Airplane` that implement `IVehicle`'s members, each in its own way. The `Automobile` and `Airplane` classes might also implement methods and properties of their own apart from the `Interface` (such as `OpenTrunk` or `LowerLandingGear`).

NOTE

`Interface` Classes as Abstract Classes A lot of documentation refers to a VB `Interface` class as an "abstract class." Conceptually, this is true, because you are using an `Interface` as a model for other classes without specifying implementation details. Strictly speaking, however, it should be impossible for an abstract class to be instantiated or hold any code. Because it is possible to write code in a VB `Interface` class (although this isn't often done), and because it is possible to instantiate an object from an `Interface` class in your code, a VB `Interface` class is not technically a true abstract class.

Steps to Implement an `Interface` Class

To use `Interface` classes in your programming, you must take three basic steps:

- Create the `Interface` class and define its members.
- Implement the `Interface` in the appropriate server classes.
- Refer to the `Interface` class in client code when the client needs to use features of server classes that belong to the `Interface` specification.

The following sections discuss these three steps.

NOTE

No Events in `Interface` Classes
VB6 doesn't support events in `Interface` classes, only methods and properties.

Creating the `Interface` Class

An `Interface` class is just an empty class template containing procedures to implement class methods and properties as discussed in the subsections immediately following "Implementing COM Components Through Class Modules" in this chapter.

You don't normally put any code or `Private` variable declarations in an `Interface` class, only the blank procedures.

Therefore, the entire contents of an `IVehicle Interface` class might look like Listing 12.17.

LISTING 12.17

THIS IS ALL THE CODE REQUIRED TO SET UP THE `IVEHICLE INTERFACE` CLASS

```
Option Explicit

Public Property Let Color(newval As Long)
End Property
Public Property Get Color() As Long
End Property

Public Property Get MaxSpeed() As Single
End Property
Public Property Let MaxSpeed(newval As Single)
End Property

Public Function Travel(xStart, yStart, xEnd, yEnd) As Single
End Function
```

Notice that the `IVehicle Interface` provides two properties (`Color` and `MaxSpeed`) and one method (`Travel`).

You specify the type of each property and the number and type of the parameters that the method receives as well as its return value. You specify absolutely no behavior beyond this, however. The way that these members will function depends completely on the programmer who uses this `Interface` in other classes.

Implementing the `Interface` Class in Other Classes

When you want to use the members of an `Interface` in another object class, you need to take two steps:

- Use the `Implements` declaration to refer to the `Interface` in the implementing class's General Declarations section. VB will then automatically create code stubs inside the implementing class for all the methods and property procedures that the `Interface` defines.
- It's a good idea to put code in the implementing class into every method and property procedure provided by VB for the `Interface`. If you don't put at least one line of code in each procedure, there may be confusion about your intentions.

Listing 12.18 shows the contents of a class that implements the `IVehicle Interface` set up in the preceding section.

LISTING 12.18

IMPLEMENTING AN `Interface` CLASS IN ANOTHER CLASS

```
'Class Automobile

Option Explicit

Private m_TireInflation As Integer

Implements IVehicle
    Private m_Color As Long
    Private m_MAXSpeed As Single

Private Property Let IVehicle_Color(RHS As Long)
    m_Color = RHS
End Property

Private Property Get IVehicle_Color() As Long
    IVehicle_Color = m_Color
End Property

Private Property Let IVehicle_MaxSpeed(RHS As Single)
    m_MAXSpeed = RHS
End Property
```

continues

LISTING 12.18 *continued*

IMPLEMENTING AN Interface CLASS IN ANOTHER CLASS

```
Private Property Get IVehicle_MaxSpeed() As Single
    IVehicle_MaxSpeed = m_MAXSpeed
End Property

Private Function IVehicle_Travel(xStart As Variant, yStart As
Variant, xEnd As Variant, yEnd As Variant) As Single
   IVehicle_Travel = Sqr((xEnd - xStart) ^ 2 + (yEnd - yStart)
^ 2)
End Function

Public Property Let TireInflation(NewVal As Single)
    m_TireInflation = NewVal
End Property

Public Property Get TireInflation() As Single
    TireInflation = m_TireInflation
End Property

Public Sub SuddenStop()
    Call ApplyBrakes
    Call Skid
End Function
```

Notice the `Implements` keyword that points to the `IVehicle Interface` at the beginning of the listing. After you have typed this line into your code, VB makes `IVehicle` available as an object in your code window, as Figure 12.23 illustrates. Note in the illustration that all the member procedures (that is, the methods and the property procedures) appear in the Procedures window when you choose the `IVehicle` object from the left-hand drop-down list in the code window. You then place code in each of the procedures (methods and `Property Get/Let/Set` procedures) that the `Interface` provides in your class, as you see in Listing 12.18.

Because this class is the place where you actually specify how the `Interface` elements will behave, you will note the use of `Private` variables to hold intermediate values of properties. Note that the name of the parameter passed to `Property Let` procedures is `RHS`. VB automatically supplies this name, which stands for "right-hand side" (that is, the right-hand side of an assignment statement, such as the *b* in "a = b").

Finally, note that this class, named `"Automobile"`, has some members of its own: the `SuddenStop` method and the `TireInflation` property.

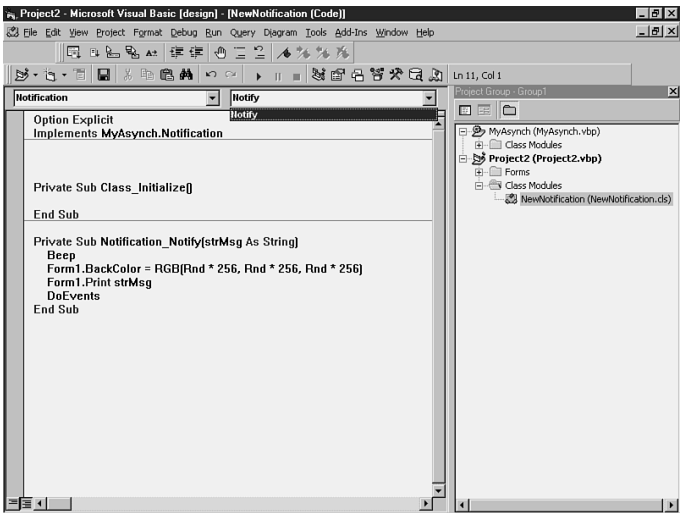


FIGURE 12.23
After you have typed the `Implements` declaration in your code, the `Interface`'s members are visible in your code window.

Notice in the listing that the members that this example implements through the `Interface` are all declared as `Private`. This is because you want any client application that uses your class to go through the `Interface` definition to reach this class's implementation of the `Interface` members.

After you have implemented the `Interface` in another class and implemented that class's own specific methods, properties, and events, you can use the implementing class and its accompanying `Interfaces` in client code, as discussed in the following section.

Methods for Using the `Interface` Class in a Client

When a client needs to refer to elements of a `Class` object that are implemented through an `Interface` class, the client must refer to the `Interface` by name to use the `Interface` elements.

There are two basic methods for referring to `Interface` elements within a class from client code. Both these methods assume that you have instantiated an object from a class that uses an `Interface` for all or part of its functionality:

- Declare an object variable in the client whose type is the `Interface` class and set that object variable to point back to another object that you have already instantiated from the server's implementing class. You can then manipulate the `Interface`-type variable to manipulate the `Interface`-provided elements of the implementing client object.
- Write wrapper routines in the client to manipulate various elements of the `Interface`. Each such routine will take as its parameter an object whose type is the `Interface` class. When you call the routines, however, you pass an instantiated object from the server's implementing class.

The following two sections discuss these methods.

Method A: Using an Object Variable in the Client

This method of referring to an object's `Interface` in client code relies on the fact that you can set a variable whose type is the `Interface` class to point to an instantiated object from a class that implements the `Interface`. After you have done this, any reference in your code to the `Interface` object variable actually refers to the instantiated implementing object.

Continuing the example from the previous sections, recall that the `Automobile` class implements the `IVehicle` interface. If you want to manipulate elements of the `Automobile` class in `autFord` that are provided by the `IVehicle` interface, you must declare a second variable of type `IVehicle` and point that variable to refer back to `autFord`.

LISTING 12.19

USING AN OBJECT VARIABLE OF THE INTERFACE TYPE TO GAIN ACCESS TO AN OBJECT'S INTERFACE ELEMENTS

```
Option Explicit

'STEP 1)
'DECLARE AND INSTANTIATE OBJECTS FROM
'THEIR BASE CLASSES
Private autFord As New Automobile

'STEP 2)
    'Declare a variable from the
    'appropriate interface class
```

```
    'to accompany the base-class
    'object instantiated
    'above (but DON'T instantiate
    'with As New)
Private vhlFord As IVehicle

Private  Sub Form_Load()

'STEP 3)
    'set a reference
    'to the variable derived from the
    'Interface class
    Set vhlFord = autFord

'STEP 3A)
    'Following line displays name of
    'Base Class,
    'which is "Automobile" in this case —
    'and NOT "IVehicle"
    Debug.Print TypeName(vhlFord)
End Sub

'STEP 4)
    'Write code that manipulates
    'the interface by manipulating
    'the appropriate
    'interface object variables.
Private Sub Command1_Click()
    MsgBox "Traveled " & _
        vhlFord.Travel(0, 0, -1, 1)
End Sub
```

Listing 12.19 shows the steps you need to take to use this method to gain access to the `Interface`-provided elements of a class object. The steps are as given in Step by Step 12.4 (step numbers are keyed to the numbers in the example of the listing).

STEP BY STEP

12.4 Using an Object Variable of the `Interface` Class

1. Declare and instantiate an object from the `Client` class. In the example, you use the `Automobile` class as described in previous sections, and give the resulting object instance the name `autFord`. Note that the example uses the `As New` keyword to instantiate the object at the same time that you declare it.

2. Declare an uninstantiated object that uses the `Interface` class as its type. In the example, you declare a second variable (named `vhlFord` in the example) of type `IVehicle` (recall that the `Automobile` class implements `IVehicle`). Make sure that it's uninstantiated. In other words, be sure that you don't use the `New` keyword when you declare it. This is because you will later set this variable to point to the `autFord` variable.
3. Use the *Set = syntax* to set the uninstantiated variable that you derived from the `Interface` class (`vhlFord` in the example) to point to the object that you instantiated from the `Client` class (`autFord`). This means that any reference to the `Interface`-type variable (`vhlFord`) will actually be a reference to the implementation of the `Interface` in the instantiated variable (`autFord`). To illustrate this point, the example code (see 3A) checks the `TypeName` of the variable `vhlFord`. Although you might expect `TypeName` to be "`IVehicle`", it actually turns out to be "`Automobile`".
4. Programmatically manipulate the members of the object declared from the `Interface` class (`vhlFord`'s members in the example). When you manipulate these members, you are actually manipulating the previously instantiated object (`autFord` in the example).

In your client code, you use the `As New` declaration to instantiate an object reference to the `Automobile` class named `autFord`. Then, you declare (but *don't* instantiate with `New`!) a second variable from the `IVehicle` interface named `vhlFord`. In your code, use the `Set` statement to cause `vhlFord` to point to `autFord`. After that, any reference to `vhlFord` actually refers to `autFord`.

Method B: Using Wrapper Routines in the Client

This method enables you to pass an object that has been instantiated from the `Client` class to a "wrapper" routine. The wrapper routine accepts a parameter whose type is of the `Interface` class. Even though its type is of the `Interface` class, this parameter provides a reference inside the wrapper routine to the original object, but only in regard to its features that are implemented in the `Interface` class.

LISTING 12.20

Using Wrapper Functions to Gain Access to an Object's Interface Elements

```
Option Explicit

'STEP 1)
'DECLARE AND INSTANTIATE OBJECTS FROM
'THEIR BASE CLASSES
Private autFord As New Automobile

'STEP 2)
    'Create procedures that
    'take parameters whose
    'type is that of the
    'Interface class and manipulate
    'that parameter as desired.
Private Function VehicleTravel _
         (vhl As IVehicle, x1, y1, x2, y2) As Double
    VehicleTravel = vhl.Travel(x1, y1, x2, y2)

'STEP 2A)
    'Following line displays name
    'of Base Class,
    'which is "Automobile,"
    'instead of the interface class
    'name, "IVehicle"
    Debug.Print TypeName(vhl)
End Function

'STEP 3)
    'Call the wrapper
    'procedures when you need to
    'manipulate the base class
    'object through its interface.
    'This calling code should pass the
    'base class object variable
    'to the procedures, and NOT
    'the variables based on the
    'interface class.
Private Sub Command2_Click()
    MsgBox "Traveled " & VehicleTravel(vhl747, 0, 0, 3, 4)
End Sub
```

Listing 12.20 shows the steps you need to take to use this second method to gain access to the `Interface`-provided elements of a `Class` object. The steps are as given in Step by Step 12.5 (step numbers are keyed to the numbers in the example of the listing).

STEP BY STEP

12.5 Using Client Wrapper Routines to Refer to an Interface Class Object

1. Declare and instantiate an object from the `Client` class. This is the same operation as performed in step 1 of the preceding method, and the same comments apply. You also use the same `Client` class as in the example, `Automobile`, to instantiate the variable `autFord`.
2. Create one or more procedures that take an object parameter whose type is the `Interface` class. Such a procedure can then manipulate the `Interface`-derived elements of this object. In the example, the procedure takes a parameter whose type is `IVehicle` (recall that `IVehicle` is implemented by the `Automobile` class). Notice that section 2A in the example checks the actual `TypeName` of the object parameter. If you run this code, you will discover that the object's type is not the type of the `Interface` class (`IVehicle`), but rather the type of the object that was passed from the calling routine (`Automobile`), as described in the next step.
3. Call the procedures created in step 2 by passing the instantiated object (`autFord` in the example) of the `Client` class as a parameter. When you pass this object to the procedure, the `Interface` manipulation code in the procedure can then access the `Interface`-implemented elements of the object.

The use of a wrapper routine typically makes your code cleaner and easier to maintain.

PROVIDING ASYNCHRONOUS CALLBACKS

- Create callback procedures to enable asynchronous processing between COM components and Visual Basic client applications.

This chapter has already discussed how to use events to provide a callback mechanism between a class object and its clients (see the section titled "Raising the Event and Implementing Callbacks in a Class Object").

This section and the following sections discuss a somewhat more complicated way to allow an ActiveX client to receive asynchronous callback notifications from objects. This second method requires the client and the server object to share a *callback object*. A callback object is an object that client and server share so that they can communicate with each other.

Because client and server share the callback object, they must agree on the structure of the object (that is, its properties and methods). A callback object is therefore a perfect candidate for an `Interface` as discussed in the previous sections. Both client and server programs (and therefore programmers!) must cooperate for the callback object to work. The server will define the callback object's `Interface`, but the client will actually implement the `Interface` in another class of its own, instantiate an object from that implementation, and pass the instantiated object to the server. The server can then manipulate the callback object to provide notifications to the client.

A client-server callback object's lifetime goes through the following steps:

1. The server provides an abstract `Interface` that gives an abstract definition of the callback object (that is, its methods and properties). The server, however, does *not* typically implement the callback object. The `Interface` defines at least one callback method.
2. The client implements the callback object from the abstract definition given by the `Interface`. The client defines how the callback method that it derived from the `Interface` will function.
3. The client instantiates the callback object from its implementation of the `Interface`.
4. The client calls a `Server` method, passing the callback object as an argument.
5. The server receives the callback object and sets a classwide variable to point to the object. This makes the callback object available throughout the `Server` class's code.

6. When the server needs to notify the client, it invokes the callback method of the callback object.
7. The callback method then behaves in whatever manner that the client has specified in its implementation—typically providing some sort of notification that the client can use.

To make these seven steps work, the server programmer and the client programmer must perform three major tasks:

- The server programmer provides the `Interface` that defines the callback object class's properties and methods. One method is typically considered the callback method.
- The client programmer implements the callback class from its `Interface` definition and then passes an instance of the resulting object to the `Server` object by calling the appropriate method or methods of the server.
- The server programmer provides a method that receives an instance of the callback object and then assigns a classwide object variable to point to this instance. The server will call the object variable's callback method whenever it needs to notify the client.

The following sections discuss each of these tasks.

Providing an `Interface` for the `Callback` Object

If you want to give your server's clients the capability to use callback objects, you must define an `Interface` for the callback object in a `Server` class.

To define the callback object's `Interface`, you must take the following steps:

1. Create a class in your server project and name it appropriately (remember the convention of using the letter *I* as a prefix to `Interface` class names).
2. Set the class `Instancing` property to `Global MultiUse`.
3. Define the `Interface` class's members (the properties and methods).

As you will recall, an `Interface` is a blank class template, so it's not a difficult proposition to define its properties and methods. Moreover, a callback class really only requires a single method to implement the callback. An `Interface` definition of a callback class could therefore be as simple as the code in Listing 12.21.

LISTING 12.21

A CALLBACK INTERFACE CLASS

```
'The Class name = INotification
Option Explicit

Public Sub Notify(msg As String)
End Sub
```

Notice that the code in the listing specifies a `String` parameter. This would allow the server to send a message back to the client as part of the notification. It is up to you whether you want to provide such features in your notification method.

The next section discusses how a client programmer could use this `Interface` to implement a callback object and pass it to the server.

Implementing the `Callback` Object in the Client

When you are programming a client application and want to use the callback functionality that an `Interface` provides, you must take the following steps outlined in Step by Step 12.6 to use a callback object in your application.

STEP BY STEP

12.6 Implementing a `Callback` Object in the Client

1. Add a class to the client application and name it appropriately.
2. Make sure that the `Instancing` property of the new class is `Private`. (If the client project is not an ActiveX EXE or ActiveX DLL, the `Class Instancing` property will not be available to you, and `Instancing` will be `Private` by default.)

3. Implement the server's callback Interface in this class (see Listing 12.22). Put appropriate notification code in the callback's notification method.
4. Instantiate copies of the callback object (should be `Public` variables) and call the appropriate methods of the server class objects that will receive instances of the callback object as parameters (see Listing 12.22).

Listing 12.22 gives the code in the class module that you would add to the client. As mentioned in step 3, this `Client` class implements the server's `Interface` class, `INotification`, which was discussed in the section titled "Providing an Interface for the `Callback` Object." Because `INotification` has only one member, the `Notify` method, you need only to write code to implement that method.

The `Notify` method will be called from inside the server to send a notification back to this client. Because this client implements the `Notify` method, it can therefore specify the exact way in which it will be notified.

Recall from the preceding section that the `Interface` for `INotification` specifies that the `Notify` method will accept one `String` parameter, so you must implement the `String` parameter here.

LISTING 12.22

IMPLEMENTING THE INTERFACE FOR A CALLBACK IN A `Client` CLASS

```
'Class Name = CallBack
Option Explicit

Implements INotification

Private Sub Inotificatiion_Notify(msg As String)
    MsgBox msg
    'or do something else with msg
End Sub
```

In Listing 12.23, the client code instantiates a server class object, instantiates an object from the callback class that you defined in Listing 12.22, and then passes the resulting callback object to the server object's `LongProcess` method.

LISTING 12.23

PASSING A Callback OBJECT TO A SERVER OBJECT

```
Public objServer As New MyServer.MyClass
Public cbCurrent As CallBack
.
.
Set cbCurrent = New CallBack
objServer.LongProces cbCurrent
```

The server's `LongProcess` method is specifically designed to accept an object that's been instantiated from a class that implements the `INotification Interface` (see the section titled "Manipulating the `Callback` Object in the Server"). After the server has received the object, it may at some point call the object's `Notify` method—which you have implemented in the client to provide a client-side notification.

Manipulating the `Callback` Object in the Server

To use a callback object in your server, you must take steps outlined in Step by Step 12.7.

STEP BY STEP

12.7 Using a `Callback` Object in the Server

1. Declare a `Public` variable in the `Server` class where you will be receiving the `Callback` object. The variable will be of the same type as the `Interface` class and will not be instantiated (see Listing 12.24). You will use this variable as a classwide pointer to `Callback` objects that client applications pass to this server (see the following step).
2. Write a method that receives a ByVal parameter whose type is the type of the Interface class. This parameter holds the callback object that the client is passing to the server. Within this method, set the Public variable declared in the

NOTE

`Callback` Object Should Be Public

Note the importance of using a `Public` variable to hold the callback object both in the server and in the client (see preceding section). If the callback object is not `Public`, you will receive a compiler error, because it's illegal to pass a `Private` object to a class method and also illegal to set a method's object parameter to a `Private` object.

In addition, you must always remember to mark the callback object parameter as `ByVal`.

preceding step to point to the callback object parameter. Pointing the Public variable to the parameter will make the object in the parameter available throughout the entire class (see Listing 12.24).

3. Use the Public variable (which now points to the callback object) to call the object's notification method when your server needs to send a notification to the client. This will cause the method to run on the client side (see Listing 12.25).

Listing 12.24 shows code in the `Server` class that will receive a `Callback` object, as described in steps 1 and 2. In the General Declarations section, you declare (but don't instantiate with `New`) a `Public` object variable named `cbObject` whose type is `INotification`, the same as the `Interface` definition that client and server are using for the `Callback` object class.

A method of the server (named `LongProcess` in the example) receives a parameter whose type is `INotification` (the `Interface` type). Using the `Set` statement, make the `Public` variable `cbObject` point to the parameter. Because `cbObject` is a `Public` variable, it's visible in the entire class, and thus effectively makes the `Callback` object parameter available throughout the class.

LISTING 12.24

DECLARING AND RECEIVING THE `Callback` OBJECT IN THE SERVER

```
'General Declarations of a server class
Public cbObject As INotification
.
.
Public Sub LongProcess(ByVal cbCurr As _
    INotification)
    Set cbObject = cbCurr
    .
    .
    .
End Sub
```

Listing 12.25 shows a line of code elsewhere in the `Server` class (perhaps just farther down in the `LongProcess` method's code). This code calls the `Notify` method of the `Public` object class, `cbObject`.

LISTING 12.25

USING THE `Callback` OBJECT'S METHOD TO NOTIFY THE CLIENT

```
.
.
.
'somewhere in your server class code:
obObject.Notify "The sky is falling!"
.
.
.
```

Recall from Listing 12.24 and from steps 1 and 2 in the preceding Step by Step that `cbObject` really points to the callback object parameter that the client passed to this server when it called the `LongProcess` method. Calling this method will therefore run the code that the client implemented for the callback object, as described in "Implementing the `Callback` Object in the Client."

REGISTERING AND UNREGISTERING A COM COMPONENT

When your COM component is ready for distribution, you will want it to be permanently listed in the Windows Registry on your development workstation. You will also want the component to be registered on the workstations belonging to users to whom you will distribute the component.

Normally, the setup applications created by Setup and Deployment Wizard will take care of the details of registration automatically. The following sections provide "under-the-hood" information for situations when you want more control over the registration of your component.

Registering/Unregistering an Out-of-Process Component

You can register an ActiveX executable in several ways (some of them inadvertent!):

- Compile it (obviously works only on the developer's workstation).
- Run it standalone—keeps on running after registering itself.
- Run it standalone with the `/REGSERVER` argument—terminates as soon as it registers itself.
- Install it with a setup routine created by Setup and Deployment Wizard.

The Setup and Deployment Wizard is, of course, the recommended way to install components.

You can remove your ActiveX executable from the Registry by

- Running it standalone with the `/UNREGSERVER` argument.
- Editing the Windows Registry (not recommended!).
- Running the Uninstall procedure from Windows (use the Add/Remove applications icon in Control Panel). This option works if you've installed the component using a setup routine that was created with Setup and Deployment Wizard.

Running the Windows Uninstall procedure is preferable, when available.

Registering/Unregistering an In-Process Component

You can permanently register an in-process component on the developer's workstation by compiling it.

Once again, you may also let Setup and Deployment Wizard create a setup routine for your component that will automatically register the component when users run the setup routine on their systems.

To register an in-process component from outside the development environment without a setup routine, you can run a utility called REGSVR32.EXE against the DLL file. REGSVR32.EXE is distributed on Visual Studio 6.0's installation CDs.

To use REGSVR32 to register an in-process component called, say, SPORT.DLL, you would run it from a command line as follows:

```
Regsvr32 sport.dll
```

To unregister your in-process component, you can run REGSVR32.EXE against it with the `/u` option in the command line, as in the following example:

```
Regsvr32 /u sport.dll
```

If you've installed your component using Setup and Deployment Wizard, you can uninstall it using the normal Windows application removal procedures.

Sending Messages to the User from a COM Component

- Implement messages from a server component to a user interface.

Depending on its function, your COM component might or might not have any user interface. If it does require user interface, that interface will have to be based on one or more forms included in the server component project. You must be aware of the way each form affects the lifetime of the server component application. When the server component is an in-process server component, you must also be aware of the effect your form has on the client application.

It is important to keep in mind that the forms and other elements of the user interface, such as message boxes, belong to your COM component, and not to the client application. Although this is an obvious fact, its consequences might not be so obvious. When an out-of-process server component displays its interface, for example, that interface might not be on top of other windows that the client is displaying.

Managing Forms in an Out-Of-Process Server Component

In an out-of-process server component, a form may be the startup form for the server component application's project. You must unload this form and any other forms before the server component can be unloaded.

Because of this, you shouldn't use `formname.Hide` in your out-of-process server component's code when it is completely done with a form. Instead, the out-of-process server component application should load the form when it needs it, and unload it whenever it does not need to be visible. If you have copies of the form in memory, an out-of-process server component will continue to run even after it is not needed.

Managing Forms in an In-Process Server Component

A form is never the entry point into an in-process server component, because the server component runs in the same process space as the client, and the client itself initiates and terminates the server component.

Although you can generally display modal forms as needed in an in-process server component, modeless forms present more problems. The only client applications that will support modeless forms in an in-process server component as of this writing are as follows:

- Clients that are written in VB5 or VB6.
- Clients that use Visual Basic for Applications 5.0 or later. This includes the Microsoft Office 97 suite and later versions, as well as any third-party applications that carry the Visual Basic Technology logo.
- Internet Explorer 4.0 and above.

When you program a server component, you can't foresee which clients will attempt to use it, so you can't tell whether your server component's client supports modeless server component forms. If you want to be very cautious, you can just avoid ever displaying a modeless form in an in-process server component.

For more flexibility, however, you can verify the `App` object's `NonModalAllowed` property in the server component's code. If the property is `True`, the current client supports modeless forms in the server component. You might write code similar to Listing 12.26 so that your server component could display a form either modelessly or modally, depending on what the client allows.

LISTING 12.26

DETERMINING WHETHER YOUR IN-PROCESS SERVER COMPONENT CAN SAFELY DISPLAY A MODELESS FORM

```
If App.NonModalAllowed Then
    frmMsg.Show vbModeLess
Else
    frmMsg.Show vbModal
End If
```

Even if an in-process server component still has forms loaded, it may be able to unload anyway, providing all forms are invisible and certain other conditions are fulfilled.

Note that an out-of-process server component can't terminate with any forms loaded, even though they are invisible.

CHOOSING THE RIGHT COM COMPONENT TYPE

You may encounter several questions on the certification exam that expect you to know the appropriate type of COM component for a particular solution. Here is a list of the major types of COM components and the things that they do best:

- **In-process components (ActiveX DLLS)**. Best for performance.
- **Out-of-process components (ActiveX EXEs)**. Best for background, asynchronous processing. Best for exposing an application's object model to clients.
- **ActiveX controls**. Best for objects that have mainly to do with standardizing parts of the user interface.
- **Active documents**. Best for objects that need to run across the Internet or an intranet.

Some less-than-positive considerations for various COM component types:

- **In-process components (ActiveX DLLS)**. Problems if you give them a user interface. Crashes clients more readily than EXEs.

NOTE

`App.NonModalAllowed` Gives Incorrect Results in the VB Debugging Environment Typically, you can test your component against another application by running your app from the VB environment and specifying the test client from the Debug dialog box. The VB environment will always return `App.NonModalAllowed` as `True`, however, regardless of whether the external application really does support nonmodal forms in a component.

The only way to really find out whether a potential client supports nonmodal forms is to compile your application and test the client against the compiled version—or against a small compiled test application that checks the value of `App.NonModalAllowed`.

- **Out-of-process components (ActiveX EXEs)**. Slower than DLLs or controls.
- **ActiveX controls**. Perform worse than DLLs, but better than EXEs.
- **Active documents**. Performance not good. Not particularly helpful in exposing an object model.

Implementing Scalability Through Instancing and Threading Models

Although the exam objectives for COM components do not mention scalability directly, you can expect one or more questions on the exam that require you to know something about the impact of certain COM component choices for the *scalability* of a VB solution.

Recall that scalability refers to the ease with which a solution can be transferred to a more demanding environment than the environment for which you originally intended it. Scalability issues typically have to do with transferring the application from desktop to enterprise, or adding more users or other clients to the application's environment.

Following are several considerations for COM component scalability that may appear in the certification exam:

- Classes in out-of-process components may have their `Instancing` property set to `SingleUse`.

  `SingleUse` will guarantee that each object instantiated from the class will run in its own address space. As the number of connections to a particular COM component on a system rises, this will eliminate contention between different clients for a component, because there will only be one object per client.

  Of course, the trade-off for `SingleUse` instancing is that more server resources will be required to instantiate each new copy of the object every time that a client requests a copy of the object.

- The different threading choices discussed in "Managing Threading in Out-of-Process Components" can have an impact on scalability for out-of-process servers.

  Multithreading shows the best advantage when there are either multiple physical processors on the server, or when the processes that clients are likely to request from a component will be of uneven length and will be blocked most of the time (that is, they will wait on file I/O or communications port activity).

  In general, there is less advantage to a multithreaded COM component solution when there is only one physical processor on the system and the component's object processes are all of relatively equal length and do not block system resources for very long with activities such as file access or communications. This is because such a scenario will cause about the same amount of work for the single processor either way (multi- or single-threaded), with the added disadvantage for multithreading that the server must add more overhead for each thread.

As a rule of thumb, you will get the best performance out of a COM component if you choose options that will guarantee close to *one active thread per physical processor*. This does not mean only one thread *running* for the entire component, but rather one *active* thread. If three threads are running at the moment, for example, but two of them are blocked waiting on file I/O, there is only one active thread.

On single-CPU systems, therefore, you will see that you will need to do some investigation and testing to determine whether multithreading is desirable. For best scalability, choose round-robin threading with a low number of threads. Single-use instancing can also be a good choice for scalability, as long as you know that you have a powerful server.

Under-the-Hood Information About COM Components

Although there are no explicit VB exam objectives that discuss COM internals, you may encounter several questions that assume a knowledge of some of COM's inner workings.

In particular, you may need to know about the `IUnknown` and `IDispatch` interfaces, and about *vtable binding*.

`IUnknown` and `IDispatch` are object model interfaces, such as those discussed earlier in the section titled "Using Interfaces to Implement Polymorphism." As a VB programmer, however, you typically never need to know about `IUnknown` or `IDispatch`, which are provided automatically with every COM component. These interfaces are implemented internally and automatically by VB for every class object that you create in VB.

The purpose of `IUnknown` is to keep track of open references to an object and to provide the caller with the functionality that the object supports.

`IUnknown` has three methods. The details of their function are beyond the scope of the certification exam, but their names are not:

- **`AddRef`** This method increments an internal reference counter every time a client creates a new reference to an object.
- **`Release`** This method decrements the internal reference counter every time a client releases a reference to an object. When the counter reaches zero, the object can be destroyed.
- **`QueryInterface`** This method finds out whether an object supports a particular interface and, if it does, makes that interface's functions available to the calling application.

The purpose of `IDispatch` is to provide a standard way to access the members of a given interface for a particular object.

`IDispatch` has four methods. Once again, their names are the important thing to know for the certification exam:

- **`GetTypeInfoCount`** Returns a non-zero value if type information is available for this interface.
- **`GetTypeInfo`** Returns type information for the interface.
- **`GetIDsOfNames`** Finds the internal ID (known as the dispatch ID) of a particular property of method.
- **`Invoke`** Executes a method or accesses a property.

Vtable binding is a particular type of early binding that VB typically uses when you declare objects in client applications using the name of a registered object type. For purposes of the certification exam, you should just remember that questions about vtable binding will be the same as the same question asked about early binding.

CHAPTER SUMMARY

KEY TERMS

- ActiveX
- ActiveX control
- ActiveX document
- Apartment-model threading
- Callback
- Class
- CLS file (.CLS)
- COM
- COM component
- Early binding
- `IDispatch`
- `Instance`
- Instantiate
- `Interface` class
- `Iunknown`
- Marshaling
- Member
- Modal
- Modeless
- Object variable
- Object
- vtable binding
- Windows Registry

This chapter has covered the following topics:

- Overview and definition of COM components, including ActiveX DLLs, ActiveX EXEs, ActiveX controls, and Active documents
- Choosing the right COM component for the job
- Choosing the appropriate threading model for a COM component
- Choosing the appropriate instancing property for COM component classes
- Using a COM component to implement business rules and logic
- Programming with VB class modules
- Implementing an object model within a COM component
- Sending error information from a COM component to a client computer
- Using Visual Component Manager to manage components
- Using callback procedures for asynchronous processing between COM components and VB client applications
- Registering and unregistering a COM component in the Windows Registry
- Implementing messages from a server component to a user interface

APPLY YOUR KNOWLEDGE

Exercises

12.1 Using Class Modules to Create a COM Component That Implements an Object Model for Business Logic

Estimated Time: 45 minutes

In this exercise, you create an example of an in-process component that could implement some simple business rules. You also create a second project to test the component.

1. Begin a new ActiveX DLL project in VB.
2. Name the default class **Credit** (right-click the `Class` object in Project Explorer to access its properties, including the name).
3. Give the `Credit` class two read/write properties, `CustID` and `TransactionAmount`. Create property procedures for each property and declare `Private` variables to hold the property values:

```
Option Explicit
Private m_CustID As String
Private m_TransactionAmount As Double

Public Property Get CustID() As String
    CustID = m_CustID
End Property
Public Property Let CustID(strNewCustID As
➥String)
    m_CustID = strNewCustID
End Property

Public Property Let
TransactionAmount(dNewAmount As Double)
    m_TransactionAmount = dNewAmount
End Property
Public Property Get TransactionAmount() As
➥Double
    TransactionAmount = m_TransactionAmount
End Property
```

4. Create a read-only property, `MaxAmtPerTransaction` (just omit the `Property Let` procedure to make it read-only), and declare a `Private` variable to hold its value. Initialize the `Private` variable in the class's `Initialize` event procedure:

```
'In General Declaration section
Private m_MaxAmtPerTransaction As Double

Public Property Get MaxAmtPerTransaction()
    MaxAmtPerTransaction =
➥m_MaxAmtPerTransaction
End Property

Private Sub Class_Initialize()
    m_MaxAmtPerTransaction = 100000
End Sub
```

5. Create an event for the class, `CustBalChange`. It will take two parameters:

```
Public Event CustBalChanged(CustID As String,
➥ChangeAmt As Double)
```

6. Create a method, `ValidateTransaction`. This method will act as a function and will return an Integer value, 1 if the transaction is acceptable, and 0 if it's unacceptable. The internal code for `ValidateTransaction` will compare the value stored for the `TransactionAmount` property with the value stored for the `MaxTransactionAmount` property. If the transaction amount doesn't exceed the maximum allowed, the transaction is accepted. A successful transaction will also cause the `CustBalChange` event to fire:

```
Public Function ValidateTransaction() As
➥Integer
    If (m_TransactionAmount <=
MaxAmtPerTransaction) Then
        RaiseEvent CustBalChanged(m_CustID,
m_TransactionAmount)
        ValidateTransaction = 1
    Else
        ValidateTransaction = 0
    End If
End Function
```

APPLY YOUR KNOWLEDGE

7. Make sure the project is named `BusRules` (set its name on the General tab of the Project, Properties menu option). Save the project as BusRules.VBP. The project name will be the component's name in the Windows Registry.

8. Now you will create a test client project and run it against the component that you have just created. Add a new standard EXE project to the current group (use the File, Add Project option on the VB menu). Make it the group's startup project by right-clicking the new project in Project Explorer and choosing Set As Startup from the resulting shortcut menu.

9. On the test project's default form, add two text boxes, two labels, and a command button, as shown in Figure 12.24. Name the text boxes `txtCustomerID` and `txtTransactionAmt`, respectively, and name the command button `cmdValidateTransaction`. Clear out the text properties of the text boxes.

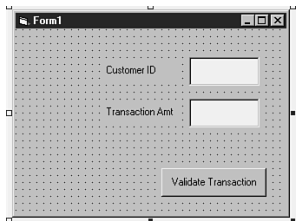


FIGURE 12.24
The form for the test project in Exercise 12.1.

10. Set a reference in the test EXE project to the `BusRules` component (Use the Project, References dialog box, find the `BusRules` component in the list of components, and check it on).

11. Declare a `Private` instance of the `BusRules.Credit` object (call it `objCredit`) in the test project's form using the `WithEvents` keyword and instantiate the object in the `Form_Load` event procedure:

```
Option Explicit
Private WithEvents objCredit As
BusRules.Credit
Private Sub Form_Load()
    Set objCredit = New BusRules.Credit
End Sub
```

12. In the code window for the test project, find the object variable's name in the left-hand drop-down list of objects. Selecting this object will automatically position the cursor in the event procedure for `CustBalChanged` (it's the only event exposed by the `Credit` class). Write code to react to the event:

```
Private Sub objcredit_CustBalChanged _
               (CustID As String,  _
                ChangeAmt As Double)
    MsgBox "Balance of customer " &  _
            CustID & " changed by " &
➥ChangeAmt
End Sub
```

13. In the `Click` event of the command button, place code to exercise the object model of `BusRules.Credit`:

```
Private Sub cmdValidateTransaction_Click()
    objcredit.CustID = Trim$(txtCustomerID)
    objcredit.TransactionAmount = CDbl(
➥txtTransactionAmt)
    Me.Cls
    If objcredit.ValidateTransaction Then
        Beep
        Me.Print "Approved"
    Else
        Beep
        Me.Print "Not approved"
    End If
End Sub
```

14. Run the application, attempting amounts both over and under the amount permitted by the `MaxTransactionAmount` property of the `Class` object.

APPLY YOUR KNOWLEDGE

Because you didn't take time to validate data properly, make sure that you have a numeric amount in the text box for Transaction Amount.

12.2 Setting the Threading Model for a COM Component

Estimated Time: 5 minutes

In this exercise, you view the threading options for in-process and out-of-process servers.

1. In VB, start a new ActiveX DLL project, choose Project, Properties from the VB menu, and view the threading options on the General tab.
2. Start a new ActiveX EXE project and view its threading options in the same way.

12.3 Controlling the Instancing of a Class Within a COM Component

Estimated Time: 5 minutes

In this exercise, you examine the instancing options available to classes in ActiveX DLL (in-process) and ActiveX EXE (out-of-process) components.

1. In VB, start a new ActiveX DLL project. Right-click its default class to bring up the shortcut menu, and choose Properties from the menu. Click the drop-down arrow next to the `Instancing` property in the Properties window, and note the options.
2. Start a new ActiveX EXE project. Right-click its default class to bring up the shortcut menu, and choose Properties from the menu. Click the drop-down arrow next to the `Instancing` property in the Properties window, and note the options. Which instancing options have been added over the instancing options for the ActiveX DLL?

12.4 Returning Error Information from a COM Component

Estimated Time: 45 minutes

In this exercise, you use the project/test project pair that you created in Exercise 12.1 to demonstrate the two techniques of reporting errors from a component to a client: raising errors directly in the component, or returning an error code from a method.

1. Use the same project group that you created in Exercise 12.1.
2. In the `Credit` class of the `BusRules` component, modify the `ValidateTransaction` method's code so that it looks like the following:

```
Public Function ValidateTransaction() As
Integer
    If m_TransactionAmount = 0 Then Err.Raise
       Number:=600, _
       Source:="ValidateTransaction", _
       Description:="Zero Transaction Amounts
➥not allowed."
    If m_CustID = "" Then
        ValidateTransaction = -1
        Exit Function
    End If
    If (m_TransactionAmount <= MaxAmtPer-
➥Transaction) Then
        RaiseEvent CustBalChanged(m_CustID,
➥m_TransactionAmount)
        ValidateTransaction = 1
    Else
        ValidateTransaction = 0
    End If
End Function
```

Notice that you are mixing error-handling styles here (not recommended for real applications, but good for the example). You are both raising an error to the client and also manipulating the return value of the method to reflect different problems that might occur: a blank customer ID or a zero transaction amount, respectively.

APPLY YOUR KNOWLEDGE

3. In the test project, modify the `Click` event of the command button as follows:

```
Private Sub cmdValidateTransaction_Click()
    objcredit.CustID = Trim$(txtCustomerID)
    On Error Resume Next
    objcredit.TransactionAmount = CDbl(
➥txtTransactionAmt)
    Me.Cls
    Dim iResult As Integer

    On Error GoTo ValidateTransaction_Error
    iResult = objcredit.ValidateTransaction
    If iResult = 1 Then
        Beep
        Me.Print "Approved"
    ElseIf iResult = 0 Then
        Beep
        Me.Print "Not approved"
    ElseIf iResult = -1 Then
        Beep
        Me.Print "Error in
ValidateTransaction method"
    End If
Exit_Validate_Click:
    Exit Sub
ValidateTransaction_Error:
        MsgBox "Error in " & Err.Source & ".
➥Error #" & Err.Number & ": " &
Err.Description
        Resume Exit_Validate_Click
End Sub
```

   Notice that the code both checks the return value of the function for an error condition (–1) and uses an error trap to see whether the component has raised an error.

4. Set Break option to Break on Unhandled Errors using the General tab of the Tools, Options dialog box.

5. Run and test the application with various combinations of blank customer IDs and transaction amounts of zero.

12.5 Using Visual Component Manager

Estimated Time: 45 minutes

In this exercise, you publish a project in VCM, and then you find and use that project in another VB application.

> **NOTE**
> **Refer to Figures in Text** This exercise includes no figures. For illustrations of the tasks discussed here, refer to the figures under the section in this chapter on Visual Component Manager.

1. Use the same project group that you created for Exercise 12.4 or Exercise 12.1.

2. In the test project, remove the reference to the DLL project.

3. Make sure that the Visual Component Manager icon appears on your toolbar. If not, choose Add-ins, Add-in Manager, find VCM in the list of add-ins, and mark it as Loaded. Also mark Load on Startup if you want it to be loaded every time.

4. Right-click the DLL project name in Project Explorer and choose Publish. Under the submenu, choose Build Outputs.

5. Skip the first screen of the VCM Publish Wizard. On the second screen (Select Repository/Folders), choose the appropriate folder in the Local database repository and adjust the name of the component if necessary. Click the Next button.

6. Read the Title and Properties screen. Make no changes. Click the Next button.

APPLY YOUR KNOWLEDGE

7. On the More Properties screen, type an appropriate description (such as **Business rules**) for the component. Add two keywords, `Business` and `Credit`, by clicking the plus button (+) and typing the keywords. After you have added the keywords, click the Next button.
8. Read the Select Additional Files screen, but make no changes. Click the Next button.
9. Read the COM Registration screen, but make no changes. Click the Next button.
10. On the Finished screen, click the Finished button.
11. Now your component is ready to use in VB projects. Start a new EXE project.
12. Click the VCM icon on the VB toolbar, and make sure that the highest node in the Folders tree of VCM is selected.
13. Click the Search (binoculars) icon on the VCM screen.
14. In the resulting Search dialog box, enter **Credit** in the box labeled Containing Text. And check the box labeled `Containing text in a keyword`. Then click the Find Now button.
15. Your component should show up at the bottom of the Search dialog box. Right-click your component's icon and choose Add to Project from the resulting drop-down menu, responding with default answers to any message boxes that appear.
16. You should see a message that your component was successfully added to the project. Because you chose the Build Outputs option in step 4, your project now contains a reference to the component, which you can view in the Project, References dialog box.

12.6 Implementing a Callback Procedure for Asynchronous Processing

Estimated Time: 35 minutes

In this exercise, you implement a callback between client and component using an `Interface` class to provide a template for a callback notification object.

1. Create an ActiveX DLL project. Name the project `MyAsynch`.

   Use the default class as the `Interface` to implement the callback object. Name the class `INotification`. Its entire contents will just be the empty template for a `Notify` method:

   ```
   Option Explicit

   Public Sub Notify(strMsg As String)
   End Sub
   ```

2. Add a class and name it `MainApp`.
3. Within `MainApp`, declare a `Public` object variable of the `INotification Interface` type:

   ```
   Option Explicit
   Public pobjNotification As INotification
   ```

4. Also in `MainApp`, create a method, `DoSomething`, that receives an object parameter of the `INotification Interface` type and sets the `Public` object *variable* to point to the object *parameter*. Let the method invoke a time-consuming routine:

   ```
   Public Sub DoSomething(objNotification As
   ➥INotification)
           Set pobjNotification = objNotification
           LongTimeProcess 5  'number of seconds
   ➥to run
   End Sub
   ```

5. Write the time-consuming routine invoked by the method. When the routine finishes, let it call the `Notify` method of the `Private` object variable.

APPLY YOUR KNOWLEDGE

```
Private Sub LongTimeProcess(iSeconds As
➥Integer)
    Dim dtStart As Date
    Dim dtEnd As Date
    Dim dSecondsInADay As Double
    dSecondsInADay = CDbl(24) * CDbl(60) *
➥CDbl(60)
    dtStart = Now
    dtEnd = dtStart + CDbl(iSeconds) /
➥dSecondsInADay
    Do While dtEnd > Now
        DoEvents
    Loop
    pobjNotification.Notify "Done after " &
➥iSeconds & " seconds."
End Sub
```

Because a client will implement the specific behavior of the `Notify` method, the `Notify` method will do whatever the client's implementation has specified.

6. Add a new standard EXE project to the group. On the surface of its default form, place a command button and a text box, as shown in Figure 12.25.

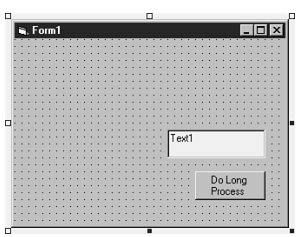


FIGURE 12.25
The form for the test project in Exercise 12.6.

7. Add a reference to the `MyAsynch` component (see Exercise 12.1 for details on how to do this).
8. Add a class to the current EXE project that implements the notification interface. Notice that class modules declared in a standard EXE have no `Instancing` property. Name the class `NewNotification`. Have it implement `MyAsynch.INotification`.

   Make the `Notify` method of `NewNotification` do something that can be perceived, such as beep or display a message. The entire implementation of `NewNotification` should look something like this:

```
'In the NewNotification Class module
Option Explicit
Implements MyAsynch.INotification

Private Sub Notification_Notify(strMsg As
➥String)
    Beep
    Form1.BackColor = RGB(Rnd * 256, Rnd *
➥256, Rnd * 256)
    Form1.Print strMsg
    DoEvents
End Sub
```

9. In the command button's `Click` event, declare and instantiate instances of the `NewNotification` class and of the `MySynch.MainApp` component class. Then call the `DoSomething` method, passing to it as an argument the instance of the `NewNotification` class that you just created:

```
'In Form1
Private Sub cmdDoLongProcess_Click()
    Dim objMainApp As New MyAsynch.MainApp
    Dim objnotification As New NewNotification
    objMainApp.dosomething objnotification
End Sub
```

10. Make sure that the test project is the group's startup project, and make sure that the error-handling option is Break in Class Module. Then run. Invoke the method by clicking the command button. Type something into the text box while waiting for the notification. This will prove that the client and component are truly asynchronous to each other.

APPLY YOUR KNOWLEDGE

```
'Class NewNotification
Option Explicit
Implements MyAsynch.Notification
Private Sub Notification_Notify(strMsg As
➥String)
    Beep
    Form1.BackColor = RGB(Rnd * 256, Rnd *
➥256, Rnd * 256)
    Form1.Print strMsg
    DoEvents
End Sub
'Form1
Private Sub cmdDoLongProcess_Click()
    Dim objMainApp As New MyAsynch.MainApp
    Dim objnotification As New NewNotification
    objMainApp.DoSomething objnotification
End Sub
' MyAsynch.MainApp
Option Explicit
Public probjNotification As Notification

Public Sub DoSomething(objNotification As
➥Notification)

        Set probjNotification = objNotification
        LongTimeProcess 5  'number of seconds
➥to run
End Sub

Private Sub LongTimeProcess(iSeconds As
➥Integer)
    Dim dtStart As Date
    Dim dtEnd As Date
    Dim dSecondsInADay As Double
    dSecondsInADay = 24 * 60
    dSecondsInADay = dSecondsInADay * 60
    Dim dSecondAsFractionOfDay As Double
    dSecondAsFractionOfDay = 1 /
➥dSecondsInADay
    dtStart = Now
    dtEnd = dtStart + iSeconds * 1 /
➥(CDbl(24) * CDbl(60) * CDbl(60))
    Do While dtEnd > Now
        DoEvents
    Loop
    probjNotification.Notify "Done after " &
➥iSeconds & " seconds."
End Sub
Option Explicit
'Class MyAsynch.Notification
Public Sub Notify(strMsg As String)
End Sub
```

12.7 Registering and Unregistering a COM Component

Estimated Time: 25 minutes

In this exercise, you experiment with registering and unregistering COM components in the Windows Registry.

1. Make sure that REGSVR32.EXE is on your system. It should be in the System directory under your Windows Install. If you don't see it, search for it on the installation media for VB6 or Visual Studio 6. After you find it, copy it to the System directory.
2. Open the ActiveX DLL project from Exercise 12.1. Compile the project by choosing File, Make *.DLL from the VB menu.
3. Fully exit VB and then re-enter. Start a new standard EXE project. In the Project, References dialog box, you should now note a reference to your project, because it has been entered in the Registry through the act of compiling it.
4. Close VB again. Unregister your component by running the following line from Windows' Start, Run menu option:

   ```
   Regsvr32 -u PathToComponent\ComponentName.dll
   ```

   where *PathToComponent* is the fully qualified drive and path to the folder where your compiled component resides, and `ComponentName.dll` is the name of the compiled file.
5. Return to VB and note that your component can no longer be found in the Project, References dialog box.

APPLY YOUR KNOWLEDGE

6. Exit VB and reregister your component. This time, use REGSVR32 by running the following line from Windows' Start, Run menu option:

   ```
   Regsvr32 PathToComponent\ComponentName.dll
   ```

   where, of course, *PathToComponent* is the fully qualified drive and path to the folder where your compiled component resides, and *ComponentName*.dll is the name of the compiled file. Run VB again and note that the component has returned to the Project, References list.

7. Now for the same steps with an ActiveX EXE: first, open the ActiveX EXE project from Exercise 12.x and compile it. This will register the component, just as compiling the out-of-process component did in step 2.

8. Unregister the ActiveX EXE component by running the following line from the Windows Start, Run menu:

   ```
   exepath\exename /UNREGSERVER
   ```

   where exepath\exename is the path and name of the compiled file.

9. Reregister your ActiveX EXE component by running it again from the Windows Start, Run menu, this time with the line:

   ```
   exepath\exename /REGSERVER
   ```

10. Unregister the ActiveX EXE component, and then reregister it by just running it from the command line without the /REGSERVER switch. When you run it without the switch, the EXE continues running. When you run it with the switch, the EXE runs only to register itself, and then unloads.

12.8 Implementing Messages from a Server Component to a User Interface

Estimated Time: 15 minutes

In this exercise, you display forms and message dialog boxes to the user of a component.

1. Use the project group that you created for Exercise 12.4.

2. To the ActiveX DLL, add a form and name it frmMsg. Add a label to the form, and change the label's font size to 24. Change the label's caption to Error.

3. In the Credit class of BusRules, add the following code to the top of the ValidateTransaction method:

   ```
   If App.NonModalAllowed Then
           frmMsg.Show
       Else
           frmMsg.Show vbModal
   End If
   ```

4. When you run the test application, note that the other messages from the method and the calling routine in the client continue to show while the form is onscreen, meaning that the form was called modelessly.

Review Questions

1. You want to create a read-only property. How would you define the Property procedure?

2. Why would you create a property using a Property procedure pair rather than a Public variable?

3. How would you create a new method in a class?

APPLY YOUR KNOWLEDGE

4. You have an ActiveX EXE program with a class declared as `SingleUse`. What effect could this have on the memory of the computer when multiple clients access this object?
5. You are writing the code that takes advantage of an event (`Done`) fired by another class named `SomeTask`. What line should you add to the General Declarations section of the module that will respond to the event?
6. What must you do to a class to make it into an interface?
7. When implementing automation components, which types of VB6 application can be used?
8. Your programmers are complaining that it takes too long to execute methods of Automation Server. You have already determined that the application is already tuned. What can be done to the Automation Server to speed up the calls between client and server?
9. What is the purpose of the `WithEvents` keyword? Write a declaration with this keyword.
10. What is the threading model for Visual Basic applications?
11. What is the difference between `GlobalMultiUse` and `MultiUse` instancing?
12. In what two different formats can you publish a component in Visual Component Manager?
13. What are the two basic techniques for sending error information from a COM component to a client?
14. What is the name of the utility that you can use to register COM components?
15. How can you tell whether it is safe for an ActiveX DLL to use nonmodal forms?
16. What is the difference between in-process and out-of-process COM servers?

Exam Questions

1. You are defining a new property. The property is meant to hold an employee's ID number, which is made up of two characters and four numbers. Which of the following property procedures would you have to define? (Select all that apply.)

   A. `Property Set`

   B. `Property Get`

   C. `Property Let`

   D. `Property EmployeeID`

   E. `Property New`

2. The following line of code is in the General Declarations section of a class module:

   ```
   Public Age as Integer
   ```

   Which of the following is true?

   A. The class defined by the class module will have a new property called `Age`.

   B. All modules in the same project as the class module will be able to access the variable.

   C. An error will occur because this is not a valid declaration.

   D. The value of `Age` can be accessed from any other project that knows about this class.

APPLY YOUR KNOWLEDGE

3. You are designing a class module that will be used as a standalone EXE server. Which of the following is true about the `Instancing` property for the class?

   A. The class must be defined `Private`.

   B. A `PublicNotCreatable` class must have another class that provides access to it.

   C. A program using a class defined as `GlobalMultiUse` needs to explicitly mention the class name to use its properties and methods.

   D. The class can be defined as `SingleUse`.

4. You have created a new property procedure in a class module. You have not specified its scope. What scope is it?

   A. `Private`

   B. `Friend`

   C. `Public`

   D. `Global`

   E. None. You must specify the scope for all properties.

5. Which of the following are valid settings for the `Instancing` property of a class in an in-process server? (Select all that apply.)

   A. `GlobalMultiUse`

   B. `MultiUse`

   C. `PublicCreatable`

   D. `SingleUse`

   E. `Private`

6. You have declared a new object variable of type `CAccount`, with the following line:

   ```
   Dim theAccount as CAccount
   ```

   What must you do before using this variable?

   A. Nothing. The variable is ready to use.

   B. `Set theAccount = New CAccount`

   C. `theAccount = New CAccount`

   D. `Set theAccount = CAccount`

7. What action should you not do in a class's `Initialize` event?

   A. Set default values

   B. Create dependent objects

   C. Set an object variable to a new copy of the class

   D. Open files or other resources to be used by the class

8. What is the syntax to assign the return value for a `Property Get` procedure named `LastName`, where `LastName` is a String?

   A. `Set LastName = ReturnValue`

   B. `Return ReturnValue`

   C. `Exit ReturnValue`

   D. `LastName = ReturnValue`

9. You are using a program that accesses a `SingleUse` ActiveX EXE program. After creating a second copy of an object defined by the program, what should you expect to happen?

   A. More memory is used.

   B. Memory is freed.

APPLY YOUR KNOWLEDGE

C. An error occurs. You can only run one copy of a `SingleUse` program.

D. Nothing.

10. What happens if you don't write code to react to all the events an object generates?

A. A runtime error occurs.

B. A compile-time error occurs.

C. No error occurs, but the events you do add code to will not run until you complete all event handlers.

D. No error occurs, and the events you react to will execute.

11. What happens if you don't write code for all the methods and properties of an interface?

A. A runtime error occurs.

B. A compile-time error occurs.

C. No error occurs, but the methods and properties you do add code to will not run until you complete all `Interface` procedures.

D. No error occurs, and the methods and properties you have added code for will execute.

12. You have an ActiveX Automation Server that displays forms. What must be done to have the application run unattended and support multiple threads?

A. You cannot have unattended Automation Servers.

B. There is nothing that must be done.

C. Remove the forms and code that refers to forms. Then select the Unattended Execution option.

D. Select the Unattended Execution option.

13. What does the Thread per Object option provide for an application?

A. This option is not available in the current release of Visual Basic.

B. Each `SingleUse` class will get its own thread.

C. Each `MultiUse` class will get its own thread.

D. Multiple objects will be in a thread as defined by the object count.

14. You have an `Application` class and a `Workspace` class. The users of your Automation Server must be able to create the `Application` but not the `Workspace` class. The only way the user should be able to use the `Workspace` class is through the use of a function of the `Application` class. How do you achieve this goal?

A. Define the `Application` class as `MultiUse` and the `Workspace` class as `Private` and return the object via a function of the `Application` class.

B. Define the `Application` class as `MultiUse` and the `Workspace` class as `GlobalMultiUse`.

C. Define the `Application` class as `SingleUse` and the `Workspace` class as `GlobalSingleUse`.

D. Define the `Application` class as `MultiUse` and the `Workspace` class as `PublicNotCreatable`.

E. Define the `Application` class as

APPLY YOUR KNOWLEDGE

`PublicNotCreatable` and the `Workspace` class as `MultiUse`.

F. This cannot be done in Visual Basic.

15. Which of the following statements will create a read-only property?

A.
```
Public FirstName as String
```

B.
```
Property Get FirstName() as string
        End Property
Property Let FirstName(strLast as string)
        End Property
```

C.
```
Property Get FirstName() as string
        End Property
Property Set FirstName(strLast as string)
        End Property
```

D.
```
Property Get FirstName() as object
        End Property
Property Let FirstName(strLast as object)
        End Property
```

E.
```
Property Get FirstName() as string
        End Property
```

F.
```
Public ReadOnly FirstName as string
```

G.
```
Friend FirstName as String
```

16. Which of the following statements is true about automation components?

A. They can only be implemented as DLLs.

B. They can only be implemented as executables.

C. They are neither executables nor DLLs.

D. They can be either executables or DLLs.

17. Your programmers are complaining that it takes too long to execute methods of Automation Server. You have already determined that the application is already tuned. What can be done to the Automation Server to speed up the calls between client and server?

A. Nothing can be done.

B. Implement the Automation Server as a DLL and run it locally.

C. Implement the Automation Server as an executable and run it locally.

D. Implement the Automation Server as a DLL and run it on a different machine.

18. Which of the following variable definition using the `WithEvents` keyword is correct?

A. `Private WithEvents m_obj as object`

B. `Private WithEvents m_obj as TextBox`

C. `Private WithEvents m_obj as Variant`

D. `Private WithEvents m_obj as New TextBox`

19. Which of the following statements is true about the `Implements` keyword?

A. It is used to provide a class interface to the Windows API.

B. It is used to define a C++ interface.

C. It denotes that the module will provide code to support the interface.

D. It is used to support implementation inheritance.

APPLY YOUR KNOWLEDGE

20. When implementing custom properties with `Get/Let/Set` procedures, you store the value of the property where?

    A. In a `Public` variable of the class

    B. In a `Private` variable of the class

    C. In a `Static` variable of the `Let` procedure

    D. In a `Static` variable of the `Get` procedure

21. How do you implement a collection's built-in features (such as Count, Add, Remove, and Item)?

    A. By writing wrapper procedures in the dependent class

    B. By writing wrapper procedures in the collection class

    C. By writing wrapper procedures in the parent class

    D. By invoking them directly from calling code

22. When implementing a collection in an application, where would you put the following statement?

    ```
    Private colStars as Collection
    ```

    A. In the parent class

    B. In the dependent class

    C. In the dependent collection class

    D. Elsewhere in the application

23. A class's `Terminate` event will always run

    A. When the class's object variable goes out of scope.

    B. An `END` statement executes in the server's code.

    C. After all forms have been unloaded.

    D. After all forms have become invisible.

24. A collection's built-in `Item` method

    A. Has an integer parameter that indexes the item's position in the collection.

    B. Has a `String` parameter that looks up the item's key in the collection.

    C. Has a `Variant` parameter.

    D. Has no parameter.

25. To cause a custom class event to fire,

    A. You declare the event at the point in the class module code where you want the event to fire.

    B. You use the `RaiseEvent` method on the `Object` variable you create from the class.

    C. You perform some action in the class module's code that will cause the event to fire, followed by the `DoEvents` statement.

    D. You use the `RaiseEvent` statement in the class module's code.

26. How do you implement callback functionality from a class to its calling code in a standard executable?

    A. Provide a custom class event with at least one `Boolean By Val` parameter.

    B. Provide a custom class event as a function with a Boolean return value.

    C. Provide a custom class event with at least one `By Reference` parameter.

    D. There is no way to provide callback functionality from a class in a standard executable.

APPLY YOUR KNOWLEDGE

27. When does an event procedure for a class's custom event appear in the code window of a VB application?

    A. You must manually declare the event procedure.

    B. After you write a declaration of the event in the class.

    C. After you use `WithEvents` to declare an object variable of the class.

    D. After you write code in the class to raise the event.

28. A COM component

    A. Can be either in-process or out-of-process if it is an EXE.

    B. Can be either in-process or out-of-process if it is a DLL.

    C. Can have its class `Instancing` property be `SingleUse` if it is a DLL.

    D. Can have its class `Instancing` property be `MultiUse` if it is an EXE.

29. A COM component project's Name option as found on the Project tab of the Options dialog box

    A. Must be the same as the project's filename.

    B. Will be the `objecttype` component when a controller instantiates `servername.objecttype` in the server.

    C. Will be the `servername` component when a controller instantiates `servername.objecttype` in the server.

    D. Must be the same as the main class name in the server.

30. The `PublicNotCreatable` `Instancing` property setting

    A. Makes the class `SingleUse` by default

    B. Makes the class `MultiUse` by default.

    C. Makes the class invisible to clients, but visible throughout the server application.

    D. Means that clients can see the class, but must use other classes in the component to access it.

31. It is true that

    A. You can't specify Unattended execution for an out-of-process server.

    B. You can't specify Unattended execution for an in-process server.

    C. You can't specify number of threads for an out-of-process server.

    D. You can't specify number of threads for an in-process server.

32. An `Interface`

    A. Usually contains code in its procedures, which implementing classes can override.

    B. Must reside in a separate project from client and server.

    C. Must be defined within the class that will implement it.

    D. May be referenced in other classes with the `Implements` keyword.

33. A callback object (Pick two)

    A. Is implemented in the server.

    B. Is manipulated in the server.

APPLY YOUR KNOWLEDGE

C. Is defined in the server.

D. Is instantiated in the client.

34. A callback object

A. Is used by the server to raise an event in the client.

B. Is used by the client to raise an event in the server.

C. Is passed from the client to the server.

D. Is passed from the server to the client.

35. To register an out-of-process COM server, you can (Pick two)

A. Run the REGSVR32 utility against it.

B. Run it standalone.

C. Compile it with `File Make OLE DLL`.

D. Run it with the `/REGSERVER` option.

36. A `GlobalSingleUse` or `GlobalMultiUse` class

A. Will be available throughout a server application, but not to clients.

B. Will be available to clients using the `Implements` keyword.

C. Will be available to clients without the need for object syntax.

D. Will be available throughout a server application, as well as to clients, as long as you use object syntax.

37. The `Friend` keyword

A. Makes a class available throughout a component project, but not in clients.

B. Makes a class available as a `Global` class in clients, but not in the component project.

C. Makes a member available throughout a component project, but not in clients.

D. Makes a member available globally in clients, but not in the component project.

38. You need to implement rules that your business uses to format, display, and enter currency amounts.

Your application currently runs with a SQL Server database on a network server with individual users connected through PC workstations.

What is the best way to implement these rules?

A. A component of a larger executable running on each user's workstation

B. Triggers and stored procedures in the database

C. An ActiveX control

D. An Active document downloaded from the corporate intranet

E. An ActiveX DLL component residing on a network server

39. You need to implement several business rules in your enterprise system, such as maximum credit limits for customers and employee vacation time policies.

Your application currently runs with a SQL Server database on a network server with individual users connected through PC workstations.

What is the best way to implement these business rules?

A. A component of a larger executable running on each user's workstation

B. Triggers and stored procedures in the database

C. An ActiveX control

APPLY YOUR KNOWLEDGE

D. An Active document downloaded from the corporate intranet

E. An ActiveX component residing on a network server

40. The `IDispatch` interface supports the following members (Pick all that apply)?

    A. `AddRef`.

    B. `GetTypeInfoCount`.

    C. `Invoke`

    D. `QueryInterface`.

41. The following code will cause vtable binding (Pick two)

    A.
    ```
    Dim strObj As Object
    Set strObj = CreateObject
    ➥("MyServer.MyClass")
    ```

    B.
    ```
    Dim strObj As New MyServer.MyClass
    ```

    C.
    ```
    Dim strObj As MyServer.MyClass
    Set strObj = New MyServer.MyClass
    ```

    D.
    ```
    Dim strObj As New Object
    Set strObj = New MyServer.MyClass
    ```

Answers to Review Questions

1. Only define a `Property Get` procedure. This is useful when the property is not supposed to be set by the object's client. See "Implementing Properties with Property Procedures."

2. Creating a property by using a `Property` procedure pair (`Property Get`/`Let` or `Get`/`Set`) allows for error checking, and ensures that the value passed is valid. It also gives you more control because you can run other procedures or fire events. See "Implementing Properties with Property Procedures."

3. Create a new `Public` subroutine or function in a class module. See "Implementing Custom Methods in Class Modules."

4. Each client gets its own copy of the ActiveX EXE program. Each copy of the ActiveX EXE program uses up memory as the EXE is loaded. See "The `Instancing` Property of COM Component Classes."

5. Declare a new variable by using the `WithEvents` keyword. For example, `Private WithEvents theTask as SomeTask`. See "Implementing Custom Events in Class Modules."

6. Nothing. Every class is already usable as an interface. See "Using Interfaces to Implement Polymorphism."

7. Automation components can be implemented as either executables or dynamic link libraries (DLLs). See "Overview of COM Component Programming."

8. One of the easiest methods of the increasing speed is to implement the component as a dynamic link library (DLL). In addition, you can use early binding on the client side to significantly improve function invocation performance. See "Overview of COM Component Programming."

9. The `WithEvents` keyword is used to define an object variable that supports events. An example of defining a variable that supports events is shown as follows:

   ```
   Private WithEvents m_obj as TextBox
   ```

   See "Declaring `WithEvents`."

APPLY YOUR KNOWLEDGE

10. Apartment-model threading is the threading model for Visual Basic applications. See "Managing Threads in a COM Component."

11. When you create a class whose `Instancing` property is set to `MultiUse`, client applications must declare instances of that class with the syntax:

    ```
    Dim InstanceName As Servername.Classname
    ```

    When you create a class whose `Instancing` property is `GlobalMultiUse`, client applications can declare instances of the class with the syntax:

    ```
    Dim InstanceName As ClassName
    ```

    See "Using `GlobalSingleUse` to Avoid Explicit Object Creation" and "Using `GlobalMultiUse` Instancing to Avoid Explicit Object Creation."

12. You can publish either a component's source code or the compiled component itself in Visual Component Manager. See "Managing Components with Visual Component Manager."

13. You can implement error codes from a COM component either by setting a method's return value or by raising an error to the client. See "Handling Errors in the Server and the Client."

14. REGSVR32.EXE is the name of the utility used to register COM components on a machine's system Registry. See "Registering/Unregistering an In-Process Component."

15. You can check `App.NonModalAllowed` in an ActiveX DLL component's code to see whether the client supports nonmodal forms. See "Managing Forms in an In-Process Server Component."

16. "In-process" refers to ActiveX DLLs, and "out-of-process" refers to ActiveX EXEs. See "Comparing In-Process and Out-of-Process Server Components."

Answers to Exam Questions

1. **B, C**. Both a `Property Get` and a `Property Let` procedure should be written. This will allow the property to be written to and read. A `Property Set` procedure is not necessary because the variable is not an object variable. For more information, see the section titled "Implementing Properties with Property Procedures."

2. **A, B, D**. Using a `Public` variable in the General Declarations section of a class module will define a new property for that class that is accessible from other code within the class and from all other applications that instantiate the class object. For more information, see the section titled "Implementing Properties as Public Variables."

3. **D**. An ActiveX EXE project may be defined as `SingleUse`, `MultiUse`, `GlobalSingleUse`, or `GlobalMultiUse`. For more information, see the section titled "The Instancing Property of COM Component Classes."

4. **C**. The default scope for a property is `Public`. For more information, see the section titled "Using `Public`, `Private`, and `Friend`."

5. **A, B, E**. The `Instancing` property of an in-process server class may be set to `GlobalMultiUse`, `MultiUse`, `PublicNotCreatable`, or `Private`. `SingleUse` and `GlobalSingleUse` can only be set for out-of-process servers. For more information, see the section titled "The Instancing Property of COM Component Classes."

6. **B**. Although the variable has been declared, it has not been initialized. This can be done by setting the variable to a new instance of the class. For more information, see the section titled "Overview of COM Component Programming."

APPLY YOUR KNOWLEDGE

7. **C**. Creating a new copy of the class before the class is initialized would lead to an infinite loop because each new class attempts to create a new copy until the computer runs out of memory or resources. For more information, see the section titled "Built-In Events of Class Modules."

8. **D**. Returning a value from a `Property Get` procedure is the same as returning a value from a function. Set the name of the procedure to the return value in the body of the procedure. For more information, see the section titled "Implementing Properties with Property Procedures."

9. **A**. Each copy of a `SingleUse` ActiveX EXE program can provide one object. Creating a second object would cause another copy of the EXE to be loaded in memory. For more information, see the section titled "Using `SingleUse` Instancing for Separate Instances of Every Object."

10. **D**. You need only to write code to react to the events you want to deal with. For more information, see the section titled "Implementing Custom Events in Class Modules."

11. **B**. You must write code for all the methods and properties of an interface. Not doing so will lead to a compile-time error. For more information, see the section titled "Using Interfaces to Implement Polymorphism."

12. **D**. You must set the Unattended Execution option. Any existing user interface messages will be logged according to the application's logging options and the operating system. In VB5, answer C would have been correct, because a server could not have any forms and be marked for Unattended Execution. In VB6, however, this is possible. For more information, see the section titled "Managing Threads in a COM Component."

13. **B**. The Thread per Object option initiates a thread for each new `SingleUse` class that a client instantiates. Note that Thread per Object is not available for in-process servers (ActiveX DLLs). For more information, see the section titled "Managing Threads in COM Components."

14. **D**. The `Workspace` class must be defined as `PublicNotCreatable` and is passed to the client via a function provided in the application. For more information, see the section titled "Using `PublicNotCreatable` Instancing for Dependent Classes."

15. **E**. To create a read-only property, the `Property Get` procedure must be defined and the `Property Let` and `Property Set` procedures must be omitted so that the client has no way of assigning a property value. For more information, see the section titled "Implementing Properties with Property Procedures."

16. **D**. Automation Servers can be implemented as executables or DLLs. For more information, see the section titled "Comparing In-Process and Out-of-Process Server Components."

17. **B**. When a client uses an Automation Server implemented as a DLL, the server runs in the same address space as the client. The invocation overhead is much smaller than calling an executable. For more information, see the section titled "Comparing In-Process and Out-of-Process Server Components."

18. **B**. The `WithEvents` keyword is used to define a variable that supports events. For more information, see the section titled "Implementing Custom Events in Class Modules."

APPLY YOUR KNOWLEDGE

19. **C**. The `Implements` keyword is used to specify that a specific module will provide an implementation of a specific interface. For more information, see the section titled "Creating the Interface Class."

20. **B**. When implementing custom properties with `Get/Let/Set` procedures, you store the value of the property in a `Private` variable of the class. For more information, see the section titled "Implementing Properties with Property Procedures."

21. **B**. You implement a collection's built-in features by writing wrapper procedures in the `Collection` class. For more information, see the section titled "Implementing Built-In Collection Features in the `Dependent Collection` Class."

22. **C**. When implementing a collection in an application, you would put the statement to declare the collection in the dependent collection class. For more information, see the section titled "Setting Up the Dependent Collection Class."

23. **A**. A class's `Terminate` event will always run when all object variables referring to the class go out of scope. The `END` statement ends the component abruptly without any opportunity to run events. The other scenarios cannot always guarantee that the class will terminate. For more information, see the section titled "Built-In Events of Class Modules."

24. **C**. A collection's built-in `Item` method has a single `Variant` parameter. This parameter can be used either as a traditional Integer-type index number or as a String-type unique key value to identify the specific item. For more information, see the section titled "The Collection's `Item` Method and `Class Wrapper` Method."

25. **D**. To cause a custom class event to fire, you use the `RaiseEvent` statement in the class module's code. For more information, see the section titled "Raising the Event and Implementing Callbacks in a Class Object."

26. **C**. To implement callback functionality from a class to its calling code in a standard executable, provide a custom class event with at least one `By Reference` parameter. For more information, see the section titled "Raising the Event and Implementing Callbacks in a Class Object."

27. **C**. An event procedure for a class's custom event appears in a VB application's code window after you use `WithEvents` to declare an object variable of the class. For more information, see the section titled "Handling a Class Event."

28. **D**. A COM component can have its `Instancing` property set to `MultiUse` if it is an EXE. ActiveX EXEs cannot be in-process. ActiveX DLLs cannot be out-of-process. ActiveX DLLs cannot be `SingleUse`. For more information, see the sections titled "Comparing In-Process and Out-of-Process Server Components" and "The `Instancing` Property of COM Component Classes."

29. **C**. An ActiveX component project's Name option as found on the Project tab of the Options dialog box will be the `servername` component when a controller instantiates `servername.objecttype` in the server. For more information, see the section titled "Steps to Create a COM Component."

30. **D**. The `PublicNotCreatable` instancing property setting means that clients can see the class, but that those clients must use other classes in the component to access it. For more information, see the section titled "The Instancing Property of COM Component Classes."

APPLY YOUR KNOWLEDGE

31. **D.** You can't specify the number of threads for an in-process server. For more information, see the section titled "Managing Threads in COM Components."

32. **D**. An `Interface` may be referenced in other classes with the `Implements` keyword. For more information, see the section titled "Using Interfaces to Implement Polymorphism."

33. **B, D**. A callback object is manipulated in the server (by calling a `Notify` method), but is instantiated in the client before being passed to the server. For more information, see the section titled "Providing Asynchronous Callbacks."

34. **C**. A callback object is passed from the client to the server. For more information, see the section titled "Providing Asynchronous Callbacks."

35. **B, D**. To register an out-of-process COM server (an ActiveX EXE), you can run it standalone or run it with the `/REGSERVER` option. For more information, see the section titled "Registering and Unregistering a COM Component."

36. **C**. A `GlobalSingleUse` or `GlobalMultiUse` class will be available to clients without the need for object syntax. For more information, see the section titled "The Instancing Property of COM Component Classes."

37. **C**. The `Friend` keyword makes a member available throughout a component project, but not in clients. For more information, see the section titled "Using `Public`, `Private`, and `Friend`."

38. **C**. An ActiveX control would be the best way to implement rules for data entry, because it naturally provides a user interface. For more information, see the section titled "Choosing the Right COM Component Type."

39. **E**. An ActiveX component residing on a network server would be the best implementation for business rules in general. For more information, see the section titled "Implementing Business Rules with COM Components."

40. **C, D**. The `IDispatch` interface supports the `Invoke` and `QueryInterface` methods. The other methods listed belong to `IUnknown`. For more information, see the section titled "Under-the-Hood Information About COM Components."

41. **B, C**. You will cause vtable binding by correctly using the `New` keyword in code. Answers A and D are incorrect because A does not use the `New` keyword and uses a generic `Object` variable type, and D uses the `New` keyword incorrectly with the `Object` variable type. For more information, see the section titled "Under-the-Hood Information About COM Components."

OBJECTIVE

This chapter helps you prepare for the exam by covering the following objective and its subobjectives:

Create ActiveX controls (70-175 and 70-176).

- **Create an ActiveX control that exposes properties.**
- **Use control events to save and load persistent properties.**
- **Test and debug an ActiveX control.**
- **Create and enable property pages for an ActiveX control.**
- **Enable the data-binding capabilities of an ActiveX control.**
- **Create an ActiveX control that is a data source.**

- Just as you can distribute standalone compiled ActiveX components in the form of DLL or EXE files (see Chapter 12, "Creating a COM Component that Implements Business Rules or Logic"), you can also distribute compiled ActiveX controls in separate OCX files to other programmers and end users.
- You and other programmers can then use the compiled ActiveX control in new programming projects just as you would use any custom 32-bit ActiveX control from Microsoft or third-party vendors.
- To create an OCX file implementing one or more custom controls, you create a special ActiveX control project in its own file (extension .CTL). This file will contain the ActiveX components you wish to distribute.
- In the rest of this chapter, we discuss what you must know to create an ActiveX control, as specified in the exam objective.

CHAPTER 13

Creating ActiveX Controls

OBJECTIVE

- As you'll notice from the list of subobjectives, there is quite a bit of preoccupation with the management of an ActiveX control's properties. This is because you must program for different phases in the control's life cycle: design time (when another programmer is using your control as a component) and runtime (when the user runs an application with your control in it).
- It is up to you, the creator of the control, to provide appropriate property persistence so that properties of your control "remember" their assigned values between the various phases of the control's life cycle.
- As you'll recall from Chapter 8, many standard VB controls can be bound to data by using their `DataSource` and `DataField` properties. We'll see how to implement this same behavior in your own ActiveX control. We'll also see how you can create an ActiveX control that acts as a `DataSource` just like Microsoft's ADO Data Control.
- Finally, as the objective states, we'll also discuss how to create your own property page for an ActiveX control or for a property on the control.

OUTLINE

Overview of ActiveX Control Concepts 612

ActiveX Controls as ActiveX Components 612

Creating ActiveX Controls from Constituent Controls 613

Creating User-drawn ActiveX Controls 614

The Lifetime of an ActiveX Control 614

Control Authors and Developers 615

Special Considerations for ActiveX Control Development 615

Steps to Creating an ActiveX Control that Exposes Properties 616

The UserControl Object 616

Implementing User-Drawn Graphic Features 623

Implementing Custom Methods 624

Implementing Custom Events 625

Implementing Custom Properties 627

Implementing Property Persistence 631

Creating Data-aware ActiveX controls 645

Enabling the Data-Binding Capabilities of an ActiveX Control 645

Creating an ActiveX Control that is a Data Source 647

OUTLINE

Create and Enable Property Pages for ActiveX Controls **652**

Creating the PropertyPage Object's Visual Interface 653

Determining which Controls are Selected for Editing with the `SelectedControls` Collection 654

Using the `SelectionChanged` Event to Detect When the Developer Begins to Edit Properties 655

Flagging Property `Changes` with the `Changed` Property 656

Saving Property `Changes` with the `ApplyChanges` Event 657

Connecting a Custom Control to a Property Page 658

Connecting a Single Complex Property to a Property Page 658

Detecting which Complex Property is being Edited with the `EditProperty` Event 660

Connecting a Property to a Standard VB Property Page 661

Testing and Debugging your ActiveX Control **661**

Testing Your ActiveX Control with Existing Container Applications 662

Testing and Debugging your ActiveX Control in a Test Project 663

What to Look for When Testing Your Activex Control 666

Chapter Summary **669**

STUDY STRATEGIES

- Create a simple ActiveX control with properties, methods, and events.
- Focus on implementing several properties in the control because the exam will emphasize properties more than methods and events.
- Understand the function and concept of the `PropertyBag` object.
- Learn how to persist custom properties by programming and experimenting with the `UserControl`'s `Initialize`, `InitProperties`, `ReadProperties`, and `WriteProperties` events, taking care to use the `ReadProperty` and `WriteProperty` methods of the `PropertyBag`.
- Know the implications of initializing properties in the `Initialize` event versus initializing them in the `InitProperties` event.
- Make sure you are comfortable with running and testing an ActiveX control project in the VB IDE. If you are familiar with ActiveX controls from VB5, you should note that there are new ways to test an ActiveX control from the IDE.
- Create a Property Page and connect it to the properties of an ActiveX control.
- Implement properties in the ActiveX control that allow you to bind the control to a data source.
- Create an ActiveX control that acts as a data source. Besides the simple example of Exercise 13.6, you can look at a more sophisticated example in the sample files that install with VB6. The VB project is named `MyDataControl.vbp`, and you can find it in the folder for a project group named AXData under the directory where your VB6 examples files were installed.

INTRODUCTION

This chapter discusses a very important part of Microsoft's strategy for VB development—the ability to create ActiveX controls. Since Version 5 of VB, the VB programmer is not limited to using standard VB controls provided by Microsoft or custom controls provided by Microsoft and third parties.

Now VB developers can create their own ActiveX controls, either as an integral part of a larger VB project or as a standalone reusable component in an .OCX file. The information in this chapter provides a basic acquaintance of VB6 ActiveX control programming while covering the ActiveX control objectives of the VB6 certification exams.

OVERVIEW OF ACTIVEX CONTROL CONCEPTS

In this section, we discuss concepts behind ActiveX control programming. In the rest of the chapter, we discuss what you need to do to create and distribute an ActiveX control.

ActiveX Controls as ActiveX Components

ActiveX controls are a special type of ActiveX component. Therefore, most of the ActiveX concepts discussed in Chapter 14, "Creating an Active Document," apply directly to the subject of ActiveX controls. In particular, an ActiveX control has the following notable features of any ActiveX component class:

- Programmer-definable properties
- Programmer-definable methods
- Programmer-definable events

Moreover, you can implement these custom-definable members of the ActiveX control class in the same way that you implement customized members of other ActiveX components. Of course there are special considerations for ActiveX control members (especially properties), and we discuss the specifics of ActiveX control members in the rest of this chapter.

Standalone (OCX) ActiveX Controls

Just as you can distribute stand-alone compiled ActiveX components in the form of DLL or EXE files, you can also distribute compiled ActiveX controls in separate OCX files to other programmers and end users. An OCX is another type of DLL that lives in a host application.

To create an OCX file implementing one or more custom controls, you create a special ActiveX control project containing the ActiveX components you wish to distribute. Each ActiveX component will be implemented with source code in its own file (extension .CTL). When you follow the steps to create an ActiveX control as described in the rest of this chapter, you then can compile the project into an OCX.

NOTE

Licensing Considerations for ActiveX Control Distribution If you plan to distribute your OCX file commercially to end users or other programmers or if you plan to distribute the file to the public at large over the Internet, you must also take into account the special licensing and distribution considerations for ActiveX controls.

ActiveX Controls in Other Projects

An ActiveX control can also form part of another VB project. In that case, you would include the control's CTL file as part of a Standard EXE or ActiveX EXE or DLL project.

You might choose to implement an ActiveX control as part of another project when the control's function is so specialized that it would never be used outside its host application.

You might also keep an ActiveX control inside other projects to prevent other programmers from accessing it independently. You can also accomplish the goal of keeping your control proprietary, however, by using a licensing scheme described in "Licensing and Distributing Your ActiveX Control."

Creating ActiveX Controls from Constituent Controls

The end-user interface for an ActiveX control often consists of an assemblage of one or more constituent controls. Such an ActiveX control is sometimes called a *composite control*.

Constituent controls are controls that already exist in the programming environment, and they can be intrinsic Windows controls or custom controls from Microsoft or other vendors. You can then piece together a new custom control by taking parts of the features of the constituent controls and adding your own custom functionality as well.

NOTE

Licensing Considerations for Composite Controls You must make sure that you have proper licensing rights before distributing a constituent control as part of your own ActiveX control.

A special case of the composite control would be the control which contains only one constituent and whose purpose is to enhance the functionality of the constituent by implementing different methods, properties, and events.

Creating User-Drawn ActiveX Controls

Besides building an ActiveX control out of constituent controls, you can also create a custom control whose appearance is completely determined by you. You, the control author, are responsible for providing and maintaining the control's appearance through graphics methods or by other means.

You can refresh your control's appearance in the `UserControl` object's `Paint` event procedure discussed in the section "Implementing User-Drawn Graphic Features."

Of course, you can combine the techniques of user-drawn and composite controls to create a custom control. It can use constituent controls, but also has part or all of its appearance controlled by the control author.

The Lifetime of an ActiveX Control

Managing a custom ActiveX control's events and behavior is a bit more complex than for most other objects, such as standard controls and Forms or other types of ActiveX components.

This complexity is due to the fact that you must keep two levels of behavior in mind for your control rather than just knowing the usual runtime behavior. In other words, you must also manage the behavior and features of an ActiveX control when another programmer uses it at design time in an application.

Most of the increased complexity in a custom ActiveX control has to do with the management of the contents of its properties as the control moves from abstract definition to being an instantiated object in a VB project and from there to life in a running VB application.

Control Authors and Developers

In order for your ActiveX control to provide its services to end users, some intermediate programmer must put the control into his or her application. As the author of an ActiveX control, therefore, you must specify design-time features of your control as well as runtime features.

The ultimate user of the runtime features is that archetypal, shadowy being we programmers have come to know, love, and fear as the end user.

The user of the design-time features will be another programmer. In this chapter we will call you, the programmer creating the ActiveX control, the ActiveX control *author*. In this chapter, we will refer to the programmer using the ActiveX control in an application as the *developer*.

WARNING

Who Is the "User" of an ActiveX Control? Much of the documentation on ActiveX custom controls employs the word "user" loosely, sometimes meaning the end user of the application where your control is sited and, at other times, meaning the developer who is programming with your control.

Special Considerations for ActiveX Control Development

Your custom control will be used by other developers who will in turn employ it in their own programs for their own end users. This fact complicates custom control creation in several major areas:

- You must manage property values so that they persist from the developer's design-time changes to the runtime environment.
- You must allow for the interaction between the members of your control, its constituent controls, and both the design-time and runtime environment in which the developer has placed your control.
- You must ensure that your control cannot be used maliciously by developers.
- You must decide whether your control can be used only by licensed developers and, if so, you must implement a licensing scheme.

We discuss the first two points in the section that follows titled "Steps to Creating an ActiveX Control that Expose Properties." We discuss control security, distribution, and licensing in the sections at the end of this chapter.

Steps to Creating an ActiveX Control that Expose Properties

To implement your own ActiveX control, you must take the following basic steps:

- Put a `UserControl` container into your application. This object is implemented in its own separate file similar to a Form. The `UserControl` provides the basis for the control.
- If you decide to implement user-drawn features in your control, you must plan and implement those graphic features, putting code in the appropriate event procedures.
- Create custom methods and events in the `UserControl` as you would in a Class Module.
- Create custom properties in the `UserControl` as you would in a Class Module, remembering to make extra provision for property persistence.
- If you decide to implement constituent controls, then you must place any constituent controls that you will be using on the `UserControl`'s surface and implement their *delegated methods*, *delegated events*, and *delegated properties* as well.
- Put additional code in `UserControl` event procedures so that you initialize your control's properties appropriately and cause them to persist.

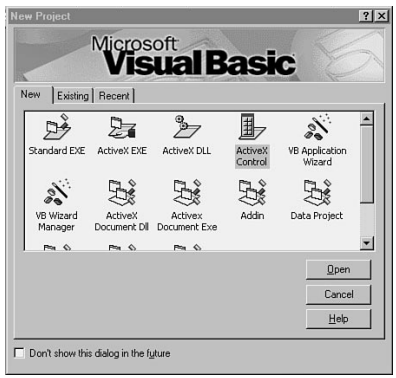
The UserControl Object

The `UserControl` object is the basis for any custom ActiveX control. It provides the container that holds all the rest of the control's features.

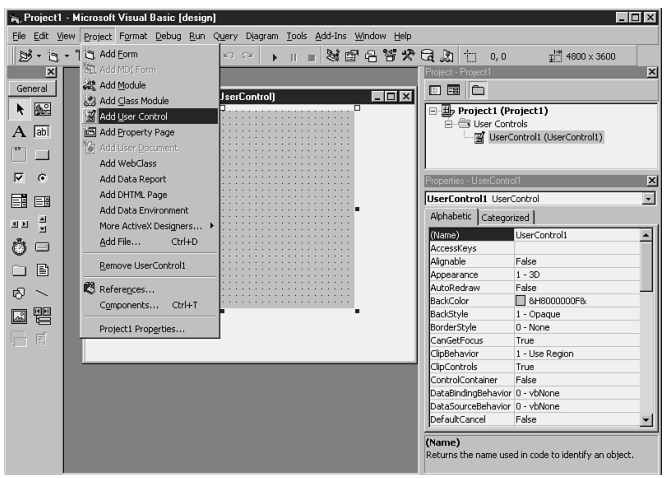
You may initiate your project's `UserControl` in one of two ways:

- Begin your project as an ActiveX Control project (see Figure 13.1).
- Add a `UserControl` object to your existing project (see Figure 13.2).

If you chose to add a `UserControl` to your project, then the project is probably not an ActiveX project to begin with. This may be exactly the situation you desire if you've chosen to implement your ActiveX control as part of another project (see "ActiveX Controls in Other Projects"). If, however, you wish to convert your project to an ActiveX control project, all you need to do is change the Project Type option on the General tab of the Project Property dialog box (see Figure 13.3).

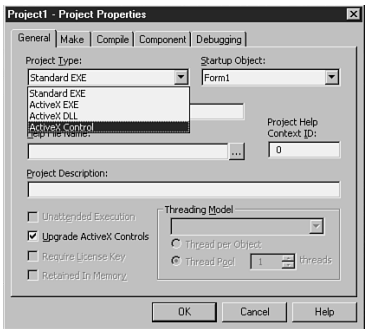


◀FIGURE 13.1
Beginning your project as an ActiveX Control project.



◀FIGURE 13.2
Adding a `UserControl` object to your project.

FIGURE 13.3
Adjusting the project type.



WARNING

Conflict Between `UserControl` Name and Project Name You can't give a `UserControl` the same name as its Project. VB will raise an error as soon as you try to do

WARNING

Referring to the `UserControl` in Code When you refer to the `UserControl` in its own code, always use the term "`UserControl`" rather than the actual name you've chosen. For example:

```
UserControl.BackColor = vbRed
```

NOTE

Referring to the `UserControl`'s Members in Code When you write code in the `UserControl`, you can refer to the `UserControl`'s own built-in members (properties and methods) by referring to the members alone without reference to the `UserControl` object. For example:

```
BackColor = vbRed
```

This is similar to the way that you can program within a Form. Note, however, that the `Me` keyword does not work in a `UserControl` (you will receive a compile-time error if you use `Me`).

Programming with a `UserControl` is a lot like programming with a form. The `UserControl` object has its own Designer Window in the VB design-time environment, just like a form does, and a `UserControl` appears as one of your project's elements in the Project Explorer. It appears under a special node labeled "User Controls." You place constituent controls on the surface of the User Control from the toolbox, just as you do when programming with a form.

The `UserControl` has its own properties, which we discuss in some of the following sections. You can access these properties at design-time just like you'd access the design-time properties of a form:

Make sure that you've selected the `UserControl` object itself (and not one of its constituent controls), and then go to the Properties Window by pressing F4 or right-clicking the mouse.

To access a `UserControl`'s event procedures, make sure that you've selected the `UserControl` itself (and not one of its constituent controls), then double-click the mouse to view `UserControl` code.

Giving Your Control a Custom Bitmap

Since your custom control will eventually end up on a developer's toolbox, it needs a bitmap to display in the developer's VB toolbox. You can let the default ActiveX control bitmap show up in the toolbox, or you can set the `UserControl`'s `ToolBoxBitMap` property to determine the custom control's appearance in a developer's toolbox.

`ToolBoxBitMap` is like the `Icon` or `Picture` property of other controls: You set it by double-clicking the `ToolBoxBitMap` property or clicking the Add button to the right of the property in the Property Window of the `UserControl`. Note, though, that unlike the `Picture` and `Icon` properties, the `ToolBoxBitMap`'s File dialog box doesn't allow you to choose icon (.ico) files (see Figure 13.4). Instead the `ToolBoxBitMap` can be a .bmp (bitmap), .gif, .dib, .pal or .jpg file. The image size should be 16×15 pixels, or else it will be scaled by the system with unpredictable (and sometimes unsightly) results.

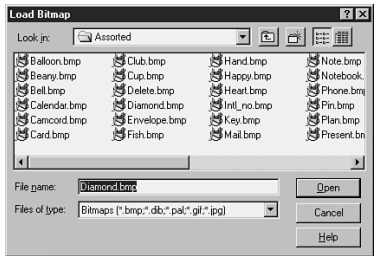


FIGURE 13.4
Choosing a picture file for the `UserControl`'s `ToolBoxBitMap`. Note that the file types allowed are different from most `Picture` and `Icon` properties' file types.

Accessing Ready-Made Control Features with the `UserControl`'s `Extender` Object

The ActiveX standard allows all container objects, such as `Forms` and `PictureBox` controls, to provide certain members (properties, events, and methods) to all controls (including custom ActiveX controls) that are placed on their surfaces. Some of these members might include, for instance, `Top`, `Left`, `Width`, and `Height` properties, an `Enabled` property, a `Move` method, and `GotFocus` and `LostFocus` events.

These ready-made members provided by your control's container object are all contained in an object known as the `Extender` that belongs to your custom control.

Your custom ActiveX control will therefore already have certain methods, properties, and events provided for it automatically through its `Extender` object. If you had just created an ActiveX control by putting a new `UserControl` object into your project and had not yet implemented any of your own custom members, you would still see quite a few default properties in the control's Properties Window when you tested it (see Figure 13.5). These properties would be the properties provided in the `Extender` object.

NOTE

Example Here Doesn't Persist the Property Don't expect the small example in this section to allow the property value to persist when you attempt to run an .exe project that tests this control. In order to see the Label's caption carried over into an instance of your control at runtime, you would also need to implement a Caption or other property for your control and persist it with calls to `ReadProperty` and `WriteProperty` in the `ReadProperties` and `WriteProperties` event procedures respectively. See the following sections for more detail on how to implement property persistence.

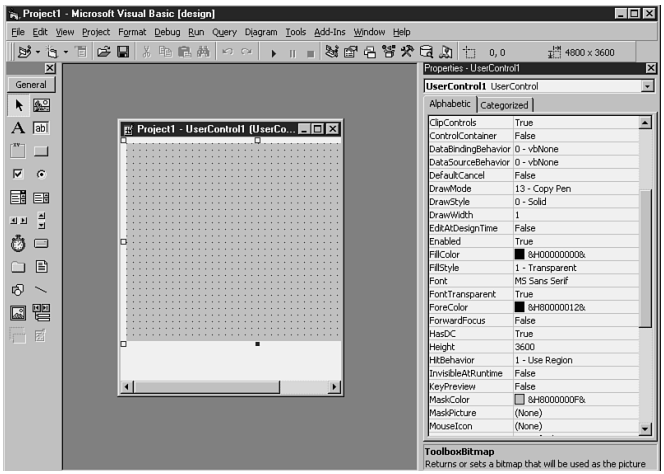


FIGURE 13.5
The custom control instance whose properties we are viewing in this test project doesn't have any custom properties, yet we see many properties provided through the `Extender` object in its Properties Window.

NOTE

Standard (More or Less) `Extender` Properties The ActiveX standard recommends (but does not require) that all containers implement the following `Extender` properties: `Name`, `Visible`, `Parent`, `Cancel`, `Default`. You may, however, confidently reference all these properties in code without fearing an error, since VB returns default values for any of the above properties not implemented by a container.

WARNING

Naming Conflicts Between `Extender` and Custom Members
If you implement a member with the same name as an `Extender` member, the `Extender` member will always override your custom member because the developer will always see the `Extender` member and not your custom member.

As control author, you should not be very concerned about the `Extender` object's members since they are intended for the developer's use.

However, you may check the values of `Extender` properties in order to find out what the developer or the design-time environment has done with your control.

For instance your control may have a Label whose Caption represents the Caption of your custom control. As you know it's customary for a VB control with a Caption property to begin life with its caption equal to the control's name. You can emulate this behavior by checking the `Name` property of the `Extender` object in your control's `InitProperties` event procedure and setting your control's Caption to the value of the `Name`, as in the following line of code:

```
lblCaption.Caption = Extender.Name
```

Getting Information About Your Control's Environment with the `Ambient` Object and the `AmbientChanged` Event

The `Ambient` object gives you information about selected properties (such as `ForeColor`, `BackColor`, and `Font`) of your custom control's container. The `AmbientChanged` event fires whenever one of these selected properties changes, thus giving your control the chance to react to changes in its container.

A common use of the `Ambient` object would be to synchronize the custom control's background color with its container's in the `AmbientChanged` event, as in Listing 13.1. Notice that the `AmbientChanged` event receives a single parameter indicating which container property has just changed. We check the parameter to see if a property we are interested in has changed and, if so, we synchronize our control with the value of the same property of the `Ambient` object.

LISTING 13.1

CHECKING TO SEE WHICH CONTAINER PROPERTY HAS CHANGED IN THE AMBIENTCHANGED EVENT PROCEDURE

```
Private Sub UserControl_AmbientChanged _
              (PropertyName As String)
    If UCase$(PropertyName) = "BACKCOLOR" Then
        BackColor = Ambient.BackColor
    End If
End Sub
```

The `LocaleID` `Ambient` property is a number representing the country or language of the developer. For example, 1033 represents U.S. English (these numbers are often referred to in hex notation in documentation).

WARNING

Don't Change `Extender` Properties in Code You should never try to change the values of `Extender` properties. You will cause runtime errors or get unpredictable results.

In particular, you may be tempted to use the `Visible` property of the `Extender` object to make your control invisible. Instead of this `Extender` property, you should use the `UserControl`'s `InvisibleAtRunTime` property.

NOTE

`Ambient` Property Names Aren't Always the Same as Underlying Property Names The names of `Ambient` properties aren't always the same as the names of the underlying container properties they reflect. For instance when the container's `ScaleMode` property changes, you'll see a change in the `Ambient.ScaleUnits` property.

The `UserMode` property (a Boolean property) is an important `Ambient` object property for the control author. It tells you whether the current control instance is in design or run mode. When `UserMode` is `True`, the control instance is in a running application. When it's `False`, the control instance is in a design-time instance of its container.

You can check UserMode in the `UserControl`'s `Paint` and `Resize` events or in a custom property's `Property` procedures, for example, to vary the behavior of your control. You can give your control one behavior if it's in a running application (`Ambient.UserMode = True`) and another when it's in the VB design environment (`Ambient.UserMode = False`). In the example of Listing 13.2, we implement a property that is writeable at design time but not at runtime by checking `Ambient.UserMode` in the `Property Let` procedure.

LISTING 13.2

THIS PROPERTY IS ONLY WRITEABLE AT DESIGN TIME

```
Property Let MyProp (sValue as String)
    If Not Ambient.UserMode Then
        m_MyProp = sValue
    End If
End Property
```

Some `Ambient` properties don't represent states of the container *per se* but rather states of your control *relative to* the container. For instance, the `DisplayName Ambient` property represents the name by which your control is known to the container (you should use `Ambient.Displayname` in error messages that you display in your control's code). Likewise the `DisplayAsDefault Ambient` property is `True` if the developer has set your control to be the `Default` control on its container (`Default = True`).

Comparison of the Ambient and Extender Objects

Since it's easy to confuse purpose and function of the `Ambient` and `Extender` objects and their respective properties, here is a head-to-head comparison of the two objects:

- The `Ambient` object gives information about your `UserControl`'s container object (such as a Form or a `Frame` or `PictureBox` control) at runtime and, in some cases, about the relation between your `UserControl` and its container.
- The `Extender` object implements properties that come "for free" with your ActiveX control: That is, properties that are part of the default arsenal of any control and that you, therefore, didn't have to implement as custom properties of the control. Such properties would include `Font`, `Top`, `Height`, and others.

Implementing User-Drawn Graphic Features

If you are implementing a control with user-drawn features instead of a control made up of constituent controls, you'll use the `UserControl`'s `Paint` event to manage your control's appearance.

The `Paint` event occurs at design time and at runtime whenever the operating system determines that your control needs to redisplay graphics. This would include such events as control siting and resizing or a control being uncovered after another window had obscured it.

You will put in the `Paint` event's procedure whatever graphics commands and methods you need to display the control's graphics. You will probably need to recompute the sizes and shapes of graphics objects to allow for resizing of the control.

In the example in Listing 13.3, we use the `Paint` event procedure to draw an ellipse with a particular color and shading style on the surface of the control after first re-computing its size and coordinates.

You could also output text with the `Print` method, call other graphics methods, and adjust persistent graphics objects such as `Images`, `PictureBoxes`, `Shapes`, and `Lines`.

LISTING 13.3

REDRAWING A UserControl'S GRAPHICS IN ITS Paint EVENT PROCEDURE

```
Private Sub UserControl_Paint()
    'Compute radius of ellipse (longest side)
    Dim lRadius As Long
    If UserControl.ScaleHeight < _
       UserControl.ScaleWidth Then
        lRadius = UserControl.ScaleWidth / 2
    Else
        lRadius = UserControl.ScaleHeight / 2
    End If

    'Save UserControl's existing fill style
    Dim lOldFillStyle As Long
    lOldFillStyle = FillStyle
    'and set it to be diagonal lines
    FillStyle = vbUpwardDiagonal

    'save UserControl's existing FillColor
      Dim lOldFillColor As Long
    'and set it to red
    lOldFillColor = FillColor
    FillColor = vbRed

    'draw an ellipse whose shape is
    'proportional to the UserControl's shape
    Circle (ScaleWidth / 2, ScaleHeight / 2), _
         lRadius, vbRed, , , _
           UserControl.ScaleHeight / _
           UserControl.ScaleWidth

    'restore previous FillStyle and Color
    'settings of the UserControl
    FillStyle = lOldFillStyle
    FillColor = lOldFillColor
End Sub
```

Implementing Custom Methods

You implement a method for your ActiveX control in pretty much the same way that you implement methods for other ActiveX components and for classes in general. As mentioned above, custom methods for an ActiveX control are implemented as Public Procedures in the `UserControl` object.

If, for example, you wanted to give your developers an `Alarm` method for the control you are developing, you would put code similar to the code of Listing 13.4 into your `UserControl`'s code.

LISTING 13.4

CODE FOR A CUSTOM CONTROL METHOD

```
Public Sub Alarm(Optional Severity)
    'If the alarm is more severe,
    'then beep more times
    If IsMissing(Severity) then Severity = 1
    Dim iCount As Integer
    For iCount = 1 To Severity
        Beep
    Next iCount
End Sub
```

NOTE

Using an `Optional Variant` Parameter Notice that our example method takes an `Optional` parameter. We've let the parameter's type default to `Variant` so that we can check for its existence with the `IsMissing` function (recall that `IsMissing` only works properly on `Optional` parameters of `Variant` type).

Assuming that your developer has created an instance of your control named "MyControl" in a project, then the developer could write a line of code such as

```
MyControl.Alarm 7
```

Implementing Custom Events

The story for custom control events is much the same as for custom control methods: You implement an event for your ActiveX control in pretty much the same way that you implement events for other ActiveX components and for classes in general. Besides the custom events that you create, you can also implement delegated events or custom events that provide wrappers for the events of an ActiveX control's constituent controls.

Declaring and Raising Events

You declare an event for a custom control just as you would declare an event in any Class module:

In the `UserControl` object's General Declarations section you put a declaration beginning with the words "Public Event" followed by the event name you wish to use and then parentheses enclosing a list of the names and types of any parameters that the event will use. For example, let's say you declared a `UserControl` event called `FoundOne` that passes a single string parameter describing the object it has found. It would look like the following:

```
Public Event FoundOne(FoundName As String)
```

You must then put code somewhere else in the `UserControl`'s code to actually fire the event with the `RaiseEvent` statement:

```
RaiseEvent FoundOne("Morsel")
```

Developers who use your control in their applications will see event procedures in their code windows for the `Found` event.

NOTE

Programming Your Own Custom `Class` Events For a more detailed discussion of programming custom events, see Chapter 12, "Creating a COM Component that Implements Business Rules or Logic."

Implementing a Default User Interface Event with the Procedure Attributes Dialog Box

Imagine that you've just placed a new form in a standard VB project. Now imagine that you double-click the form before doing anything else—quick! What do you see? Of course, anyone who's programmed for more than a few days with VB will answer, "The `Form/Load` event procedure."

The `Load` event procedure is the `Form` event that the VB editor will show you by default, all other things being equal (that is if there's been no code placed in any other event procedures). The `Load` event is the *default user interface* event—the event whose procedure will show up by default in the Code Window if no other event procedures contain code. You can probably think of the default user interface events for many other controls: `Click` for a `CommandButton`, `Change` for a `TextBox`, and so on.

You can give your custom control its own default user interface event by following these steps:

STEP BY STEP

13.1 Giving Your Custom Control Its Own Default User Interface Event

1. Make sure you're in a Code Window for the `UserControl` object.
2. Bring up the Tools, Procedure Attributes option on the VB menu.
3. In the resulting dialog box Name field, choose the name of the custom event you wish to designate as the default user interface event.
4. Click the Advanced button to bring up more choices.
5. In the Attributes section, check the box labeled User Interface Default.
6. Click OK to save your choice and exit the dialog box.

Implementing Custom Properties

Up to this point, we've seen that you can implement custom control methods and events in more or less the same way as you implement methods and events for other ActiveX object classes.

The situation for custom control properties is more complex however. This is because, as the author of a custom control, you must manage the persistence of property values as instances of a control are created and destroyed at design time.

You must do a little more work than for events and methods in order to get custom control properties to behave properly.

As we've already mentioned several times, this is because instances of your control are created and destroyed numerous times during the development life cycle. For more detail on the actual activities that cause such carnage, see the accompanying sidebar.

EVENTS IN A CONTROL'S LIFETIME THAT AFFECT PROPERTIES

During the developer's design, test, compile, and redesign cycle, instances of your control will be created, then placed or sited on their container object (a `Form`, `PictureBox`, or other object capable of holding controls), and destroyed—not just once, but many times.

The reason for all this activity is that an instance of your control must be destroyed and created whenever the control and its container move between the various possible degrees of being and non-being during the phases of development. Here is a list of the occasions when instances of a control would be created and destroyed:

- **Created**. The developer creates an instance of the control from the toolbox and sites it on its container.
- **Destroyed**. The developer closes the Design window with the control's container.
- **Created**. The developer reopens the Design window with the control's container.
- **Destroyed**. The developer begins to run the application in the VB environment, which temporarily destroys design-time instances of objects (including your control).
- **Created**. The control's container loads into memory and sites the control as the application runs.
- **Destroyed**. The control's container is destroyed as the application runs.
- **Destroyed**. The control's container is destroyed because the application stops running.
- **Created**. The application has stopped running and VB returns to the design environment thus resiting your control (provided the design-time environment currently includes a sited instance of the control).
- **Destroyed**. The developer closes the project or closes VB.
- **Created**. The developer opens the project provided that the current view of the project includes an instance of the control's container.

You may wonder why it's important for you to know about the timing of the creation and destruction of ActiveX controls. After all, we can manipulate standard controls and other objects without having to know such details.

As control author, you must be able to react to control creation, destruction, and change. It's up to the control's author to determine when and how long changes to control properties will persist. As you probably know, control properties get stored with a form's other information in the form's .frm file. While a developer works on an application in the VB design environment, the copy of the form (and all its associated information) that resides in memory gets destroyed and re-created as described in the above list. It is the responsibility of each control to save and retrieve information about its own state (including any changes to property values) during these times.

You can implement this management of your custom ActiveX control's properties by programming the event procedures of various events of the `UserControl` object and by taking certain measures whenever a property might change, as discussed in the following subsections.

In the following sections, we discuss the bare minimum you need to do in order to get custom control properties to function adequately.

We first discuss how to create a new custom property (this is the same as for other object classes in VB) then we discuss how to initialize properties in the `InitProperties` event procedure. Finally, we cover the particulars of how to make properties persistent by programming the `UserControl`'s `Property Bag` object in several key `UserControl` events.

Defining Properties with `Get`/`Let`/`Set` Procedures or `Public` Variables

Just as you do with other object classes in VB, you define custom properties for ActiveX controls either with `Public` variables or with `Property` procedures (`Property Let`/`Set` and `Property Get`).

Because it's very likely that a developer will use more than one instance of your control in an application, the use of `Public` variables is strongly discouraged. Values for the same property in different instances of the control could become confused if a `Public` variable were used to implement a property that the system accessed in more than one instance of a control.

Instead, you will always want to use `Property` procedures to implement control properties. As you'll recall from other discussions of class properties, you'll need to set up the following elements for each custom property:

NOTE

Property Procedures See the subsection of Chapter 12, "Creating a COM Component that Implements Business Rules or Logic" under the main section "Compiling a Project with Class Modules into a COM Component," to review how to use `Property` procedures including the concept of how to implement a read-only property by not supplying a `Property Let` procedure.

- A variable that is Private to the `UserControl` and that will serve as a storage place for the property's actual value. `Property Get` and `Property Let/Set` will respectively retrieve and store this value.
- A `Property Let` or `Property Set` procedure that acts like a `Sub` procedure and accepts a parameter whose value is the new value to be assigned to the property. You will store this parameter's value in the Private storage variable just mentioned in the previous point.
- A `Property Get` procedure that acts like a Function procedure and returns the value kept in the `Private` storage variable.

The code in Listing 13.5 illustrates how you would implement your own property for a custom control. Notice that we use `Property` procedures and the `Private` variable in almost exactly the same way as we do for custom properties in standard Class modules.

NOTE

Storing Property Values in Delegated Properties Instead of storing the custom property's value in a `Private` variable, you might choose to store its value in a property of one of the `UserControl` object's constituent controls. You would then be implementing a *delegated property*, as discussed in "Implementing Delegated Properties."

The only difference in Listing 13.5 from standard class modules' `Property` procedures is the extra line in the `Property Let` procedure that calls the `PropertyChanged` method. This call notifies VB that a property has been changed and ensures that the `WriteProperties` event will fire at the appropriate moment. For more detail, see the following section, "Calling the `PropertyChanged` Method to Trigger `WriteProperties`".

You must take into account a number of extra considerations to make your custom control properties work properly. In the following sections, we discuss what you need to know in order to create fully functioning custom properties for your controls.

NOTE

Automatically Implementing Color Dialog Boxes for Properties of Type `OLE_COLOR`

If you create a custom property and declare its type to be `OLE_COLOR`, then VB will automatically show the `Color` property dialog box to the developer in the Properties Page of an instance of your control. Remember that you must then consistently refer to its type as `OLE_COLOR` throughout the control's code (in its `Private` variable and `Property` procedure declarations).

LISTING 13.5

IMPLEMENTING A PROPERTY FOR A CUSTOM CONTROL

```
[in the general declarations section
 of the UserControl]
Option Explicit
Private mOverallColor As Long
Property Get OverallColor() As Long
    OverallColor = mOverallColor
End Property
Property Let OverallColor(lColor As Long)
    mOverallColor = lColor
    'informs the system that a
    'property has changed:
    PropertyChanged
End Property
```

Implementing Property Persistence

You can keep information about properties whose values you want to make persistent in the `Property Bag` object of the `UserControl`. The `Property Bag` object has two methods you will call: `ReadProperty` (to retrieve a property's value from the `Property Bag`) and `WriteProperty` (to store a property's value in the `Property Bag`).

In addition, you can maintain properties as you would in any ActiveX Class module: by writing and reading the `Private` variables that are used to store the property values between calls to `Property` procedures.

On the other hand, if you've chosen to implement properties with `Public` variables, you can also manipulate and monitor these properties by reading and writing the `Public` variables.

You can use these techniques to initialize and maintain persistent property values in your `UserControl`.

Here is a list of the events you need to know about to maintain properties. For each event listed, we tell when the event occurs and give a brief note about how you would use the event's procedure to maintain persistent property values. We provide more detailed information on the `Property Bag` and on these events in the following sections.

- The `InitProperties` event happens only once in the lifetime of a custom control's instance: when the developer actually creates a new instance of the control by clicking on the control's toolbox icon and placing the new instance in a container. Use the `InitProperties` event to set default initial values for your custom properties and for properties provided by the `Extender` object.
- The `Initialize` event is essentially the same as the `Initialize` event of any Class module. It happens as the instance of the `UserControl` is about to be created.
- The `ReadProperties` event happens whenever an instance of the control is created. If this is the first time the control has been created (i.e. the developer has used the toolbox to place a copy of the control on the form), then `ReadProperties` happens after `InitProperties`. You can put code in the `ReadProperties` event procedure to retrieve saved persistent property values.

NOTE

Using Consituent Control Properties to Store `UserControl` Property Values
If you're implementing delegated properties, you may map properties of constituent controls instead of `Private` variables to store the values of properties.

- The `WriteProperties` event happens whenever an instance of the control is destroyed and at least one property has been changed (you can notify the system that a property has been changed by calling the `PropertyChange` method of the `UserControl`). You can put code in the `WriteProperties` event procedure to save property values that you would like to persist.
- The `Terminate` event is essentially the same as the `Terminate` event of any Class module. It happens as the instance of the `UserControl` is about to be destroyed.

Using the `InitProperties` Event to Set Default Starting Property Values

The `InitProperties` event fires just once in the lifetime of a control: When the developer first sites it on its container from the toolbox.

The `InitProperties` event procedure is therefore the perfect place to set initial default values for your control's properties since you'll want these default values to show up when the developer first sites the control on its container. However, you won't want the default values to override any later changes the developer makes after siting the control.

You will mainly use `InitProperties` to initiate default values for your custom control's properties. If you've implemented a property as a `Public` variable, you can simply set the variable name to the desired default value. On the other hand, if you're using `Property` procedures to implement a property, you can set its default value by writing to the `Private` memory variable or other holder (such as a constituent control property) that maintains the property's value.

You can set these default properties in a number of ways, depending on the type of property you want to set and on how you've chosen to implement the property:

- Assign a literal value to the `Private` storage variable implementing the custom property.
- Assign a default constant value (as discussed in the previous section) to the `Private` storage variable.
- Derive a property's initial value from some `Extender` property.
- Derive a property's initial value from some `Ambient` property.

The `InitProperties` event procedure shown in Listing 13.6 gives examples of the techniques listed above.

The code in the following listing assumes that `Slider` and `TextBox` controls exist in the project with respective names `Slider1` and `txtDate`. It also pre-supposes `Private` variables and constants that implement property values and their defaults, as shown in the General Declarations section of the listing.

LISTING 13.6

Various Techniques for Initializing Properties to Default Values in the InitProperties Event Procedure

```
Option Explicit
Private Const defCelsius = 30
Private m_TemperatureDate As Date
Private m_Celsius As Single
Private m_Caption as String

Private Sub UserControl_InitProperties()
    'Set temperaturedate to a value
    'determined right here
    m_TemperatureDate = DateSerial(1997, 1, 1)
    'Set Celsius to a default
    'constant value
    m_Celsius = defCELSIUS
    'Compute Fahrenheit from that
    RecalcFahrenheitFromCelsius m_Celsius
    'Set the Background color to
    'match the container's background
    BackColor = Ambient.BackColor
    'Set the caption to be the
    'same as the object's initial name
    'm_Caption = Extender.Name
    'and set Slider to reflect temperature
    Slider1.Value = m_Celsius
    'and textbox to match date
    txtDate.Text = m_TemperatureDate
End Sub
```

Note in the listing that variable names storing property values do not follow the usual Hungarian notation naming convention (a two-four letter lowercase prefix indicating variable type). Instead, variable names that refer to the intermediate storage of the values of properties implemented with `Property Get`/`Let`/`Set` are prefixed with "m_"—a Microsoft documentation standard for Class modules. Microsoft's Control Wizard generates such variable names, automatically deriving them from the names of properties that you specify in the wizard.

The `InitProperties` Event Versus the `Initialize` Event

You may wonder why we need the `InitProperties` event since the `UserControl` property already has an `Initialize` event in common with Class modules and other objects, such as forms.

The reason that we need `InitProperties` to initialize default values is that the `Initialize` event happens too often, that is, every time an instance or your control "wakes up." You only want the default property values to be assigned when the developer first sites a new copy of your control on a container. After that, you want the developer to be able to define persistent property values.

WARNING

Don't Use the `Initialize` Event to Set an ActiveX Control's Default Property Values If you put code to initialize properties to their default values in your custom ActiveX control's `Initialize` event instead of in the `InitProperties` event, then you will have some very frustrated developers on your hands. Your default values will override the values the developer has assigned at design time every time the developer runs an application using your control.

Using the `Property Bag` to Store Property Values

The `Property Bag` is a persistent `UserControl` object containing the values of your control's custom, extender, and delegated properties.

In fact, the `Property Bag` is so persistent that it doesn't get destroyed with the instances of the `UserControl`. This means you can store property values in the `Property Bag` just before an instance of the `UserControl` is destroyed and then retrieve the stored values when a new instance of the `UserControl` "wakes up" in another part of the development life cycle.

The `Property Bag` has two methods to store and retrieve values respectively:

- The `WriteProperty` method
- The `ReadProperty` method

You must know how to manipulate the `Property Bag` in the following situations that we discuss in the sections immediately following this one:

- You store property values into the `PropertyBag` by calling its `WriteProperty` method in the `WriteProperties` event procedure.
- You retrieve property values from the `PropertyBag` by calling its `ReadProperty` method in the `ReadProperties` event procedure.
- You ensure that the `WriteProperties` event will fire by calling the `PropertyChanged` method. You'll usually do this in the Property Let procedures of your custom properties or at other appropriate places in your code where the storage value of a property changes.

Using `ReadProperties` and `WriteProperties` Events and Corresponding Methods to Maintain Persistent Properties

The operating environment fires a `UserControl`'s `ReadProperties` and `WriteProperties` events whenever it thinks that the instantiated object's properties need to be re-initialized (`ReadProperties` event fires) or stored for safekeeping (`WriteProperties` event fires).

This arrangement makes it much easier for you, the control author, to manage these properties since you don't have to think about all the possible occasions when property values might need reading or writing. You simply need to put code for reading and writing property values in two centralized places: the `ReadProperties` and `WriteProperties` event procedures.

Both the `ReadProperties` and `WriteProperties` event procedures receive a single parameter named `PropBag`. This `PropBag` parameter obviously represents the `Property Bag` object that holds the `UserControl`'s property values.

The `PropertyBag` object represented by the `PropBag` parameter has one method for reading properties (`ReadProperty`) and another for writing properties (`WriteProperty`).

Usually, the only code you need to write in the `ReadProperties` event procedure will be a series of calls to the `ReadProperty` method so you can retrieve persistent values for individual properties.

Conversely, the only code you usually need to write in the `WriteProperties` event procedure will be a series of calls to the `WriteProperty` method so you can store persistent values of individual properties.

Storing Persistent Property Values with the `WriteProperties` Event and the `WriteProperty` Method

Visual Basic fires the `UserControl`'s `WriteProperties` event just before it fires the `UserControl`'s `Terminate` event provided that at least one property value has changed. In other words, the `WriteProperties` event fires whenever the current instance of the control is about to be destroyed and any property values that you want to persist have changed and, therefore, need to be saved.

As its name implies, you use the `WriteProperties` event procedure to save persistent property values. The specific mechanism you use to save property values is to call the `WriteProperty` method of the `Property Bag` for each property whose value you wish to save. The `Property Bag` is available in the event procedure of the `WriteProperties` event as a parameter named `PropBag`. The example code in Listing 13.7 shows how you would call the `Property Bag's WriteProperty` method to save individual property values. Notice that we use whatever repository has been storing the property value as the source for the current value: at times this might be a private memory variable, and at other times it might be a property of a constituent control (as in the final line before the `End Sub`).

LISTING 13.7

USING THE WRITEPROPERTIES EVENT PROCEDURE TO SAVE PROPERTY VALUES TO THE PROPERTY BAG

```
Private Sub UserControl_WriteProperties _
  (PropBag As PropertyBag)
    'Store the values of the custom properties
    'to the Property Bag
    PropBag.WriteProperty _
      "BackColor", BackColor
    PropBag.WriteProperty _
      "Celsius", m_Celsius
    PropBag.WriteProperty _
      "Fahrenheit", m_Fahrenheit
    PropBag.WriteProperty _
      "TemperatureDate", m_TemperatureDate
    PropBag.WriteProperty _
      "Caption", lblCaption.Caption
End Sub
```

Calling the `PropertyChanged` Method to Trigger `WriteProperties`

The system automatically fires the `WriteProperties` and `ReadProperties` events whenever it thinks you may need their services. To ensure that the system knows a property has changed, you have to call the `PropertyChanged` method. An example of this would be when you change the value of a `Private` variable that implements the value of a property. The system will have no way of knowing that this variable is connected with a property, and therefore it will not fire the `WriteProperties` event based solely on the change you have made.

In such cases, you can call the `PropertyChanged` method. This method informs the system that a particular property has changed and so ensures that the `WriteProperties` event will fire before the current instance of the control is destroyed. If you've written the appropriate code in the `WriteProperties` event, then your property values will be stored in the `Property Bag`.

You should call the `UserControl`'s `PropertyChanged` method whenever you do something in code that will cause a change to a property whose value you wish to persist. The most typical place for you to call the `PropertyChanged` method would be in a `Property Let` or `Property Set` procedure (see Listing 13.8). Note that we check the `CanPropertyChange` method that we discuss in "Calling the `CanPropertyChange` Method Before Allowing a Property Value to `Change`."

LISTING 13.8

Calling the PropertyChanged Method to Ensure That WriteProperties Will Fire

```
Property Let Celsius(sValue As Single)
    If CanPropertyChange("Celsius") Then
        'assign incoming value to be stored
        'in Private variable
        m_Celsius = sValue

        'invoke UserControl's PropertyChanged method
        'so it knows to trigger WriteProperties and
        'store new value
        PropertyChanged ("Celsius")

        'perform other housekeeping specific to this
application
        Slider1.Value = m_Celsius
        RecalcFahrenheitFromCelsius sValue
        DisplayTempsFromSlider
    End If
End Property
```

Retrieving Persistent `Property` Values with the `ReadProperties` Event and the `ReadProperty` Method

The `ReadProperties` event fires when a custom control is re-instantiated at some point in the development cycle (the Project where it resides has been retrieved and its container has been instantiated, the developer has just entered run mode from the design mode, or the developer has just returned to design mode from run mode).

Notice that we said that `ReadProperties` fires when the custom control is *re*-instantiated. We used this phrasing to purposely exclude the case when the developer places an instance of the control on its container for the first time from the Toolbox. For such first-time instantiation, the `ReadProperties` event doesn't fire. Instead, the `InitProperties` event fires (see "Using the `InitProperties` Event to Set Default Starting Property Values"). The `ReadProperties` event, as its name implies, is the event that you will use to restore the values of properties that have been kept in the `Property Bag`. The `Property Bag` appears in the `ReadProperties` event procedure as a parameter named `PropBag`. You call `PropBag`'s `ReadProperty` method for each property whose value you wish to restore, as in Listing 13.9.

Notice that the `ReadProperty` method takes two arguments: the name of the property as a string and then a default value for the property (in case the property's value has not been initialized in the `Property Bag`).

We store the results of each call to `ReadProperties` in the appropriate variable or control property that implements the property within this control.

LISTING 13.9

USING THE READPROPERTIES EVENT PROCEDURE TO RESTORE PERSISTENT PROPERTY VALUES FROM THE PROPERTY BAG

```
Private Sub UserControl_ReadProperties _
 (PropBag As PropertyBag)
    m_Celsius = PropBag.ReadProperty("Celsius", 30)
    m_TemperatureDate = _
        PropBag.ReadProperty _
        ("TemperatureDate", DateSerial(1997, 1, 1))
    m_caption = PropBag.ReadProperty _
        ("Caption", Extender.Name)
```

```
    BackColor = PropBag.ReadProperty _
        ("BackColor", Ambient.BackColor)
    'make housekeeping adjustments
    'to bring constituent controls
    'into line with these property values
    lblCaption.Caption = m_caption
End Sub
```

Implementing Constituent Controls

Recall that one of the ways you can implement an ActiveX control is through constituent controls—other controls you use as building blocks for the functionality and appearance of your custom control.

You can build your custom control from constituents by simply placing instances of controls from the toolbox on the `UserControl`'s designer surface at design time, just as you would place controls on the surface of a Form in a standard EXE project (see Figure 13.6).

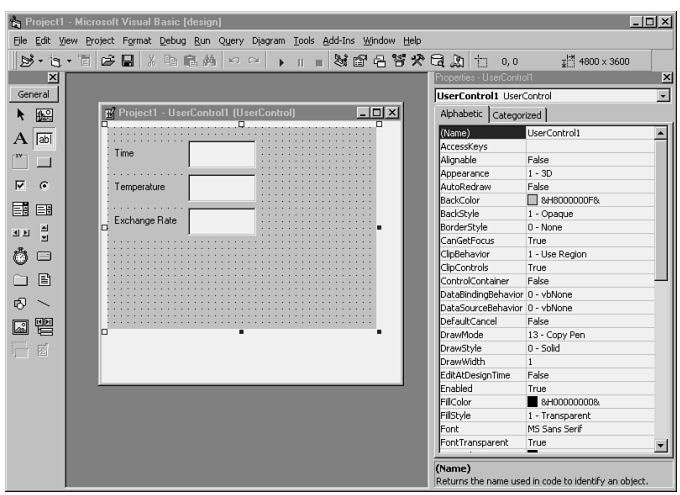


FIGURE 13.6
Placing constituent controls on a `UserControl`'s surface.

Managing Constituent Controls in the `Resize` Event

Constituent controls do not respond automatically as `UserControl`s are resized. Instead, you must programmatically make provisions for constituent controls to respond to changes in the `UserControl`'s size and shape.

The `UserControl`'s `Resize` event will fire when a developer resizes a design-time instance of your control and also when the control's shape or size change at runtime.

You can, therefore, employ the `UserControl`'s `Resize` event procedure to manage the appearance of the constituent controls within the ActiveX control and also perhaps to restrict the way in which the control can be resized.

In the example of Listing 13.10, we use the `Resize` event procedure to make sure that our custom control never shrinks below a certain minimum size. We also make sure that the constituent controls are always centered on the surface of the `UserControl` and bear the same relative positions.

LISTING 13.10

USING THE RESIZE EVENT PROCEDURE TO ENFORCE A CUSTOM CONTROL'S SIZE, SHAPE, AND THE VISUAL CONFIGURATION OF ITS CONSTITUENT CONTROLS

```
Private Sub UserControl_Resize()
    'minimum amount of space for borders
    Const MIN_BORDER_SIZE = 100
    Dim lMinHeight  As Long
    Dim lMinWidth As Long
    'minimum allowable height & width
    'are determined from min border size
    '& sizes of constituent controls
    lMinHeight = 2 * MIN_BORDER_SIZE + _
                 cmdApply.Height
    lMinWidth = 3 * MIN_BORDER_SIZE + _
                 cmdApply.Width + _
                 txtEnter.Width

    'if this UserControl has been
    're-sized to below minimum
    'sizes, then re-size it to
    'minimum allowable size
    If (ScaleHeight < lMinHeight) Then
        Height = lMinHeight + _
                  (Height - ScaleHeight)
    End If
    If (ScaleWidth < lMinWidth) Then
        Width = lMinWidth + _
                  (Width - ScaleWidth)
    End If

    'now figure out how big the
    'horizontal border can be
```

```
    'given the constituent control
    'sizes and the size of the UserControl
    Dim lhorizBorderSize As Long
    lhorizBorderSize = _
      (ScaleWidth - _
        txtEnter.Width - cmdApply.Width) / 3

    'set the horizontal alignment of
    'controls based on computed
    'horizontal border size
    txtEnter.Left = lhorizBorderSize
    cmdApply.Left = txtEnter.Left + _
                    txtEnter.Width + _
                    lhorizBorderSize

    'figure out vertical border
    Dim lVertBorderSize As Long
    lVertBorderSize = _
       (ScaleHeight - cmdApply.Height) / 2

    'set vertical alignment of controls
    cmdApply.Top = lVertBorderSize
    txtEnter.Top = lVertBorderSize
End Sub
```

Implementing Delegated Methods

If you wish to expose some of the methods of your control's constituent controls to the developer, then you will need to implement delegated methods. A delegated method is a custom control method that acts as a wrapper for the method of an underlying constituent control usually with a single line of code calling the constituent control's method.

A delegated method is the only way you can let the developer access a method of a constituent control since constituent controls are Private to the `UserControl` object and so are unavailable to the developer.

You may give the delegated method the same name as the constituent control method it implements, or you might give it a different name to show that it's a slightly different animal.

As an example of a delegated method, you might have a `ListBox` constituent control on your `UserControl` and wish to allow the developer to call the `ListBox` `Clear` method to clear out the items in the `ListBox`. You could provide a custom method, `ListClear`, to delegate the `Clear` method of the `ListBox`, as in Listing 13.11.

LISTING 13.11

DELEGATING THE CLEAR METHOD OF THE ListBox

```
Public Sub ListClear()
    lstMyList.Clear
End Sub
```

Implementing Delegated Events

If you wish to expose some of the events of your control's constituent controls to the developer, then you will need to implement *delegated events*. A delegated event is a custom control event that provides a wrapper for the event of an underlying constituent control, usually by placing a single line of code in the constituent control's event procedure to raise the custom event.

A delegated event is the only way you can let the developer see the events of a constituent control since constituent controls are private to the `UserControl` object, and so they're unavailable to the developer.

You may give the delegated event the same name as the constituent control event it implements, or you might give it a different name to distinguish it from the actual constituent control's event.

Let's say that you have a constituent `TextBox` control named `txtEntry` on your `UserControl` object and you wish its `Change` event to be visible in the client application. You would declare an event called, say, "`EntryChange`" in the `UserControl`'s General Declarations section that you see at the top of the example in Listing 13.12. Then in the `Change` event of the constituent `TextBox` control, you would simply raise the custom event, as shown in Listing 13.12.

LISTING 13.12

DELEGATING A CONSTITUENT TEXTBOX CONTROL'S CHANGE EVENT

```
[General Declarations of UserControl]
Public Event EntryChange()
[in the Constituent control's code]
Public Sub txtEntry_Change()
    RaiseEvent EntryChange()
End Sub
```

A developer would then see the `EntryChange` event procedure in code windows of projects that used this control, and the `EntryChange` event would fire whenever the constituent `TextBox` control's `Change` event fired.

Implementing Delegated Properties

If you wish to expose properties of your control's constituent controls to the developer, then you will need to implement *delegated properties*. A delegated property is a custom control property that provides a wrapper for the property of an underlying constituent control usually by placing code in the delegated property's `Property` procedures and modifying the lines in the `InitProperties`, `ReadProperties`, and `WriteProperties` event procedures that manage the property's value.

A delegated property is the only way you can give the developer access to the properties of a constituent control since constituent controls are private to the `UserControl` object and so their very existence, let alone their individual members, are unknown to the developer.

You may give the delegated property the same name as the constituent control property it implements, or you might give it a different name to distinguish it from the actual constituent control's property.

As an example of a delegated property, let's say that you have a constituent control, `txtEntry`, and you wish to expose its `Text` property to the developer using your control. You could create a custom property `EntryText` to delegate the `txtEntry.Text` property. To accomplish the property delegation, you'd follow these steps:

STEP BY STEP

13.2 Property Delegation

1. Create `Property Get` and `Property Let` procedures for the `EntryText` property (see Listing 13.13). Note that we don't use a `Private` variable in this example to store the property's value: Rather, we use the `Text` property of the constituent control `txtEntry`.
2. Put a line of code in the `InitProperties` event to implement the default value for this property. Note again in Listing 13.14 the use of the `Text` property of the constituent control rather than a `Private` variable as our repository for the property's value.

3. Put the appropriate code to manage the property's value into the `ReadProperties` and `WriteProperties` event procedures. Note once again in Listing 13.15 that we use the constituent control's `Text` property (and not a variable) to store and retrieve the property's value.

LISTING 13.13

PROPERTY LET AND PROPERTY GET PROCEDURES FOR A DELEGATED PROPERTY

```
Property Let EntryText(strValue as String)
    txtEntry.Text = strValue
    PropertyChanged "EntryText"
End Property
Property Get EntryText() as String
    EntryText = txtEntry.Text
End Property
```

LISTING 13.14

INITIALIZING A DELEGATED PROPERTY IN THE INITPROPERTIES EVENT PROCEDURE

```
Private Sub InitProperties()
    'Do some other initialization activies....

    txtEntry.Text = ""
End Sub
```

LISTING 13.15

MANAGING A PERSISTENT DELEGATED PROPERTY IN THE READPROPERTIES AND WRITEPROPERTIES EVENT PROCEDURES

```
Private Sub WriteProperties(PropBag As PropertyBag)
    'Write some other properties...

    PropBag.WriteProperty _
        "EntryText", txtEntry.Text
End Sub

Private Sub ReadProperties(PropBag As PropertyBag)
    'Read some other properties...

    txtEntry.Text = PropBag.ReadProperty _
        ("EntryText", "")
End Sub
```

Creating Data-Aware ActiveX Controls

Controls in the VB development environment can be aware of data in two ways:

- A control can be bindable to data fields in the Recordset of a `DataSource` control. Examples of some controls that can be bound to data in VB are `TextBox` controls, `Labels`, `ListBoxes`, and `ComboBoxes`.
- A control can be a `DataSource`, exposing a Recordset's data fields to which other controls can bind. The ADO Data Control discussed in Chapter 8 is an example of such a control.

In the following sections, we discuss how you can create your own ActiveX controls that are either data-bound controls like the `TextBox` or `DataSource` controls like the ADO Data Control.

Enabling the Data-Binding Capabilities of an ActiveX Control

As you may recall from Chapter 8, many controls provide `DataField` and `DataSource` properties, so you can bind them to a particular field provided in the Recordset of a Data Control.

You can implement these fields and other data-bound fields as well with ActiveX custom controls. The following sections discuss how you can do this.

Providing `DataSource` and `DataField` Properties with the Procedure Attributes Dialog Box

A bound control must have a `DataSource` property to indicate the Data Control to bind to and a `DataField` property to indicate which field to bind from the `DataSource`'s Recordset.

In addition, a moment's thought will tell you there is also a third property involved in this arrangement: a property that actually reflects the contents of the bound field. For a `TextBox`, this is the `Text` property, and for a `Label`, on the other hand, it's the `Caption`.

You can give your control `DataField` and `DataSource` properties and indicate a third property that will reflect the bound data by following these steps:

STEP BY STEP

13.3 Giving Your Control DataField and DataSource Properties

1. `Change` the `UserControl`'s `DataBindingBehavior` property's value to `1-vbSimpleBound`.
2. Make sure your text cursor is in the code window for your `UserControl`, preferably in one of the `Property` procedures of the property you wish to reflect the bound data.
2. Choose the Tools, Procedure **A**ttributes option from the VB menu (see Figure 13.7).
3. On the resulting dialog box, make sure the property that will reflect the bound data is chosen in the Names field.
4. Click the Advanced button.
5. In the Data Binding section at the bottom of the dialog, check that the box labeled `Property is data bound`, and then check that the box labeled `This property binds to DataField`.

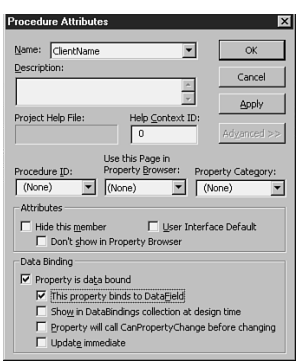


FIGURE 13.7
Specifying a property to bind to the `DataField`.

If you follow these steps, then developers using your control will see the `DataField` and `DataSource` properties in the properties window of each instance of your control. If the developer points the `DataSource` and `DataField` properties to a valid Data Control and a valid field in the Data Control's `Recordset`, then the custom property you specified in step 3 above will reflect the contents of the underlying data field in the designated property.

The custom property's `Property Let` procedure will fire every time the underlying data in the specified field changes. To be completely safe in the `Property Let` procedure, you should call the `CanPropertyChange` method, as described in the following section.

Calling the `CanPropertyChange` Method Before Allowing a `Property` Value to `Change`

You may wonder what will happen if you implement a data-bound property as described in the previous section only to have a developer bind your property to a field in a read-only data source. Obviously, there will be a problem when there is an attempt to update the contents of the property and, therefore, the underlying data.

You therefore need to take precautions to ensure that an attempt to update your property won't cause some sort of runtime error if the underlying data it's bound to can't be updated.

The `CanPropertyChange` method is supposed to return a Boolean value indicating whether it's safe to attempt to update a property's value. If `CanPropertyChange` returns `False`, then it's not safe to update the data underlying the property. You can use `CanPropertyChange` in the `Property Let` procedure. Only if it returns `True` will you perform the actions needed to update the property's value, as illustrated in Listing 13.16.

LISTING 13.16

CALLING THE CANPROPERTYCHANGE METHOD IN A PROPERTY LET PROCEDURE

```
Public Property Let LastName(ctrVal As String)
     If CanPropertyChange("LastName") Then
          m_LastName = ctrVal
          PropertyChanged "LastName"
     End If
End Property
```

NOTE

`CanPropertyChange` Doesn't Do Anything in Current Versions—Use Only for Future Compatability Note that we said that `CanPropertyChange` is *supposed* to return a result indicating whether it's safe to update a property. As of Version 6 of VB, VB's `CanPropertyChange` method *always* returns `True`. Microsoft still recommends that you use it, however, as you may want it in place for future compatibility.

So how does VB currently handle an attempt to update a read-only data field if the `CanPropertyChange` method always says it's OK to update? VB raises no runtime errors and simply ignores the request to update the field.

Creating an ActiveX Control that Is a Data Source

Starting with VB6, you can create an ActiveX control that functions as a *data source*. A data source control furnishes fields from a Recordset to which other controls can bind. Examples of data source controls that come out-of-the-box with VB would be the traditional Data Control and the ActiveX Data Control (discussed in Chapter 8).

The minimum that you'll need to do to implement a control as a data source is

1. Set the `UserControl`'s `DataSourceBehavior` property.
2. Program the `UserControl`'s `GetDataMember` event to return a reference to a Recordset object. This event fires whenever a data consumer (usually a bound control) has its `DataSource` property set to point to the data source control.

These two steps are enough if your data control's behavior will be very tightly constrained; that is, programmers who use the data source cannot determine the type of data connection nor the data that the data source exposes. In this restricted scenario, the `GetDataMember` event procedure will connect to a hard-coded set of records in a hard-coded database using a hard-coded data driver.

However, you may want to give programmers of your data source more choice about how the control connects to data. In that case you'll want to give programmers more of the features that standard Microsoft data source controls furnish, namely:

- Properties that allow the programmer to specify connect strings and the text of queries that retrieve data to create specific recordsets. The `GetDataMember` event procedure would then dynamically read these properties to initialize and return the Recordset.
- A `Recordset` property so that programmers can directly manipulate your data source control's Recordset in their own code.

The `GetDataMember` Event

The `GetDataMember` event occurs when the `DataSource` property of a data consumer that depends on the current control is set.

The purpose of this event is to return a reference to a valid Recordset object via its second parameter. This Recordset then becomes available to the data consumer that caused the event to fire in the first place. In Listing 13.17, the code in a `GetDataMember` event procedure always returns a reference to the Recordset of the `Employees` table in the `Nwind` Access database.

LISTING 13.17

THE GETDATAMEMBER EVENT PROCEDURE

```
Private Sub UserControl_GetDataMember(DataMember As String,
➥Data As Object)


    On Error GoTo GetDataMemberError

    ' rs and cn are Private variables of the UserControl
    ' if this is the first time through, then they haven't been
➥set yet
    If rs Is Nothing Or cn Is Nothing Then
        ' Create a Connection object and establish
        ' a connection.
        Set cn = New ADODB.Connection
        cn.ConnectionString =  _
            "Provider=Microsoft.Jet.OLEDB.3.51;Data
➥Source=c:\northwind.mdb"
        cn.Open
        ' Create a RecordSet object.
        Set rs = New ADODB.RecordSet
        rs.Open "employees", cn, adOpenKeyset,
➥adLockPessimistic
        rs.MoveFirst
    End If

    Set Data = rs

    Exit Sub

GetDataMemberError:

    MsgBox "Error: " & CStr(Err.Number) & vbCrLf & vbCrLf &
➥Err.Description
    Exit Sub
End Sub
```

Note in the listing that you manipulate two `Private UserControl` variables: A variable representing the ADO Connection and another representing the ADO Recordset. The event procedure code only has to set them up on the first pass through the procedure. Once the connection and the Recordset are initialized, you skip the initialization code.

At the end of the routine (just before the `Exit Sub` to detour around the error handler), the code assigns the Recordset to the second parameter, Data. This in effect returns the Recordset to whatever data consumer has just requested it (typically another control that has just had its `DataSource` property set to point to an instance of this control).

Steps to Create a Data Source Control

The following steps will implement a fully functioning data source control:

STEP BY STEP

13.4 Creating a Data Source Control

1. Create a new ActiveX control project.
2. Set a reference in the project to the appropriate data library through the Project, References menu dialog box.
3. Set the `UserControl`'s `DataSourceBehavior` property to `1-vbDataSource`.
4. Create property procedures for custom properties that programmers will use to manipulate the data source control's connection to data. Typically, you'll implement `String` properties such as `ConnectString` (connection string to initialize a `Connection` object) and `RecordSource` (string to hold the query to initialize the data in the recordset). Create private variables to hold the values of each of the properties. Create private constants to hold their initial default values. Program the `InitProperties`, `ReadProperties`, and `WriteProperties` event procedures to persist these properties.
5. If you want to expose the control's Recordset for other programmers to manipulate, then you should create a custom property name, RecordSet. Its type will be the appropriate Recordset type that you plan to program for your control. You may choose to make it read-only, in which case you only need to give it a `Property Get` procedure. Declare a private object variable to hold its value using `WithEvents` (this exposes the event procedures to other programmers).

6. Declare a `Private` variable of the appropriate connection type that you plan to program for your control. It will not correspond to a custom property, but it's necessary in order to host the Recordset.
7. Code the `InitProperties`, `ReadProperties`, and `WriteProperties` events to properly manage and persist the values of the properties created in the previous steps.
8. Program the `UserControl`'s `GetDataMember` event procedure to initialize a recordset and return it in the second parameter. You will derive the Recordset either from information contained in custom `Private` variables (see Exercise 13.6 for an example) or from hard-coded information in the `GetDataMember` event procedure itself (see the previous section for an example). You should perform some error-trapping to ensure that you do indeed have a valid connection.
9. Put code in the `UserControl`'s `Terminate` event that will gracefully close the data connection.
10. If you want to allow users to navigate data by directly manipulating your `UserControl`, then put the appropriate user interface on your `UserControl` along with the code to navigate the Recordset variable.
11. Your new ActiveX control should now be ready to test as a `DataSource`:Add a standard EXE project to the Project Group. Now, making sure you've closed the designer for the `UserControl`, add an instance of your new control to the standard EXE's form.
12. Manipulate any necessary custom properties (such as `ConnectString` or `RecordSource`) that you may have put in your custom control.
13. Put one or more bindable controls in the test project and set their `DataSource` property to point to the instance of your Data Source Control. Set their `DataField` properties to point to fields from the exposed Recordset.

Create and Enable Property Pages for ActiveX Controls

Property Pages provide an alternative to the Properties Window for a developer who wants to edit property values at design time. As a developer you're already familiar with the property pages of standard and custom VB controls.

You should consider using Property Pages with your ActiveX control when

- your control has numerous custom properties that can be easily organized into categories.
- your control uses complex custom properties that don't represent just a single value.
- many of your custom properties are more easily edited with non-text controls, such as drop-down lists, option buttons, or check boxes.

You implement Property Pages as special modules of your ActiveX control project (.pag files).

A control can have many Property Pages (as do the `DBGrid`, `ToolBar`, and `StatusBar` controls for example) depending on the complexity and logical groupings of its properties.

To program Property Pages, you need to know how to program with:

- **The `PropertyPage` designer**. This is a Form-like interface similar to the `UserControl` designer or the designer for regular VB Forms. You place intrinsic VB controls on its surface to provide the developer with an interface for editing property values.
- **`SelectedControls` collection**. This collection holds the controls that the developer has currently selected.
- **`SelectionChanged` event**. When a developer displays the Property Page or changes selected control(s), you use the property values of the first selected control to initialize controls on the Property Page.

- **`EditProperties` event**. Use its `PropertyName` argument to determine which complex control a user is trying to edit.
- **`Changed` property**. This property flags whether changes have been made to properties with the Property Page.
- **`ApplyChanges` event**. This event fires when developers save changes they have made to properties.

After you've programmed the behavior of your Property Pages, you'll need to take one or more of the following actions to cause your control to display Property Pages correctly:

- Connect custom controls to your Property Page.
- Connect individual complex properties to your Property Page.
- Connect some properties to standard VB Property Pages.

We discuss the features of the above two lists in the following sections.

Creating the `PropertyPage` Object's Visual Interface

Once you've decided which properties you are going to implement with a Property Page, you can:

- Add a Property Page module to your control project with the Project, Add Property Page option on your VB menu.
- Place appropriate controls for editing the properties on the `PropertyPage` object's designer surface.

The Property Page under development in Figure 13.8 allows the developer to edit the values of a property in two different ways.

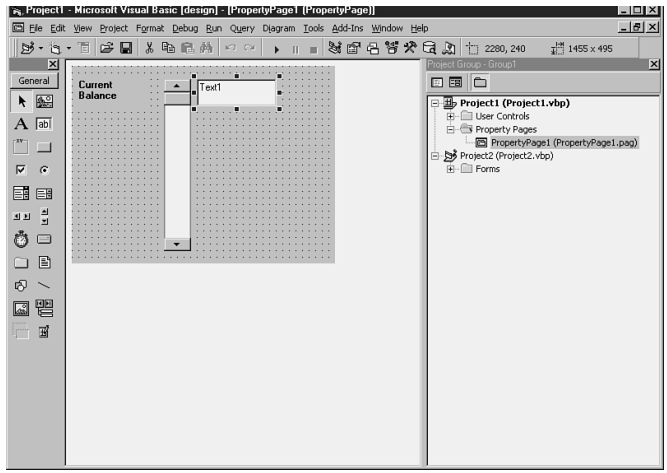


FIGURE 13.8
Editing the visual interface for a custom Property Page.

Determining which Controls Are Selected for Editing with the `SelectedControls` Collection

You'll recall that you can select multiple controls when designing a VB application by holding the Ctrl key and clicking the mouse on successive controls in the same container. You can also select multiple controls in the same physical area of the container by using the "lasso" method: holding down the mouse button outside any control on the container and keeping it down while you "lasso" or mark an area holding several controls (a rectangle of dotted lines appears around the lassoed area).

It's therefore possible that the developer could have selected more than one control when your Property Page is active. In order for you to detect which controls the developer has currently selected, the `SelectedControls` collection is available in the code of a Property Page. It contains all controls currently selected by the developer.

The `SelectedControls` collection is zero-based. That is you would denote the first element of `SelectedControls` in your code by:

```
SelectedControls(0)
```

The `SelectedControls` collection has a `Count` property to tell you how many controls are in the collection. Since `SelectedControls` is zero-based, you would denote the last control in the collection by:

```
SelectedControls(SelectedControls.Count - 1)
```

You can refer to the property settings of one of the controls in `SelectedControls` with the normal *object.property* syntax, as in this example:

```
SelectedControls(0).Backcolor
```

Normally you'll use the `SelectedControls` collection at the following places in your Property Page's code:

- In the `SelectionChanged` event procedure, in order to set Property Page values to those of the first selected control, or `SelectedControls(0)`, see the following section titled "Using the `SelectionChanged` Event to Detect When the Developer Begins to Edit Properties."
- In the `ApplyChanges` event procedure, in order to determine which controls should receive property changes (some property values might not be appropriate for all controls), see "Saving Property `Changes` with the `ApplyChanges` Event."

Using the `SelectionChanged` Event to Detect When the Developer Begins to Edit Properties

The `SelectionChanged` event of a Property Page happens when

- The Property Page displays.
- There is a change in the selected controls (the `SelectedControls` collection) while the Property Page is displayed.

You therefore use the `SelectionChanged` event procedure to set the values of Property Page elements that reflect property values.

Typical code in the `SelectionChanged` event procedure will check the values of properties in the first control of the `SelectedControls` collection (`SelectedControls(0)`) and adjust Property Page elements accordingly, as we do in the example of Listing 13.18.

It's not appropriate to allow some properties to change to a single value for multiple controls. For instance, the `Caption` properties of most controls should be unique to each control. Therefore, you might want to use the `SelectionChanged` event procedure to control which properties can change when the developer has selected multiple controls.

Notice that in Listing 13.18, we check the value of `SelectedControls`.Count to see whether the developer has selected multiple controls. If so, we disable the Property Page elements that correspond to property values that we don't want to set for multiple controls. If, on the other hand, only one control has been selected, we enable its corresponding Property Page element and assign to that element the value of the property from the selected control.

LISTING 13.18

ADJUSTING PROPERTY PAGE ELEMENTS IN THE SELECTIONCHANGED EVENT PROCEDURE

```
Private Sub PropertyPage_SelectionChanged()
    chkIsFahrenheit.Value = _
        SelectedControls(0).IsFahrenheit
    If SelectedControls.Count > 1 Then
        txtTempDate.Enabled = False
    Else
        txtTempDate.Text = _
            SelectedControls(0).TempDate
    End If
End Sub
```

Flagging Property `Changes` with the `Changed` Property

You must set the Property Page's `Changed` property to `True` in order to enable the Apply button on the Property Page.

The most appropriate place to set the `Changed` property is in the corresponding event procedures of the controls you've placed on the Property Page for editing the values of properties.

For example, if a `TextBox` control on your Property Page represents the value of a property, you would set the `Changed` property in the `TextBox` control's `Changed` event procedure. Again, if an `OptionButton` control represents a property value, then you'd set the `Changed` property to `True` in its `Click` event procedure.

Saving Property `Changes` with the `ApplyChanges` Event

You can use the `ApplyChanges` event to write the information from the Property Page back to the actual property values of the selected controls.

`ApplyChanges` fires when the user Clicks the Apply button or the OK button on your Property Page or switches to another Property Page of your control.

As you can see in Listing 13.19, we use the `SelectedControls` collection to determine the controls that the developer has selected for modification. Notice that just as we did in the `SelectionChanged` event procedure, we distinguish between properties that should be changed for just one control and properties that can be changed for all controls.

LISTING 13.19

SAVING PROPERTY CHANGES BACK TO CONTROLS IN THE APPLYCHANGES EVENT PROCEDURE

```
Private Sub ApplyChanges()
    'This change only applies if
    'just one control is selected
    If SelectedControls.Count = 1 Then
        SelectedControls(0).MyName = _
          txtMyName.Text

    End If
    'This change only applies to
    'the first control in the selection
    SelectedControls(0).Climate = _
        txtClimate.Text
    'This change applies to all
    'controls in the selection
    Dim ctrl As Control
    For Each ctrl In SelectedControls
        ctrl.TempDate = txtTempDate.Text
    Next ctrl
End Sub
```

Connecting a Custom Control to a Property Page

After you've programmed your Property Page, you must then associate it with a custom control. To connect a Property Page to a custom control, take the following steps:

STEP BY STEP

13.5 Connecting a Property Page to a Custom Control

1. Select the Custom Control's `UserDocument` in your VB project.
2. Bring up the Custom Control's Properties Window (F4 key).
3. Select the `PropertyPages` property and click the ellipsis (…) to its right.
4. In the resulting Connect Property Pages dialog box (see Figure 13.9), check all the Property Pages that you want to associate with this control.

NOTE

Using the Same Property Page for Multiple Controls If your VB project has more than one ActiveX custom control, you may connect the same Property Page to more than one type of ActiveX control.

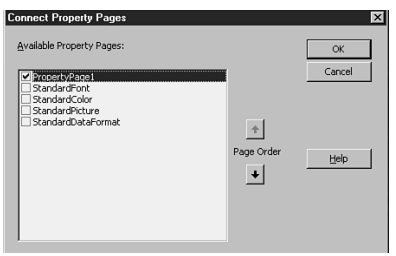
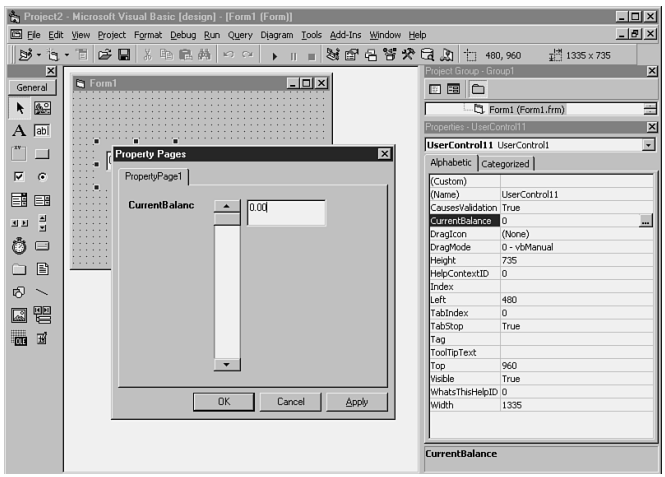


FIGURE 13.9
Use the Connect Property Pages dialog box to associate one or more Property Pages with a custom ActiveX control.

Connecting a Single Complex Property to a Property Page

It's possible to connect one or more complex properties directly to a Property Page rather than an entire control. When you connect a property directly to a Property Page in this way, the entry for the property in the Properties Window displays an ellipsis (…) to the right of the property value (see Figure 13.10). The developer can click on the ellipsis to bring up a Property Page for that property.



◀**FIGURE 13.10**
Complex properties that require a Property Page show an ellipsis (...) beside their entry in the Properties Window.

To associate a single property with a Property Page, take the following steps:

STEP BY STEP

13.6 Associating a Single Property with a Property Page

1. From a Code Window in your `UserControl`'s code, choose Tools, Procedure Attributes from the VB menu to bring up the Procedure Attributes dialog box.
2. Make sure the name of the property you want to work with appears at the top of the dialog box in the Name field.
3. Click the Advanced button and pull down the choices in the list titled "Use this Page in Property Browser" (see Figure 13.11).
4. Choose the Property Page that you want to connect with this property.
5. Click the OK button to confirm your choice and finish.

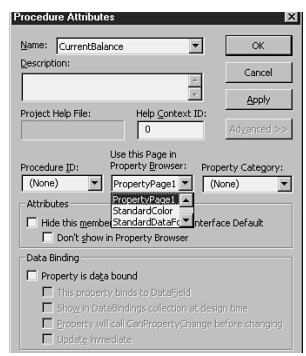


FIGURE 13.11▲
Associating a single property with a Property Page.

Detecting which Complex Property is being Edited with the `EditProperty` Event

The `EditProperty` event fires when the developer starts to edit a property that you have directly connected to a Property Page.

As we discussed in the previous section, it's possible to connect one or more complex properties directly to a Property Page rather than an entire control. When you connect a property directly to a Property Page in this way, the entry for the property in the Properties Window displays an ellipsis (…) to the right of the property value. The developer can click on the ellipsis to bring up your Property Page for that property.

Because it's possible to connect more than one complex property directly to a Property Page, there may be elements on the Property Page that aren't appropriate for every property that you've connected to it.

That is where the `EditProperty` event comes in handy. You can use the `EditProperty` event's string property parameter, `PropertyName`, to check on which property the developer is trying to edit, as we do in Listing 13.20. Depending on which property the developer is editing, you can enable, disable, hide, reveal, or set focus to elements on the Property Page.

LISTING 13.20

USING THE EDITPROPERTY EVENT PROCEDURE TO DETERMINE THE CONFIGURATION OF THE PROPERTY PAGE

```
Private Sub EditProperty(PropertyName As String)
    If Ucase$(PropertyName) = "TEMPDATE" Then
        txtTempDate.SetFocus
        chkIsFahrenheit.enabled = False
    Else
        . . . other stuff
    End If
End Sub
```

Connecting a Property to a Standard VB Property Page

If you have a custom property that represents a bitmap picture, a system font, or a color, then you're in luck: VB provides three standard Property Pages that you can use with the appropriate type of property in your ActiveX control project.

These standard Property Pages keep you from having to "reinvent the wheel" for commonly used property interfaces (such as bitmaps, color, and font dialogs).

In order to connect one of your custom properties to a standard Property Page, just follow the same steps as you would follow to connect a property to a custom Property Page (see the previous section). Instead of choosing a custom Property Page in the Procedure Attributes dialog box, choose the appropriate standard Property Page.

NOTE

OLE_COLOR-Type Properties Have Automatic Color Dialog Boxes If you create a custom property and declare its type to be OLE_COLOR, then VB will automatically show the `Color` property dialog box to the developer in the Properties Page of an instance of your control. Remember that you must then consistently refer to its type as OLE_COLOR throughout the control's code (in its `Private` variable and `Property` procedure declarations).

Testing and Debugging your ActiveX Control

You can test and debug your ActiveX Control project from the design-time environment in one of two ways:

- Test your ActiveX control inside an existing container application, such as Internet Explorer.
- Create a Standard EXE test project in the same program group as your ActiveX control and test your ActiveX control in the test project.

Notice that both methods require another program in order to test your ActiveX control. You cannot test an ActiveX control project completely on its own because, of course, ActiveX controls have no function outside of a container provided by another application.

We discuss both these methods in the following sections.

Testing Your ActiveX Control with Existing Container Applications

The most common container applications that you will want to test against your ActiveX control would be Internet browsers that support ActiveX controls such as Internet Explorer.

In order to test your ActiveX control with a browser, you will use the Debugging tab of VB's Project, Properties dialog box to indicate your control's behavior when you run it from the design-time environment.

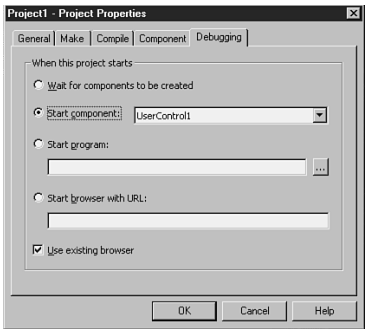
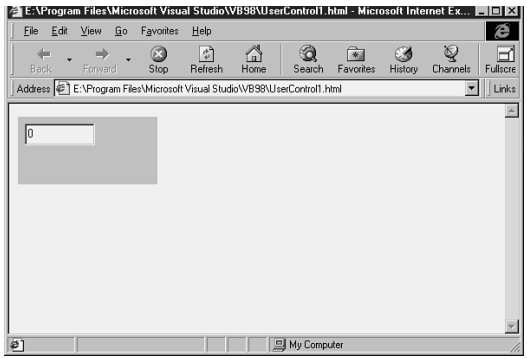


FIGURE 13.12
The Debugging tab of the Project, Properties dialog box.

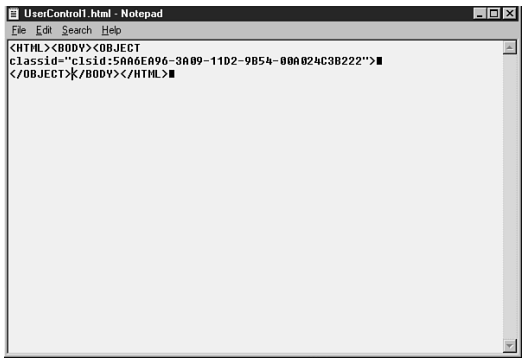
STEP BY STEP

13.7 Testing Your ActiveX Control with a Browser

1. Make sure that your ActiveX control project is the startup project in its Project Group (only necessary if there are other projects in the Project Group).
2. Run the application. The first time you do this, the Project Properties dialog box appears with its Debugging tab selected, as in Figure 13.12. You will typically want to accept the default settings with the Start Component option button selected and the Use Existing Browser check box checked.
3. Internet Explorer will load, and an instance of your control will appear in the Internet Explorer window frame, as in Figure 13.13.
4. Note that you can choose IE's View Source menu option to see the sample page that was created with your project's class ID, as in Figure 13.14. If you wish at this point, you could modify the HTML source to manipulate your control with, say, VBScript and further test its behavior in a Web page.
5. If you later want to change debugging options, such as the type of container that your control runs in, then you must use the Project, Properties menu option and choose the Debugging tab.



◀FIGURE 13.13
Internet Explorer, running with the test control loaded.



◀FIGURE 13.14
Viewing the HTML source for the test Internet Explorer page created for your control.

Testing and Debugging your ActiveX Control in a Test Project

The testing method discussed in the previous section is fine if you want to see how your control behaves in an existing application. However, you will also want to see how your control behaves when other programmers use it in their own VB projects. In order to test your ActiveX control in the VB design environment, you need to use a test project in the same Project Group as your ActiveX control.

To test your control with another VB project, you should take the following steps:

STEP BY STEP

13.8 Testing Your ActiveX Control with Another VB Project

1. Choose the File, Add Project menu option and add a standard EXE project to your Project Group.
2. Make sure that the new project is the Startup project of the Project Group by right-clicking on the project's entry in the Project explorer and choosing Set as Startup from the menu (see Figure 13.15).

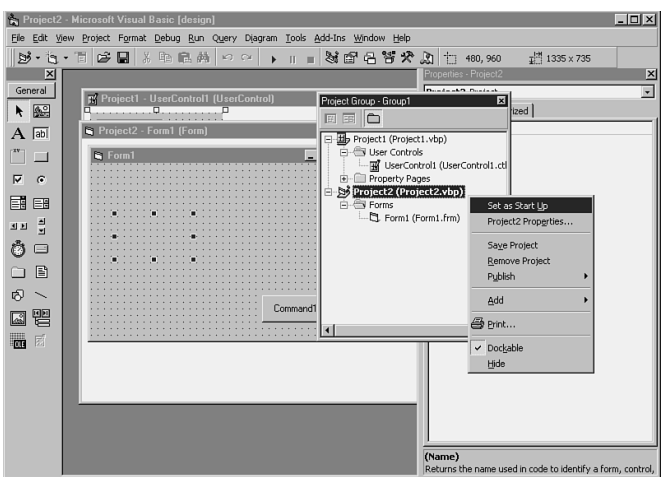


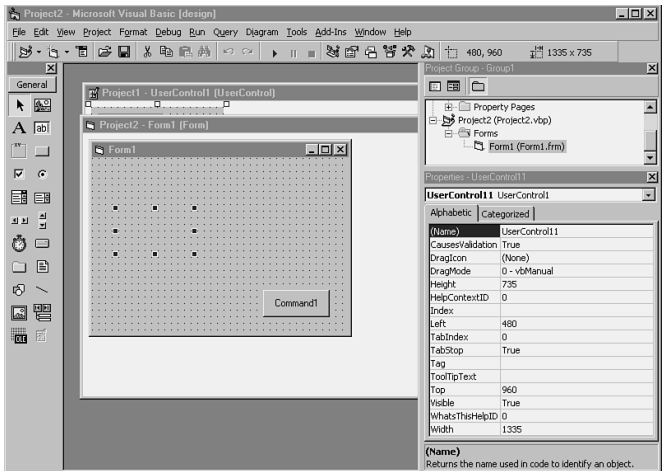
FIGURE 13.15
Add a standard EXE project to test your custom control and make sure it's the Startup project of the Project Group.

3. Make sure to close the Designer window for your `UserControl` object. If you forget to close its Designer, the custom control won't be available in your test project. You'll be able to see it in the test project, but its toolbox image and any instances you've already placed on test forms will be disabled, as illustrated in Figure 13.16).
4. Switch to the test project, and you will see the `UserControl`'s `ToolBoxBitmap` or default ActiveX control bitmap in the toolbox (see Figure 13.17).

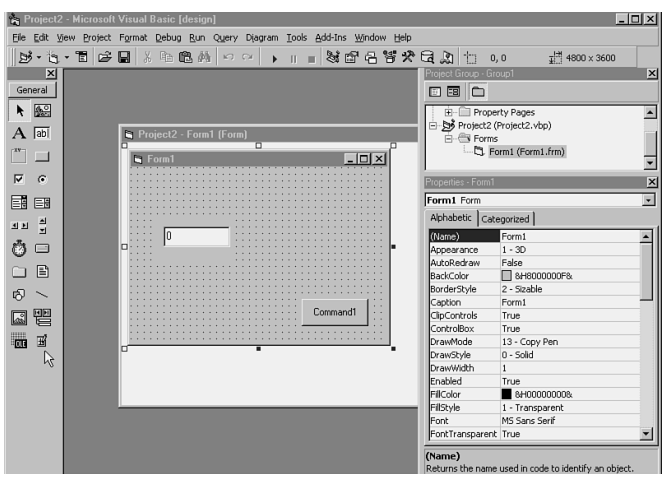
NOTE

ActiveX Control Projects Shouldn't Be the Startup of Multiple Project Groups
An ActiveX Control project shouldn't be the startup project of a Project Group—so it's a good idea to get into the habit of making sure that the test project is the Startup of your group.

In VB5, it was impossible for an ActiveX control to be the Startup project, but in VB6 an ActiveX control can be the startup, as discussed in the previous section.



◀FIGURE 13.16
This custom control's design-time instance is disabled in the test project because the developer hasn't closed the Designer window for the control itself.



◀FIGURE 13.17
This toolbox contains the default ActiveX control bitmap for a custom control that is being tested.

5. Place an instance of your control from the toolbox on the test project's startup form. Write code and manipulate properties to exercise your control.

6. Run the test project to observe the control's behavior.

What to Look for When Testing Your Activex Control

In the following sections, we discuss some considerations for testing the features that you have built into your custom ActiveX control.

In order to test these features, you must run a test project against your control, as discussed in the previous sections.

Here is a list of things you should do to make sure that your control behaves in a way that developers will recognize as standard control behavior.

- Use an instance of your control to test its methods by calling each method in code from the code of the test project.
- Test your control's events by placing an instance of your control in the test project. Then follow these steps for each event:
  - Write code in the test project that will trigger the event (what the test code needs to do depends on how you've designed the event).
  - In the test project's code, find the event procedures for your control instance's events. Then put code in the event procedures that will test whether the event fired. `Debug.Print` messages are a good idea. Message boxes are not such a good idea because they can interrupt the flow of processing, especially when keyboard input is important. Also include code to test the value of any parameters passed by the event procedure.
- Test your control's properties by placing an instance of your control in the test project. Make sure you test the following features of each property:

  Make sure that the control starts with its properties initialized to their default values: Add an instance of your control to the test project and make sure that the initial property values are what you expect. You should have specified default values in the `UserControl`'s `InitProperties` event procedure.

  Make sure that default values don't overwrite developer changes when the project goes from design time to runtime:

This might happen if you've mistakenly assigned default values in the `ReadProperties` event procedure.

Make sure that design time changes to controls persist to runtime. Check the `WriteProperties` and `ReadProperties` event procedures as well as the `Property Let/Set` and `Get` procedures if there are problems. Make sure you're calling the `PropertyChanged` method in `Property Let/Set` procedures.

Make sure that runtime changes actually take effect at runtime: Make sure you're storing property values in the appropriate intermediate variables or underlying delegated properties in the `Property Let/Set` procedures.

Make sure that data-bound properties correctly reflect underlying data: If you have implemented data-bound properties, then check your control's properties for a `DataSource` property. (If it's missing it means you haven't designated any property as the `DataField` property). Put a Data Control on your test form, connect it to data, and set the `DataSource` property to point to the Data Control. Then set the `DataField` and other data-bound properties to point to the fields provided by the Data Control's Recordset.

CASE STUDY: AN ACTIVEX CONTROL TO PROVIDE A STANDARD INTERFACE FOR CURRENCY AMOUNTS

NEEDS

Your business needs a common look and feel for entry of currency amounts on user input screens.

In the past, there have been currency-amount entry problems ranging from slight annoyances to costly mistakes.

Your business' Windows programmers are scattered across many unconnected locations. Standardization between all of them is going to be a problem as well.

REQUIREMENTS

Some of the problems encountered in the entry of currency rates might include:

- Confusion about what national currency was being used for the amount displayed in a field.
- In some environments, however, the currency symbol is seen as an annoyance and a waste of screen real estate.

continues

CASE STUDY: AN ACTIVEX CONTROL TO PROVIDE A STANDARD INTERFACE FOR CURRENCY AMOUNTS

continued

- On some screens, the requirements of the interface to mainframe data might mean that the underlying number needed to be stored with no decimal places, that is 100.03 would be stored as 10003 and 5.00 would be stored as 500. The application that consumes this data would need to be intelligent enough to know when to divide and multiply by factors of 100 both for display purposes and for computation.
- Some users might prefer nicely formatted numbers with thousands separators while others (especially those using larger numbers and requiring more space) might not like them.
- There will be localization issues since thousands separators and decimal-place markers vary internationally.

DESIGN SPECIFICATIONS

You might be able to meet these needs by distributing an ActiveX control for currency entry to all your Windows programmers. This control would enforce your company's user-interface and data storage standards and could be flexible enough to accommodate different situations.

In addition by distributing a compiled control to your company's programmers, you could automatically enforce standard programming for currency amounts across the organization.

General specifications

The control should be

- bindable to data (a data consumer with `DataField` and `DataSource` fields).
- resizable.

The following are proposed members of the control. The names should be self-explanatory.

Properties

`DisplayAmount`

`StoreAmount`

`ThousandsSeparator`

`DecimalCharacter`

`DecimalPlacesToShow`

`CurrencyCode`

`ShowCurrencySymbol`

`DecimalPlacesToShift`

Methods

`ApplyCurrencySymbol`

`ApplyThousandsSeparator`

`UnApplyCurrencySymbol`

`UnApplyThousandsSeparator`

Events

`Change` (delegated event of underlying `TextBox`)

`KeyPress` (delegated event of underlying `TextBox`)

CHAPTER SUMMARY

We have covered the following key points in this chapter:

- Standalone ActiveX controls and ActiveX controls implemented as part of larger projects
- ActiveX controls created from constituent controls
- User-drawn ActiveX controls
- ActiveX control lifetime
- Steps for creating an ActiveX control
- The `UserControl` object
- The `Extender` and `Ambient` objects
- Implementing custom methods and events
- Implementing custom properties and property persistence with the `InitProperties`, `ReadProperties`, and `WriteProperties` events and with the `Property Bag` object and its methods
- Implementing constituent controls including delegated members
- Creating controls that can be data consumers
- Creating controls that serve as data sources
- Creating Property Pages
- Testing and debugging an ActiveX control project

KEY TERMS

- ActiveX control
- Composite control
- Constituent control
- Data consumer
- Data source
- Delegated member
- Designer
- Persist
- Property Page
- Standard control

APPLY YOUR KNOWLEDGE

Exercises

13.1 Creating an ActiveX Control

In this exercise, you create a simple ActiveX control that can track and change a running balance. The property will have one event, one method, and one property. The event will be called `BalanceChanged` and will pass two double parameters. The method will be named `AddToBalance` and will take a single double parameter that will modify the `CurrentBalance` property. The method will also raise the `BalanceChanged` event. The property will be of type Double and will be called `CurrentBalance`. It will raise the `BalanceChanged` event whenever it is reset to another value. The `CurrentBalance` property will not be fully functional until you've completed Exercise 13.2 where you implement property persistence. In Exercise 13.3, you'll run this control in a test project, and in Exercise 13.4, you'll add some more properties and a Property Page to the project. To complete this exercise, follow these steps:

Estimated Time: 25 minutes

1. Create a new ActiveX Control project.
2. Implement an event named `BalanceChanged` by declaring it in the general section of the `UserControl`. In the event's declaration, specify two double parameters named `NewBalance` and `AmountOfChange`.
3. Implement a double property named `CurrentBalance`. Create a `Private` variable to hold its value and name the variable `m_CurrentBalance`.
4. In the `Property Let` procedure for `CurrentBalance`, raise the `BalanceChanged` event passing it the appropriate parameters (you'll have to compute the amount of change to pass as the second parameter to the event).
5. Implement a method named `AddToBalance` by creating a `Public Sub` procedure. Give it a double-type parameter named `AmountOfChange`, and in the procedure recompute the new balance based on its previous value and the `AmountOfChange` parameter's value. Then raise the event you declared in step 2, passing it the new balance and the amount of change as parameters.
6. Now you can test the control as described in Exercise 2. Note that because you haven't yet done anything to persist its properties, it will not behave correctly in the design-time environment.

13.2 Testing and Debugging an ActiveX Control

In this exercise, you will run the ActiveX control project created in Exercise 1 in the design environment to test it. You will use two different methods: running in a test project and running inside Internet Explorer. To complete this exercise, follow these steps:

Estimated Time: 25 minutes

1. Use the same ActiveX control project that you created in Exercise 1.
2. Without adding a test project, run the application. The first time you do this, the Project Property dialog box appears with its Debugging tab selected, shown in Figure 13.12. You will typically want to accept the default settings with the Start Component option button selected and the Use Existing Browser check box checked.
3. Internet Explorer will load and an instance of your control will appear in the Internet Explorer window frame.

   Note that you can choose IE's View Source menu option to see the sample page created with your project's class ID.

APPLY YOUR KNOWLEDGE

If you wished at this point, you could modify the HTML source to manipulate your control with, say, VBScript and further test its behavior in a Web page.

4. In order to test your control in the environment of a generic Windows application, make sure that your control project's designer is closed, and then add a new standard EXE project to the Project Group.
5. Make sure that the test project is the Project Group's Startup project. If it's not, then the ActiveX control project will behave exactly as before in the design-time environment, calling up Internet Explorer and running inside it.
6. Put an instance of your control on the default form of the test project and also place a `CommandButton` on the form.
7. In the `CommandButton`'s `Click` event, put code to call the control's `AddtoBalance` method.
8. In the form's code window, locate your control's instance in the drop-down list and then enter code in its `BalanceChanged` event to alert the user of a change in balance.
9. Run the project and click several times on the `CommandButton`, noting that the amount continues to increment with each click beginning at 0.
10. Stop the test project. Call up the Designer to display it on the screen. Then return to your test project without closing the Designer. Note that the image of your control's instance is cross-hatched, indicating that it's unavailable. This is because the control project's Designer is also loaded at the moment.

13.3 Using Control Events to Save and Load Persistent Properties

In this exercise, you will make your ActiveX control's custom properties persistent: That is, you will program with the `InitProperties`, `ReadProperties`, and `WriteProperties` event procedures and the `PropBag` object. To complete this exercise, follow these steps:

Estimated Time: 25 minutes

1. Use the project group created in Exercises 1 and 2. Make sure you're looking at the EXE project and, in Design Mode, open the Properties Window for your control and change the `CurrentBalance` property to some number besides 0. Run the project again and note that your application ignores the design-time setting, beginning again at 0 for the `CurrentBalance` property.
2. Stop the application and check the `CurrentBalance` property in the design-time window. You will notice that the environment has lost track of the value you placed into `CurrentBalance` property and is back to 0.
3. The problem in steps 1 and 2, of course, is that you've done nothing yet to implement property persistence in your control. The rest of this exercise walks you through the basic steps for making a custom property persistent for an ActiveX control.
4. In the General Declarations section of your `UserControl`, declare a Private constant `m_def_CurrentBalance` and assign it the default value that you want the control to take on when it's first sited in a container.
5. In the `UserControl`'s `InitProperties` event procedure, assign the default value (`m_def_CurrentBalance`) to the `Private` variable `m_CurrentBalance`, the variable that stores the value of the `CurrentBalance` property.

APPLY YOUR KNOWLEDGE

Recall that the `InitProperties` event procedure is the preferred place to initialize property values. If you were to use the `Initialize` event procedure, the default value would override any value you'd assigned in the property window every time you ran your host application.

6. In the `UserControl`'s `ReadProperties` event procedure, use the `Property Bag`'s `ReadProperty` method to assign the stored value for the `CurrentBalance` property to the `m_CurrentBalance` memory variable using `m_def_CurrentBalance` as the default value.
7. In the `Property Let` procedure for the `CurrentBalance` property, make sure to call the `UserControl`'s `PropertyChanged` method for the `CurrentBalance` property. This call will alert the system to the fact that a property has changed and will guarantee that the `WriteProperties` event will fire when the control goes out of memory.
8. In the `WriteProperties` event procedure, make sure that the current value of the property gets saved in the `PropBag` by calling its `WriteProperty` method.
9. Retest your control with the test project as in steps 1 and 2. This time you should see that the system correctly handles your design-time changes to the property's value.

13.4 Creating a Property Page

In this exercise, you create a Property Page and map its controls to the `CurrentBalance` property you created in your ActiveX control in Exercises 1 and 2. To complete this exercise, follow these steps:

Estimated Time: 30 minutes

1. Open the Project Group that you worked with in Exercises 1 through 3. Make sure that you have the ActiveX control project selected and not the test project.
2. Add a Property Page to the ActiveX control project.
3. Add a `TextBox` to the Property Page. Put a validation code in the `Change` event so that the value of the `TextBox` is always numeric.
4. Make sure that the Property Page will use the `CurrentBalance` property of the current control by putting code in the `SelectionChanged` event procedure. This code will assign the first selected control's `CurrentBalance` property value to the `TextBox`.
5. Make sure that the system knows about changes that the user makes on the Property Page by setting the Property Page's `Changed` property to `True` in the `TextBox` control's `Change` event.
6. Make sure that the changes get applied properly back to the control's property by coding the `ApplyChanges` event procedure. Essentially, you'll reverse the assignment that you made in the `SelectionChanged` event procedure, writing the value of the `TextBox` back to the selected control. Since the user may have selected more than one control, it would be more robust to use a `For Each...` loop to visit all selected controls and change each control's `CurrentBalance` property.
7. Add the Property Page to the ActiveX control's Property Page collection by navigating back to the control project, opening the control's Properties Window and selecting the `PropertyPages` property from the list. Check the box for this Property Page project.

APPLY YOUR KNOWLEDGE

8. In the ActiveX control project, connect the `CurrentBalance` property to this Property Page. First make sure that your cursor is in some procedure of the Control's Code Window. Choose the name of the `CurrentBalance` property from the Name drop-down list. Then choose Tools, Procedure Attributes from the VB menu and click the Advanced button. Choose the name of your Property Page project from the drop-down list labeled Use This Page in Property Browser.

9. Bring up the Properties Window for the instance of your control in the test EXE project and click on the ellipsis for the `CurrentBalance` property to test the Property Page.

13.5 Enabling the Data-Binding Capabilities of an ActiveX Control

In this exercise, you make sure that your ActiveX control can be bound to a Data Source such as VB's ADO Data Control. To complete this exercise, follow these steps:

Estimated Time: 30 minutes

1. Start a new ActiveX control project.

2. In the `UserControl` object's Properties Window, change the `DataBindingBehavior` property's value to `1-vbSimpleBound`.

3. Put a `TextBox` control in the project and name it `txtClientName`. You will delegate its Text property in a property called `ClientName`, as described in the following steps.

4. Create `Property Let` and `Property Get` procedures for a property called `ClientName`, and use the `Text` property of the `TextBox` to store the value of this property.

```
Property Let ClientName(sNewVal As String)
    txtClientName.Text = sNewVal
End Property

Property Get ClientName() As String
    ClientName = txtClientName.Text
End Property
```

5. To persist the `ClientName` property, put code in the `InitProperties`, `ReadProperties`, and `WriteProperties` events of the `UserControl` to store and read the `ClientName` property in the `Text` property of the `TextBox`. The default value of the `ClientName` property will be a blank string:

```
Private Sub UserControl_InitProperties()
    txtClientName.Text = ""
End Sub

Private Sub
UserControl_ReadProperties(PropBag As
➥PropertyBag)
    txtClientName.Text =
➥PropBag.ReadProperty("ClientName", "")
End Sub

Private Sub UserControl_WriteProperties
➥(PropBag As PropertyBag)
    PropBag.WriteProperty "ClientName",
➥txtClientName.Text, ""
End Sub
```

6. Call `PropertyChanged` for `ClientName` in the `Change` event of the `TextBox`:

```
Private Sub txtClientName_Change()
    PropertyChanged ClientName
End Sub
```

7. Place your cursor in one of the Property procedures for `ClientName`, choose Tools, Procedure **A**ttributes from the VB menu, and click the Advanced button. In the Data Binding section of the resulting dialog box, click the boxes labeled Property is Data bound and This property binds to `DataField`, as shown in Figure 13.7.

APPLY YOUR KNOWLEDGE

8. Your new ActiveX control should now be ready to test as a Data consumer: add a standard EXE project to the Project Group. Add a Data Control to the form and set its `DataBaseName` property to point to one of the sample Microsoft Access databases provided with VB. Then set its `RecordSource` property to point to one of the tables in the sample database.
9. Now, making sure you've closed the designer for the `UserControl`, add an instance of your new control to the standard EXE's form.
10. Set the `DataSource` property of your control's instance to point to the Data Control you just added to the form. Set your control's `DataField` property to point to one of the fields provided by the Data Control's RecordSource.
11. Test your control by running the application and clicking the Data Control's navigation buttons. You should see the data in your control change as you click the navigation buttons.

13.6 Creating an ActiveX Control that is a Data Source

In this exercise, you create an ActiveX control that behaves like Microsoft's Data Control. You also create a simple EXE project to test its behavior. To complete this exercise, follow these steps:

Estimated Time: 60 minutes

1. Start a new ActiveX control project.
2. Make sure that the Project References include the Microsoft ActiveX Data Objects 2.0 library (use the Project, References dialog box from the VB menu to set this reference).
3. Set the `UserControl`'s `DataSourceBehavior` property to `1-vbDataSource`.
4. Create property procedures for two custom string properties: `Connect` and `RecordSource`. Create private variables to hold the values of each of the properties. Create private constants to hold their initial default values.

```
'Default Property Values:
Const m_def_RecordSource = ""
Const m_def_Connect = ""

'Property Variables:
Private m_RecordSource As String
Private m_Connect As String

Public Property Get RecordSource() As String
    RecordSource = m_RecordSource
End Property

Public Property Let RecordSource(ByVal
➥New_RecordSource As String)
    m_RecordSource = New_RecordSource
    PropertyChanged "RecordSource"
End Property

Public Property Get Connect() As String
    Connect = m_Connect
End Property

Public Property Let Connect(ByVal New_Connect
➥As String)
    m_Connect = New_Connect
    PropertyChanged "Connect"
End Property
```

5. Create a third custom property, `RecordSet`. Its type will be `ADODB.RecordSet`, and it will be read-only so you need to give it only a `Property Get` procedure. Declare a private variable `rs` to hold its value and make sure that `rs` can support events. Also, declare a private variable `cn` of type `ADODB.Connection`. It will not correspond to a custom property, but it's necessary in order to host the `ADODB.Recordset`.

```
Private WithEvents rs As ADODB.RecordSet
Private cn As ADODB.Connection

Public Property Get RecordSet() As
➥ADODB.RecordSet
   Set RecordSet = rs
End Property
```

APPLY YOUR KNOWLEDGE

6. Code the `InitProperties`, `ReadProperties`, and `WriteProperties` events to properly manage and persist the property values.

```
Private Sub UserControl_InitProperties()
    m_RecordSource = m_def_RecordSource
    m_Connect = m_def_Connect
End Sub

Private Sub
UserControl_ReadProperties(PropBag As
➥PropertyBag)
    m_RecordSource =
PropBag.ReadProperty("RecordSource",
➥m_def_RecordSource)
    m_Connect =
PropBag.ReadProperty("Connect",
➥m_def_Connect)
End Sub

Private Sub
UserControl_WriteProperties(PropBag As
PropertyBag)
    Call
PropBag.WriteProperty("RecordSource",
m_RecordSource, m_def_RecordSource)
    Call PropBag.WriteProperty("Connect",
m_Connect, m_def_Connect)
End Sub
```

7. Program the `UserControl`'s `GetDataMember` event procedure to initialize the ADO `Connection` object, `cn`, and the ADO Recordset object, `rs`. You will base the specific connection and Recordset on information contained in the private variables that implement the `Connect` and `RecordSource` properties. You should perform some error-trapping to ensure that these two properties will in fact yield a valid connection.

```
Private Sub
UserControl_GetDataMember(DataMember As
➥String, Data As Object)

    On Error GoTo GetDataMemberError

    ' rs and cn are Private variables of the
➥UserControl

    ' if this is the first time through, then
➥they haven't been set yet
    If rs Is Nothing Or cn Is Nothing Then

        ' make sure various properties have
➥been set
        If Trim$(m_Connect) = "" or
➥Trim$(m_RecordSource) = "" Then
            Beep
            Exit Sub
        End If

        ' Create a Connection object and
➥establish
        ' a connection.
        Set cn = New ADODB.Connection
        cn.ConnectionString = m_Connect
        cn.Open
        ' Create a RecordSet object.
        Set rs = New ADODB.RecordSet
        rs.Open m_RecordSource, cn,
➥adOpenKeyset, adLockPessimistic
        rs.MoveFirst
    End If

    Set Data = rs

    Exit Sub
    ➥
GetDataMemberError:

    MsgBox "Error: " & CStr(Err.Number) &
➥vbCrLf & vbCrLf & Err.Description,
➥vbOKOnly, Ambient.DisplayName
    Exit Sub
End Sub
```

8. Put code in the `UserControl`'s `Terminate` event that will gracefully close the data connection.

9. Add four command buttons to the surface of the `UserControl` designer, and name them `cmdFirst`, `cmdPrevious`, `cmdNext`, and `cmdLast`. You may give them captions such as "First," "Next," "Last," and "Previous," or you may give them more graphical captions, shown in Figure 13.18, so your control will look more like Microsoft's Data Control.

APPLY YOUR KNOWLEDGE

10. Put code in the `Click` events of the `CommandButtons` that will navigate the `Recordset` object, `rs`.
11. Your new ActiveX control should now be ready to test as a `DataSource`: Add a standard EXE project to the Project Group. Now, making sure you've closed the Designer for the `UserControl`, add an instance of your new control to the standard EXE's form.
12. Set the `Connect` property of your control's instance to point to your sample copy of `NorthWind.MDB`, and set the `RecordSource` property to point to one of the tables in `NorthWind`.
13. Add a `TextBox` control to the form and set its `DataSource` property to point to the instance of your control. Set its `DataField` property to point to a field in the `RecordSource`.
14. Run the application and test your control by clicking its navigation buttons. You should see the data in the `TextBox` control change as you click the navigation buttons.

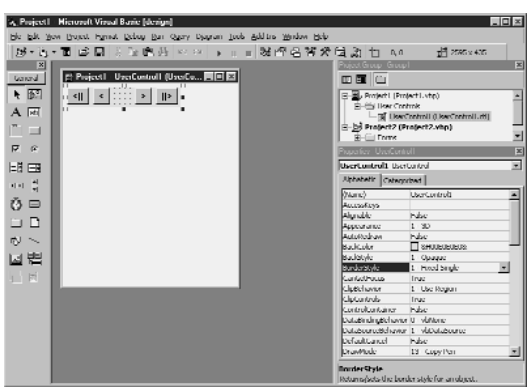


FIGURE 13.18
Your Data Source user control, complete with navigational interface for the end user.

Review Questions

1. Describe how and why raising events are necessary when creating ActiveX controls.
2. Describe the facilities provided in Visual Basic that allow a `UserControl` to load and write its properties.
3. How does the programmer prevent an ActiveX control from being visible at runtime?
4. Describe the role of the `PropertyBag` object.
5. What must you do to test your ActiveX control within Internet Explorer? What must you do to test its behavior in another VB program?
6. What is the name of the collection that provides the controls being modified by a Property Page?
7. What two things must you do to an ActiveX control project so that it will provide the programmer with `DataField` and `DataSource` properties?
8. When does the `GetDataMember` event fire?

Exam Questions

1. Give the code to define an event called `TopClick` with two parameters, x and y.

   A. You cannot define your own events in Visual Basic.

   B. `Event TopClick`(x as long, y as long).

   C. `Event Sub TopClick`(x as long, y as long).

   D. `Event Function` TopClick(x as long, y as long).

   E. `Sub TopClick`(x as long, y as long).

APPLY YOUR KNOWLEDGE

2. Your ActiveX control needs to fire an event called `TopClick` that has two parameters. The event is defined as follows:

```
Event TopClick(x as long, y as long)
```

The parameters x and y must be set to the value of 100 each. Which of the following statements will accomplish this task?

A. `Call Event TopClick`(100,100)

B. `FireEvent TopClick`(100,100)

C. `RaiseEvent TopClick`(100,100)

D. `Call TopClick`(100,100)

3. Your ActiveX control has a Property Page whose control `txtCaption` must display the current value of the first selected control's `Caption` property when the Property Page is opened. Which of the following code will provide this functionality?

A.
```
Private Sub PropertyPage_
➥SelectionChanged()
  Caption = txtCaption.Text
End Sub
```

B.
```
Private Sub PropertyPage_
➥SelectionChanged()
  SelectedControl.Caption
_txtCaption.Text
End Sub
```

C.
```
Private Sub PropertyPage_
➥SelectionChanged()
  Me.Caption = txtCaption.Text
End Sub
```

D.
```
Private Sub PropertyPage_
➥SelectionChanged()
  txtCaption.Text _
    = SelectedControls(0).Caption
End Sub
```

4. Which of the following statements will raise an event called `Click` with no parameters?

A. `RaiseEvent Click`

B. `Call UserControl_Click`

C. `RaiseEvent UserControl_Click`

D. `Raise Click`

5. Which of the following code snippets will load the `Caption` property of a control?

A.
```
Private Sub UserControl_
➥InitProperties()
  Caption = PropBag.ReadProperties
➥("Caption")
End Sub
```

B.
```
Private Sub UserControl_ReadProperties
➥(PropBag As PropertyBag)
  Caption  = PropBag.ReadProperties
➥("Caption")
End Sub
```

C.
```
Private Sub UserControl_
➥InitProperties()
Caption = _
PropBag.ReadProperties("Caption",m_def_
➥Caption)
End Sub
```

D.
```
Private Sub UserControl_Initialize()
  Caption = LoadProperty("Caption")
End Sub
```

APPLY YOUR KNOWLEDGE

6. You have created an ActiveX control. The control needs to be hidden from the user at runtime. Which of the following actions will ensure that the control is hidden at runtime?

   A. Setting the `Visible` property to `False` in the `Initialize` event of the control.

   B. Setting the `InvisibleAtRuntime` property to `True`.

   C. Call the `InvisibleNow` method of the ActiveX control.

   D. This cannot be done because ActiveX controls in Visual Basic are always visible.

7. Which of the following statements is true about ActiveX controls?

   A. ActiveX controls can only be used from Web pages.

   B. ActiveX controls cannot contain other controls.

   C. ActiveX controls must be visible at runtime.

   D. An ActiveX control can read or save its properties from a persistent location.

8. Which statement is the best description of a `Property Bag` object?

   A. The `Property Bag` is a collection of the ActiveX control properties.

   B. `Property Bag` is an object used to read and write properties to a persistent location.

   C. A `Property Bag` is an array of properties of an ActiveX control.

   D. `Property Bag` is a not a valid object type in Visual Basic.

9. Which statement best describes the `DataBindingBehavior` property?

   A. Setting it to `True` will allow the control to be a data consumer.

   B. Setting it to `vbSimpleBinding` will allow the control to be a data consumer.

   C. Setting it to `True` will allow the control to be a data source.

   D. Setting it to `vbSimpleBinding` will allow the control to be a data source.

10. Which statements are true of the `GetDataMember` event?

    A. You initiate a data connection in its event procedure.

    B. You set its first parameter to the control's data connection.

    C. It fires whenever a bound control needs to refresh its data.

    D. You set its second parameter to the control's Recordset.

11. How would you expose a DataSource control's Recordset so it can be programmed by other programmers using your control?

    A. Implement a string property called `RecordSource` and a string property called `ConnectString`.

    B. Return the `Recordset` in the second parameter of the `GetDataMember` event procedure.

    C. Initialize a property of type `Recordset` in the `GetDataMember` event procedure.

    D. Make sure the `UserControl`'s `DataSourceBehavior` property is set to `vbDataSource`.

APPLY YOUR KNOWLEDGE

12. In order to test an ActiveX control at design time, when should you make it the Startup project?

    A. Never.

    B. When you want to test it with an EXE project in the same Project Group.

    C. When you want to test it with a container application.

    D. Always.

13. When switching from your ActiveX control project to the EXE test project, you should

    A. run the ActiveX project before switching to the EXE test project.

    B. close the ActiveX project's Designer before switching to the EXE test project.

    C. make sure the ActiveX project's Designer is open before switching to the EXE test project.

    D. compile the ActiveX project before switching to the EXE test project.

Answers to Review Questions

1. Events are raised by controls by using the `RaiseEvent` statement. The `RaiseEvent` statement allows a control to fire an event that its container may respond to if something of interest occurs. If the user changes the `Text` property of an ActiveX control, for example, it may fire the `Changed` event to notify its container that the property has changed. See "Declaring and Raising Events."

2. The `UserControl` provides the developer with three events that help loading or writing properties. These events are the `InitProperties`, `WriteProperties`, and `ReadProperties`. The `ReadProperties` and `WriteProperties` events provide a `PropertyBag` object that is used to read or write property values. See "Implementing Property Persistence."

3. The developer of the ActiveX control can prevent the control from being visible at runtime by using the `InvisibleAtRuntime` property or by setting the `Top` or `Left` properties so that they are outside the visible portion of the screen. See "Implementing Property Persistence."

4. The `PropertyBag` object is used to read and write property from a persistent location provided by the container. The `PropertyBag` object is only accessible when the `ReadProperties` or the `WriteProperties` events are fired. See "Implementing Property Persistence" and "Using the Property Bag to Store Property Values."

5. To test your control with Internet Explorer, make sure that the Debug tab of the Project, Properties menu dialog box indicates that the control should be loaded automatically, and that it should be loaded in the Web browser. See "Testing Your ActiveX Control with Existing Container Applications."

6. The name of the collection that provides the controls being modified by a Property Page is `SelectedControls`. See "Determining which Controls are Selected for Editing with the `SelectedControls` Collection."

7. In order to enable an ActiveX control to be a data consumer (that is give it a `DataSource` and `DataField` property), you must set the `UserControl`'s `DataBindingBehavior` property and you must use the Tools, Procedure Attributes dialog box to specify a property that is bound to data and is associated with the `DataField` property. See "Providing `DataSource` and `DataField` Properties

APPLY YOUR KNOWLEDGE

with the Procedure Attributes Dialog Box."

8. The `UserControl`'s `GetDataMember` event fires whenever a data consumer sets its `DataSource` property to an instance of the ActiveX control. See "The `GetDataMember` Event."

Answers to Exam Questions

1. **B**. Events can be defined in Visual Basic. However, a `Sub` or `Function` statement is not allowed. For more information, see the section titled "Declaring and Raising Events."

2. **C**. Events can be raised by using the `RaiseEvent` statement. For more information, see the section titled "Declaring and Raising Events."

3. **D**. The `SelectedControls` collection of the Property Page will return all the current selected controls. In this case, the first item's caption is set. For more information, see the section titled "Determining Which Controls Are Selected for Editing with the `SelectedControls` Collection."

4. **A**. The `RaiseEvent` statement fires an event in the container. For more information, see the section titled "Declaring and Raising Events."

5. **B**. The `ReadProperties` event receives a `Property Bag` from the container used to read the property. For more information, see the section titled "Retrieving Persistent Property Values with the `ReadProperties` Event and the `ReadProperty` Method."

6. **B**. The `InvisibleAtRuntime` property is used to determine whether the control is hidden at runtime. For more information, see the section titled "Understanding the `UserControl` Module."

7. **D**. An ActiveX control can read and write its properties from a persistent location. ActiveX controls can be used from any container that supports an ActiveX control, such as Visual Basic, Powerbuilder, or Microsoft Internet Explorer. In addition, ActiveX controls can be visible or invisible at runtime. For more information, see the section titled "Implementing Property Persistence," "Testing and Debugging Your ActiveX Control," and "Accessing Ready-Made Control Features with the `UserControl`'s Extender Object."

8. **B**. The `Property Bag` is a simple interface that allows the ActiveX control to read and write its properties. For more information, see the section titled "Using the Property Bag to Store Property Values."

9. **B**. Setting the `UserControl`'s `DataBindingBehavior` property to `vbSimpleBinding` will allow the control to be a data consumer (that is it will show `DataSource` and `DataField` properties to the programmer). Its possible values are `vbNone`, `vbSimpleBinding`, and `vbComplexBinding`. This property does not have anything to do with whether or not a control is a Data Source. See "Providing `DataSource` and `DataField` Properties with the Procedure Attributes Dialog box."

10. **A, D**. The `GetDataMember` event is the place where you would initiate a data connection and recordset to provide to data consumers (bound controls). You do so by setting the second parameter to the initialized recordset. B is false because the first parameter is a String used to identify members of `DataBindings` collections. C is false

APPLY YOUR KNOWLEDGE

because the event only fires when a bound control's `DataSource` is set to this control. See "The `GetDataMember` Event."

11. **C**. To allow other programmers to program your ActiveX `DataSource` control's `Recordset`, you can expose it by implementing a property of type `Recordset` and initializing it in the `GetDataMember` event procedure. Each of the other answers describes valid things you do with a `UserControl` that is a `DataSource` but for other purposes:
A) Implementing `RecordSource` and `ConnectString` properties would give other programmers more control over how the Recordset is initialized, but these properties will not by themselves expose the Recordset nor are they necessary for exposing the Recordset; B) While it's true you must return the Recordset in the second parameter, doing so does not expose it automatically for programming; C) You must set the `UserControl`'s `DataSourceBehavior` property to `vbDataSource` in order to implement a `DataSource` control, but this does not implement an exposed Recordset by itself. See "The `GetDataMember` Event."

12. **C**. You should make your ActiveX control the Startup project when you want to test it with a container application. When you want to test it with an EXE project, make the EXE project the startup project. See "Testing Your ActiveX Control with Existing Container Applications" and "Testing and Debugging your ActiveX Control in a Test Project."

13. **B**. Before switching from an ActiveX control project to the EXE test project, you need to close the ActiveX control's Designer. Otherwise, the control will be disabled in the toolbox when you switch to the EXE project, and any instances of the control that you've already placed in the EXE will be disabled. See "Testing and Debugging your ActiveX Control in a Test Project."

OBJECTIVES

This chapter helps you prepare for the exam by covering the following objectives and their subobjectives:

Create an Active Document (70-175 and 70-176 exams).

- **Use code within an Active Document to interact with a container application.**
- **Navigate to other Active Documents.**

- The Active Document concept provides a document-centric approach to creating applications. An Active Document is a special type of COM component that is able to run inside a second application known as a container.
- The first subobjective for this objective addresses the fact that when you program an Active Document component, you need to provide for the Active Document's behavior inside its container. This behavior can vary depending on the type of container application.
- Your Active Document application also can interact with other Active Document instances that the container might have open at the moment. These other Active Document instances might include other instances of the current Active Document or instances of other Active Document types.

Use an Active Document to present information within a Web Browser (70-175 exam).

- The second major objective for this chapter refers to the most common type of container application that's currently used with an Active Document: namely a Web Browser (such as "Internet Explorer"). This objective deals mainly with how to embed the appropriate instructions in a Web page (HTML file) in order to display an instance of your Active Document on the page.

CHAPTER 14

Creating an Active Document

OUTLINE

Overview and Definition of Active Documents **687**

Steps to Implementing an Active Document **688**

Setting Up the `UserDocument` **689**

Converting an Existing Project to an Active Document 690
Creating an Active Document Project 690
Choosing Between an Active Document EXE and an Active Document DLL 691

Running Your Active Document in a Container Application **692**

Detecting the Type of Container with the `TypeName` Function and `UserDocument.Parent` 693

Managing the Events in Your Active Document's Lifetime **694**

`Initialize` Event 694
`InitProperties` Event 695
`EnterFocus` Event 695
`Show` Event 695
The `ReadProperties` Event and `ReadProperty` Method 696
The `WriteProperties` Event and the `WriteProperty` Method 697
`ExitFocus` Event 698
`Hide` Event 698
`Terminate` Event 698

Managing Active Document Scrolling **698**

The `Scrollbars` Property and `MinHeight` and `MinWidth` Properties 699
The `HScrollSmallChange` and `VScrollSmallChange` Properties 700
The `Scroll` Event Procedure and the `ContinuousScroll` Property 700

Managing the Active Document's ViewPort **701**

The ViewPort Coordinate Properties 701
`SetViewPort` Method 704

Defining Your Active Document's Custom Members **704**

Methods 705
Properties 705

Data and Property Persistence in Active Documents **706**

Saving Information in the .vbd File 706
Data Preservation Events and the Properties Bag 707

Asynchronous Download of Information **708**

Starting the Download With the `AsyncRead` Method 709
Stopping the Download With the `CancelAsyncRead` Method 710
Reacting to the Download Completion With the `AsyncReadComplete` Event 711

OUTLINE

Defining Your Active Document's Menus 712

Design Considerations for Active Document Menus 712

Negotiating With the Container's Menus 713

Merging Your Help Menu With the Container's Help Menu 714

Limitations of Modeless Forms in an Active Document Project 715

Navigating Between Documents in the Container Application 716

Using the Hyperlink Object With Internet-Aware Containers 716

Navigating the Container App's Object Model 718

Writing an Application to Handle Different Containers' Navigation Styles 718

Creating an ActiveX Project With Multiple `UserDocument` Objects 719

Testing Your Active Document in the VB Design Environment 722

Compiling and Distributing Your Active Document 724

Using Your Active Document on a Web Page 725

Chapter Summary 726

STUDY STRATEGIES

- Before reviewing this material, you should be familiar with how to create and use COM components such as ActiveX Controls and other types of components (see Chapter 12, "Creating a COM Component that Implements Business Rules or Logic" and Chapter 13, "Creating ActiveX controls").
- Create and experiment with a simple Active Document application that provides navigation between document instances (see Exercises 14.1 and 14.2).
- Create a simple HTML page and include a reference to your Active Document in the page.

INTRODUCTION

Programmers can use Active Documents to provide a variety of new services previously unavailable. Some of these services are summarized as follows:

- You can create applications that can execute within a container application such as Microsoft Internet Explorer or Microsoft Binder.
- The Active Document server will execute locally and have all the code needed to carry out its operations.
- Designing the GUI portion of Active Documents is easier than using HTML because of Visual Basic's WYSIWYG design environment.
- Built-in support of hyperlinks that allow the document to navigate a browser container to a specific document or Web site.
- Support for asynchronous data transfers.

The Active Document consists of two pieces: the Active Document (which contains the data) and the Active Document Server. The data in an Active Document cannot be viewed or edited without the Active Document Server present on the client machine. If the Active Document Server is not on the client, a different document server may view the Document contents if it can read the Document's data. This chapter covers the following topics:

- Using a `UserDocument` object, which is the basis of an Active Document, just as the `UserControl` is the basis for an ActiveX control.
- Understanding and implementing `UserDocument` lifetime events.
- Using the necessary techniques to persist property values.
- Understanding the relationship of an Active Document to various types of container applications, such as Internet Explorer or Office Binder, including the awareness of the container's ViewPort in the Active Document and the coexistence of container and document menus.
- Implementing custom Active Document properties and methods.

- Understanding asynchronous downloads of property information in the background while your document continues other processing.
- Knowing when you can use Modeless Forms in an Active Document.
- Writing code to navigate among container documents.
- Testing your Active Document project at design time.
- Compiling and distributing an Active Document including Internet distribution issues.

An Active Document application is a special type of ActiveX Component application that acts very much like a form. The Active Document, however, is unlike a form in that it integrates seamlessly into another special type of application known as an Active Document container.

Overview and Definition of Active Documents

The two most well-known Document container applications on the market today are Microsoft Office Binder and Internet Explorer (version 3.0 and above). However Microsoft's 97 Office Suite provides examples of important end-user applications that can serve as Active Document applications: You can already open a Word or Excel document in Internet Explorer without having to go through the Word or Excel programs. (Of course Word and Excel still need to be installed on the user's workstation for this to work!)

And, of course, the VB6 IDE can act as an Active Document container!

When users run a container application, they can open an Active Document component's data files directly from the container without having to think about the mechanics of the Active Document application itself.

NOTE

The Future of the Active Document Concept Although it might seem like an insignificant area because so few applications are able to act as Active Document containers, Microsoft is grooming the Active Document concept to be the future of operating system interfaces to application data. Windows 98's integration of Internet Browser and Windows Explorer is just one more step toward Microsoft's docu-centric environment of the future.

Superficially, then, an Active Document running in its container application might look like an exposed ActiveX server object in an OLE container. However, an Active Document exposes an entire application to its container. An OLE container, on the other hand, makes available only a piece of an ActiveX server application's data.

End users will no longer have to think about opening an application just to get to their data. The ActiveX container (the operating system's file interface) will use Active Document applications (such as the current versions of Word and Excel and, who knows, perhaps your applications) to offer a seamless user interface for Active Document applications' data files.

When you program your own Active Document applications, you will use many ActiveX concepts and techniques such as those discussed in the previous two chapters on ActiveX servers and ActiveX controls.

Steps to Implementing an Active Document

▶ Create an Active Document.

The following points give an overview of the general steps you need to take to set up your own Active Document application:

1. Initiate a VB ActiveX DLL or VB ActiveX EXE project based on one or more `UserDocument` objects. You might want to also convert an existing VB Standard EXE to an ActiveX project.
2. Create your own properties for your ActiveX project. Implementing these custom members is almost identical to the techniques for implementing custom members we discussed in Chapter 13, "Creating ActiveX Controls."
3. Provide for data persistence. This concept is very similar to the concept of property persistence as discussed in Chapter 13.
4. Design and implement your Active Document application's menu system. Besides creating the menu with the techniques discussed in Chapter 3, "Implementing Navigational Design," you will need to determine how your Active Document application's menus will coexist with the container application's menus.
5. Program specific Active Document features that make your Active Document aware of and able to react to its container.

6. Provide for the environments of specific container applications. To some extent an Active Document application must be aware of the specific object model of its container. Because different containers can have differing object models, you'll need to learn how to detect which container currently holds the Document. You'll need your application to act differently depending on which container it detects.
7. Provide document navigation. Different container applications' object models will provide differing techniques for moving between Active Documents. You must program effective document navigation for the different types of containers.
8. Test your Active Document application with possible containers. This usually involves running the container application while your Active Document application is in VB's design mode.
9. Compile and distribute your Active Document application bearing in mind special considerations for Internet distribution. Again, Active Document application compiling and distribution is quite similar to the compiling and distribution of ActiveX controls, as discussed in Chapter 13.

Each of the previously listed steps is discussed in the remaining sections of this chapter.

SETTING UP THE `UserDocument`

`UserDocument` objects are the basis of an Active Document application. A `UserDocument` bears the same relation to an Active Document as the `UserControl` object bears to an ActiveX Control project: It's the design basis for each component, providing built-in properties, events, and methods, and also providing a visual design surface for the component's finished appearance. In addition, you can add your own members to the finished component.

You can create an Active Document project with a new `UserDocument` object either by starting a new Active Document project (a `UserDocument` object automatically appears in the project) or by converting an existing VB Standard EXE project to an Active Document.

Both possibilities are discussed in the following sections.

Converting an Existing Project to an Active Document

Even though it's not a very automatic operation, it is possible to convert an existing VB application to a VB Active Document application. Follow these steps to perform the conversion:

1. Open the VB project whose Forms you want to convert to `UserDocument` objects.
2. Make sure you have the Active Document Migration Wizard loaded on VB's Add-Ins menu.
3. Run the Active Document Migration Wizard, which will convert all the forms that you select into `UserDocument` objects.
4. You are responsible for writing or defining any code such as `Form` event procedures that don't exist in a `UserDocument` object. In particular, you must make sure that converted procedures are compatible with the intended container applications in which the Active Document will be running. You may specify an option in the wizard to automatically comment out such code. For more specifics on container compatibility, see the section in this chapter titled "Running Your Active Document in a Container Application."

As stated in the following section, it's much easier (and therefore recommended) to create an Active Document from scratch rather than to use the techniques discussed in this section.

Creating an Active Document Project

Although it's possible to create an Active Document project by converting an existing VB project (as discussed in the previous section), you might have problems redefining all the event procedures and other code so that the application works properly. It's much easier to follow the steps described in this section and simply create a new Active Document project.

You create a new Active Document project with the same ease that you start any other type of project in VB:

STEP BY STEP

14.1 Creating a New Active Document Project

1. If VB is already open, choose File, New Project from the menu. Otherwise, Open VB and make sure you've chosen the New tab on the Project Open dialog.
2. Select either ActiveX Document DLL or ActiveX Document EXE from the icons of available project types. (See the following section for a discussion of how to choose the type of Active Document project).
3. In the VB IDE, you will now see a new UserDocument in its designer interface.

You are now ready to begin developing your Active Document project, as discussed in the rest of this chapter.

Choosing Between an Active Document EXE and an Active Document DLL

When you create an Active Document project, VB gives you the same two choices as when you create an ActiveX server: EXE or DLL. As discussed in Chapter 9, an EXE provides you with an out-of-process component while a DLL provides you with an in-process component.

When you are deciding between an ActiveX DLL document and an ActiveX EXE document, keep these differences in mind:

- An EXE supports both modeless and modal forms while DLLs only support modal forms.
- An EXE can run as a standalone application apart from a container application. A DLL must always show its data within a container application.
- A DLL offers faster performance.

In short, you must choose an EXE if

- You need to display modeless Forms from your Active Document application.
- You are concerned about keeping a completely separate address space for each instance of the Active Document application.
- You want to create an application similar to Microsoft Excel or Word that can also run as a stand-alone application. In other words you don't require that the user must use a container application to view your application's data.

If none of the above three conditions applies, then you should choose a DLL because it will give you better performance.

Running Your Active Document in a Container Application

Your Active Document application might need to behave differently depending on the type of container in which it runs.

For example, some containers (such as Internet Explorer) will be aware of the Internet and will handle Internet addresses properly while other containers such as Office Binder (at least as of Office 97) won't know URL from Earl. An Active Document that finds itself sited in an Internet-aware application can use its Hyperlink object to navigate to other documents. If the container is not aware of the Internet, then the Active Document must use other techniques specific to that container. (See the section "Navigating Between Documents in the Container Application.")

This difference in behavior will, of course, translate into different program code. Because you can't know ahead of time what type of container application will use your Active Document, you must programmatically detect your container's type and then write different code for each possible container type.

How to detect and react to the type of your `UserDocument`'s container is discussed in the following section.

Detecting the Type of Container with the `TypeName` Function and `UserDocument.Parent`

The `UserDocument`'s Parent property is an object that points to the container where the current instance of the Active Document is sited.

You can use this fact to get information about the container, including its application type. Recall that the `TypeName` function takes the name of a variable as its argument and returns a string telling the data type of the variable. Therefore, the following line of code in a `UserDocument` will display the container type for all to see:

```
MsgBox "Container is " & TypeName(UserDocument.Parent)
```

The example in Listing 14.1 will detect the type of container, store the container type in a `String` variable, and then take different actions depending on the contents of the `String`:

LISTING 14.1

DETECTING THE TYPE OF CONTAINER

```
Dim strContainerType as String
strContainerType = Ucase$(UserDocument.Parent)
'If container is Internet Explorer:
If Instr(strContainerType, "WEBBROWSER") <> 0 Then
    'behave one way
'or if it's Office Binder:
ElseIf strContainerType = "SECTION" Then
   'behave another
'or if it's VB:
ElseIf strContainerType = "WINDOW" Then
   'behave a third way
'or if the container type is unknown
Else
   'behave in a very generic fourth way
End If
```

Note that the string for the container type (as specified in the Document's Parent object) is not necessarily an intuitively obvious name for the container application.

For further discussion of this topic, see the section in this chapter titled "Writing an Application to Handle Different Containers' Navigation Styles."

Managing the Events in Your Active Document's Lifetime

The important events for Active Document event management are very similar to those of an ActiveX control. It is also important to note that the firing and sequencing of these events will not be the same for your Active Document in different container types. The important Active Document events are listed in the following sections followed by a discussion of the typical sequencing of events in an Active Document running under different container types.

The `Scroll` event is discussed in a separate section called "Managing Active Document Scrolling."

`Initialize` Event

The `Initialize` event is always the first event fired in a session with the Active Document regardless of container type. It fires when the document loads into its container.

You might use the `Initialize` event to set certain behavioral properties of the `UserDocument`, as in Listing 14.2.

LISTING 14.2

Coding the Initialize Event Procedure

```
Private Sub UserDocument_Initialize()
    UserDocument.ContinuousScroll = False
    UserDocument.HScrollSmallChange = 20
    UserDocument.VScrollSmallChange = 20
    UserDocument.MinHeight = 10000
    UserDocument.MinWidth = 10000
End Sub
```

Note that the `Initialize` event will typically fire more often under Internet Explorer than under Office Binder. This is because Internet Explorer will unload your document whenever the user has navigated to four other documents after loading your document (see the section called "`Terminate` Event"). Therefore, IE will need to reload your document if the user wishes to return to it, thus firing the `Initialize` event.

The call to URLFor in the example assumes that the project contains a custom WebItem known as SvcExcellent. The HTML returned to the browser would contain a URL computed for the WebItem that would work from the user's location. As far as the user's browser is concerned, the hyperlink would be a normal link implemented with the <A HREF=… tag format in HTML.

5. Double-click the new WebItem to add code to its Respond event procedure. The code that you put here will react to the user's choice in some manner. Perhaps the code will make an entry in a database on the server, perhaps it will send an email to someone in your organization, or perhaps it will send a new page to the user's browser by calling the WriteTemplate method of an HTML template. Listing 17.10 shows one such possible use of a WebItem's Respond event.

LISTING 17.10

USING A WEBITEM'S RESPOND EVENT TO SET SOME CLASS-LEVEL VARIABLES AND SEND A TEMPLATE TO THE USER, AND THE USE OF THOSE VARIABLES IN THE PROCESSTAG EVENT PROCEDURE OF THE TEMPLATE

continues

LISTING 17.10 *continued*

USING A WEBITEM'S RESPOND EVENT TO SET SOME CLASS-LEVEL VARIABLES AND SEND A TEMPLATE TO THE USER, AND THE USE OF THOSE VARIABLES IN THE PROCESSTAG EVENT PROCEDURE OF THE TEMPLATE

```
[illegible]
```

In the example of Listing 17.10, the Respond event sets some Private variables of the WebClass and then calls the [illegible] method of the HTML template. The [illegible] event procedure of the template then uses the variables to dynamically change the appearance of the Web page sent to the browser.

Custom Events for WebItems

Each custom WebItem that you define automatically gets its own [illegible] event, as discussed in the preceding section.

You can, however, define one or more events of your own for a WebItem. You can then embed calls to these events in the HTML code that you return to a browser by using a slightly different syntax for the [illegible] function from that discussed in the preceding section.

To implement a custom WebItem event, take the following steps:

STEP BY STEP

17.3 Implementing a Custom WebItem Event

1. After you have created a custom WebItem, right-click it in the IIS project and choose Add Custom Event from the resulting ShortCuts menu.

2. A new event will be added to the WebItem and will appear underneath the WebItem in the tree with a lightning-bolt icon. It will have a generic name, which you can change by right-clicking the event item and choosing the Rename option from the shortcut menu.

3. You can embed an invocation of the event in the HTML code that you return to the browser by using the URLFor function in the Start event procedure of the WebClass.

   This technique is similar to that discussed in step 4 in the preceding section. However, the syntax that you will use with the [illegible] function differs slightly from that of the preceding example. In the case of a custom event, you must pass the [illegible] function two arguments rather than just one. The first argument will still be the name of the WebItem, and the new second argument will be a string giving the name of the custom event.

   Listing 17.11 gives a fragment of code from a WebClass' [illegible] event procedure that uses the [illegible] function to embed a hyperlink to a WebItem's custom event,

LISTING 17.11

CODE IN WEBCLASS_START THAT EMBEDS A HYPERLINK FOR A CUSTOM EVENT OF A WEBITEM IN THE HTML RETURNED TO THE BROWSER

```
[illegible]
```

   The code assumes that a custom event named [illegible] belongs to a WebItem named [illegible].

4. You would use a similar technique to that discussed in step 5 in the preceding section to react to the user's activation of the event's hyperlink. Instead of writing code in the [illegible] event (as the preceding example shows), you would rather write code in the event procedure named for the custom event that you yourself had defined.

Dynamic Events for WebItems

You can get even more flexibility than either the [illegible] event of a WebItem or a custom WebItem event provides: You can program with dynamically defined WebItem events.

Dynamic WebItem event names are unknown at design time. The names of these events come from information that your code embeds in the HTML sent to the user's browser at runtime. You detect the firing of a custom event in the [illegible] event procedure of the WebItem. [illegible] receives one string parameter, [illegible] . [illegible] contains, of course, the dynamic name of the custom event.

To implement a dynamic event, follow these steps.

STEP BY STEP

17.4 Implementing a Dynamic Event

1. You can embed a two-argument call to [illegible] in the [illegible] event procedure, as in step 3 of the preceding section. Instead of hard-coding an *existing* event's name for the second argument to [illegible] , however, you can pass a string for which you have defined *no event* as the second argument. This string can be either a hard-coded literal string or a string variable. This technique will dynamically vary the name of the event that fires when the user clicks the associated hyperlink.
2. In the WebItem's [illegible] procedure, you can vary the reaction to the user's choice of the hyperlink based on the contents of the [illegible] parameter. In Listing 17.12, a [illegible] structure in the [illegible] procedure traps for various dynamically generated event names and reacts differently, according to the name of the event.

LISTING 17.12

TRAPPING DYNAMICALLY DEFINED EVENTS IN THE USEREVENT PROCEDURE

```
[illegible]

    [illegible]

        [illegible]
            [illegible]
            [illegible]
            [illegible]

        [illegible]
            [illegible]
            [illegible]
            [illegible]

        [illegible]
            [illegible]
            [illegible]
            [illegible]

        [illegible]
            [illegible]
            [illegible]
            [illegible]

        [illegible]
            [illegible]
            [illegible]
            [illegible]

    [illegible]

[illegible]
```

A common example of the power of dynamic events would be to generate an event whose name is based on the key field of a database record. Code in the [illegible] procedure would detect this event name representing the key field and use it to perform a lookup on the data. For a full discussion of this technique, see the Case Study for this chapter.

DHTML APPLICATIONS

Previous sections of this chapter covered the IIS application development project, which is a server-side Web development technology new with VB6.

This section turns to a second Web development technology that is also new with VB6, the DHTML project. Unlike an IIS application, an application created with DHTML runs client side, in the environment of the end user's Web browser.

A DHTML page is implemented as an ActiveX DLL and support files that are transferred to the machine where the user's browser resides.

The object model for DHTML pages includes the following:

- —Represents the browser and displays (but does not contain) the Document.
- —Represents the HTML page viewed in the browser and contains the and HTML elements.
- —Represents the DHTML runtime environment.
- —Individual elements of HTML functionality that reside on the HTML page. Such elements can either be defined by sets of HTML tags, or as HTML controls that you place on the DHTML Page Designer's surface from the ToolBox. In a VB DHTML application, programmers can manipulate HTML elements in much the same way that they manipulate control objects in standard VB applications.

Creating a Web Page With the DHTML Page Designer

To begin programming with a DHTML project, you need to take the following steps:

STEP BY STEP

17.5 Preparing to Program With a DHTML Project

1. In the VB IDE, create a new project whose type is DHTML Application.

2. By default, the new project will provide you with a Modules and a Designers folder. You will mostly work with the Designers folder when adding DHTML functionality.
3. The Designers folder by default contains a single designer. You can double-click the designer to bring up its window, which appears in two panes to the left of the Project Explorer.

The left-hand pane of the DHTML Page Designer gives a schematic tree view layout of the elements in the DHTML page. The Document object lies at the root, and under the Document object lie all the other objects that belong to the DHTML page as elements.

The right-hand pane of the DHTML Page Designer is a visual design surface for the page. You can type text and place objects from the ToolBox on this surface, as discussed in the following section.

Refer to Figures 17.7, 17.8, and 17.9 in Exercise 17.8 for illustrations of the DHTML Page Designer's panels.

Modifying a DHTML Web Page and Positioning Elements

To place text on a DHTML page, you can just position your cursor in the design-time pane (the right-hand pane) of the DHTML Page Designer and begin typing. As you type, you are actually defining the text of an HTML page that will be stored with the DHTML Designer.

Every time that you press the Enter key as you type, you will define a new *Paragraph* element in the underlying HTML. A Paragraph element is a section of HTML that is surrounded by a P ...P pair of tags. The basic function of the paragraph tags is to instruct the browser to set off the element from other elements (usually by adding an extra carriage return at the end of the displayed element).

The first element of the paragraph tag pair (the P) can contain a great deal of additional information about the formatting of the entire object, however. A complete Paragraph element might look like the following in the underlying HTML code:

The DHTML Page Designer acts like a Web authoring tool in that it enables you to just type text while it provides all the underlying formatting information for the Paragraph tags. You can use a word processor–like interface with toolbar items and menus to format the text and thus change the Paragraph tags, as well as inner HTML tags.

The inner HTML tags go inside the Paragraph tags and control formatting attributes of the text, such as boldface, italic, and underscore.

You can keep track of the paragraph objects that you place on the DHTML Designer's surface, as well as any internal formatting tags, by checking the left-hand pane of the DHTML Page Designer and opening the tree of objects under the [illegible] object.

You can add user interface objects to the DHTML page from the ToolBox, just as you do for a standard VB application. However, these objects are not the standard controls that you are used to from VB development. Instead, the objects in the DHTML application's ToolBox represent elements of the HTML interface. Placing such an object on the DHTML Page Designer surface creates HTML code to implement the object.

When you place a DHTML object on the designer's surface, you will position the object either in a *relative* or *absolute* position.

When you place an object in a relative position, it will show up attached to the currently selected [illegible] object. If other objects already belong to the current [illegible], the object will appear at the end of the list. Objects placed in relative positions will adjust their positions on the screen of the end user's machine according to the screen's physical capabilities.

When you place an object in an absolute position, it will always show up exactly as you positioned it on the designer's surface. Absolute position gives you more control over the exact placement of an object, but it also deprives your page of flexibility in adjusting to different environments.

DHTML Events

A DHTML application supports numerous events, many of them quite analogous to those of a standard VB application. In fact, the names of many DHTML events are formed from the names of the corresponding standard VB events, but prefixed by the word *on*. For example, some common DHTML events are as follows:

- onclick Corresponds to VB's [illegible] event.
- onmouseup Corresponds to VB's [illegible] event.
- onkeypress Corresponds to VB's [illegible] event.

Several other DHTML events correspond to VB standard events in functionality, but have different names:

- onfocus Corresponds to VB's [illegible] event.
- onblur Corresponds to VB's [illegible] event.

Two of the [illegible] object's events ([illegible] and [illegible]) are useful for managing critical moments in the lifetime of a DHTML page. Here is a brief account:

- Load Occurs during the loading of the page to the browser. If the loading is asynchronous, the [illegible] event occurs after the first HTML element is created. If the loading is synchronous, the [illegible] event occurs after the last HTML element is created.
- Unload Occurs during the unloading of the page from the browser, before any HTML elements have been destroyed.

Two other events, [illegible] and [illegible], are generally less useful than [illegible] and [illegible]. [illegible] happens before all objects are loaded on the page (and so you can't trust object references to work here). [illegible], on the other hand, happens after all objects have been unloaded (so it's too late to check object settings).

Navigating Between DHTML Pages and Persisting State

If you wish to programmatically navigate between DHTML pages in your application, you need to manipulate the [illegible] object. This is only logical because, as you will recall, [illegible] represents the browser—and the browser does the navigating between pages.

Assume, for instance, that you supply the user with a button to click. The button takes the user to another DHTML page furnished by your application. In the button's [illegible] event procedure, you might write the line:

[illegible]

This causes the browser to navigate to the indicated page. Note the name of the page in the string: It is formed from the project name and an underscore and the name of the page with an [illegible] extension. (This is the name of the temporary file created for the page on the user's browser.)

You may also want to persist data between calls to the same page, or to pass information between pages. Once again, you will call on the browser to help you with its Property Bag. You use the [illegible] method to write information to the Property Bag.

Suppose, for example, that you wanted to save information about the state of a particular element on the DHTML page, just before navigating to another page. You would make up an appropriate property name, and then you would store a string value to that property. The code for doing so might look like this:

[illegible]

In the example, the programmer has invented a property named [illegible] and stored the value [illegible] to this property. Note that the call to [illegible] required three arguments:

- [illegible] for which you are storing the property
- [illegible] of the property
- [illegible] of the property

Later in vthe same session, the page may be reloaded. If you want to retrieve the information that was stored for the [illegible] property, you could put a call to [illegible] in the [illegible] event of the [illegible] object:

[illegible]

By using [illegible] and [illegible] in tandem, you can cause information about a page to become persistent across multiple sessions.

Changing DHTML Element Attributes and Content

DHTML Web page elements have four runtime properties that enable you to manipulate the page's underlying HTML and thus the attributes and contents of the object as displayed on the Web page:

- [illegible] represents all the HTML tags and text used to define an object, including the *outer HTML tags* that define the object as an entity on the page, such as [illegible] ...[illegible] tags. The contents of the first member of the [illegible] ...[illegible] tag pair also includes important information such as extended formatting instructions for the object and, most importantly, the object's ID that you assigned in the DHTML Page Designer. If you replace [illegible] without repeating the ID, you will effectively destroy the object's ID, making it impossible to manipulate the object in code afterward during the same session.

- [illegible] is most useful if you want to examine all the HTML that goes into the definition of the object.

- [illegible] refers to *inner HTML tags* and text. In other words, [illegible] doesn't include the HTML tags and formatting instructions in the element's [illegible] pair. If you assign an element's [illegible] property, you will replace inner HTML tags and text inside the element's outer HTML tags with new inner tags and text.

- [illegible] represents the text inside all HTML tags. If you assign an element's [illegible] property, you will replace the inner HTML tags and the text inside the element's HTML tags with straight text. Existing outer HTML tags will be left untouched.

- [illegible] represents the text and any carriage returns supplied by the outer HTML tags.

To illustrate the use of these properties, assume that you have placed a Paragraph element with ID [illegible] on the Designer surface and typed as its text [illegible]. Further assume that you have used the DHTML Page Designer's toolbar to assign it a font of Courier, a font size of 4, and to make it appear in bold and italic. The four properties would then have the following characteristics:

- The [illegible] property would include all the HTML tags that identify the object, and might look like this:

  [illegible]

- The [illegible] property might look like this:

  [illegible]

- The [illegible] property would include the contents of [illegible], and a carriage return.
- The [illegible] property would include just this:

  [illegible]

You will find that [illegible] and [illegible] are the most useful properties of the four if you want to change attributes.

To follow the preceding example, the line of code

[illegible]

would destroy all existing inner tags and text and replace them with the simple text, [illegible].

The line of code

[illegible]

would destroy all existing inner tags and text and replace them with the HTML tags and text [illegible].

Note that if you attempted to assign HTML tags and text to the [illegible] property, the tags would end up displaying as literal text.

Changing DHTML Element Style

The [illegible] object is a property of all DHTML objects except for the [illegible] object. It contains over 80 properties that enable you to manipulate the appearance of an element. Following is a brief sampling of those 80+ properties:

- [illegible] is a string like the string used for the [illegible] property.
- [illegible] is a string whose possible values are “none,” “solid,” “double,” “groove,” “ridge,” “inset,” and “outset.”

- **color** is a string whose format is either one of the accepted color descriptions for the browser (examples would be "black," "red," "yellow," "slategray," and many others for an IE environment) or strings representing a hex number (examples would be "#FF0000" for pure red, "#00FF00" for pure green, "#0000FF" for pure blue, and "#FF00FF" for purple).
- **fontFamily** is a string giving the name of a valid system font group. Some possible values are "serif," "cursive," and "sans-serif."
- **visibility** is a string indicating visibility of the object, and its most common possible values are "hidden" and "visible" (a third value, "inherited," works if the object is the child of another object).

As you can see from the preceding brief list, Style properties having the same names or similar names to properties of standard VB controls usually don't work in exactly the same way. VB constants that you may be used to assigning to standard control properties won't work here. There is typically a set of strings that you must use instead, which you can learn from the online documentation.

CASE STUDY: DISPLAYING CUSTOMER SALES INFORMATION

NEEDS

Your company's salespeople in the field connect to your server over the Internet from their laptops. Each salesperson needs to see information on individual customer orders.

REQUIREMENTS

The usage scenario for your solution could contain the following items:

- The user will see an initial Web page that lists customers available in your company's Sales database. Each customer ID is displayed as a hyperlink.
- When the user chooses one of the customer ID hyperlinks, a second Web page shows a summary of all the customer's current orders.

DESIGN SPECIFICATIONS

Here is a general outline of how you can create an IIS project to implement this scenario:

1. Create an IIS project with two WebClasses, one for the Web page representing the customer list and one for the Web page giving a list of a single customer's sales orders.

CASE STUDY: DISPLAYING CUSTOMER SALES INFORMATION

2. In the customer list WebClass, open an ADO recordset that points to Customer data. Loop through the customer records, creating an HTML table entry for each customer that you want to display. As part of each HTML table entry, create a dynamic URL using the [illegible] function. The [illegible] function will point to a custom WebItem, [illegible], which you will create in the IIS page. The [illegible]'s second argument will be the Customer record's unique key, which will therefore define a dynamic event on the custom Web Item.
3. In the [illegible] procedure for the [illegible] WebItem, check the value of the [illegible] parameter. It should represent one of the Customer ID's chosen on a hyperlink from the table from the first WebClass' page. Use the Customer ID to open a second recordset of all the sales orders for the customer whose ID was chosen.
4. Call the [illegible] method of the second page. In the [illegible] event procedure for the second page, use the [illegible] method to write sales order details to the browser in the form of a dynamic HTML table.

CHAPTER SUMMARY

KEY TERMS

- ASP
- DHTML
- DHTML
- HTML
- IIS (Internet Information Server)
- IIS Application
- WebClass
- WebItem

This chapter covered the following topics:

‡ The creation of IIS (WebClass) applications

‡ An overview of ASP (Active Server Pages)

‡ The creation of DHTML applications

APPLY YOUR KNOWLEDGE

Exercises

17.1 Enabling Microsoft Internet Information Server for ASP and VB IIS Designer Applications with WebClasses

Although installing IIS and establishing a virtual directory for your Web server development are not required VB exam objectives, you won't be able to do any ASP or IIS application development (and therefore won't be able to do exercises 17.2 through 17.7) unless you have IIS in place on your NT Server.

> NOTE
>
> **Users of NT Workstation and Windows 95 and Above Can Use Personal Web Server** If you aren't developing in an NT Server environment, but are using NT Workstation or Windows 95 or above, you can use Personal Web Server to emulate the IIS environment of an NT Server.
>
> Also note that NT Option Pack as discussed below can be installed on Windows 95 and above.

Estimated Time (excluding time to obtain NT 4.0 Option Pack): 15 minutes

1. Make sure that you have installed NT Server 4.0 Option Pack on your NT Server machine. If you don't have the Option Pack, you can get it from the Microsoft Web site (URL may vary, so do a keyword search for the Option Pack). This step and the following step will possibly be different in the future after NT 5.0 is released.
2. On the server machine, select the program group of the Windows NT 4.0 Option Pack, the subgroup for Microsoft Internet Information Server, and finally the option for Internet Service Manager. This will run the Microsoft Management Console utility.
3. Add a virtual directory to your default Web site or to the other Web site that you plan to use for your development.
4. The alias that you give to the virtual directory will be the name of the directory that users must enter in their browsers under the Web site name to navigate to your production development files.
5. After entering an alias for the virtual directory, you need to specify a physical folder that exists on the server to associate with the virtual directory. This physical directory will be the location where you place your deployment files for the applications that you build in the following exercises. For purposes of the exercises, you can place the VB source code and other source files in the same directory as the deployment files.
6. When the New Virtual Directory wizard prompts you for access permissions on the virtual directory, you must specify at least Read and Script Access. If not, users will not be able to browser to and use your Web applications.

17.2 Creating a Simple ASP Page

This exercise is for those who are unfamiliar with ASP. In the exercise, you create a bare-bones ASP page.

Estimated Time: 20 minutes

1. Use Notepad or some other text editor to create and save a file named MYFIRST.ASP in the physical folder for the virtual Web directory that you created in Exercise 17.1. After you have finished, the contents of the file should look like this:

APPLY YOUR KNOWLEDGE

```
<HTML>
<BODY>
<%
    Dim
    Dim
    Dim
    Dim
    Dim
    Dim

    Set
    Set
    Set
    Set
        Set
            Set
        Set
            Set
        Set
            Set
            Set
        Set
            Set
    End
    Set
    Set
    Set
%>
</BODY>
</HTML>
```

2. The preceding code illustrates several basic facts about ASP pages:
   - Use the <% ...%> tag pair to demarcate ASP code in the HTML page. Everything outside of these tags in the file is interpreted as standard HTML.
   - VBScript doesn't use variable typing. All objects and variables are declared without the As clause and are therefore of type Variant.
   - The ASP environment has some object class types of its own, such as the Response object and the Server object illustrated here.
   - You can use the Write method of the Response object to create HTML in the final output of the page.
3. Open your Web browser and type in the URL to the file to display it, as illustrated at the beginning of this chapter in Figure 17.1.
4. If you want to, you can continue to experiment with the ASP code in this page by stopping the browser and editing the file.

17.3 Creating a Basic IIS Application With WebClasses

In this exercise, you begin an IIS application and become familiar in the IIS Designer environment. You should save this exercise, because you will build on it in the following exercise.

Estimated Time: 10 minutes

1. Begin a new VB IIS application.
2. Open the Code window for the WebClass' Start event procedure to examine it. Run the application, accepting the default choices on the Project, Properties Debugging tab (see Figure 17.2 earlier in this chapter). Examine the appearance of the page in your browser.
3. In your browser, view the HTML source code behind the browser display. Notice that the browser sees standard HTML code as generated by the Response.Write method in the WebClass' Start event procedure.
4. Return to Design mode by closing the browser and then stopping the application.
5. Customize the application's Start event procedure. Modify the text in the existing calls to Response.Write and add your own HTML items with new calls to Response.Write.

APPLY YOUR KNOWLEDGE

6. Use some of VB's processing power to help create the HTML output in the s event by inserting a line such as

```
'y
➥m
```

within the lines that write to the body of the HTML text. Your modifications to the s event procedure might look like those shown earlier in this chapter in Listing 17.5.

7. Run the project again to examine its appearance in your browser.

8. Make sure you save the project, because you will use it as a basis for the next two exercises.

17.4 Enhancing an IIS Application With an HTML Template and Substitution Tags

This exercise builds on the result of Exercise 17.3 and illustrates the use of substitution tags with an HTML template in an IIS application.

Estimated Time: 25 minutes

1. Start with the IIS project that you created in Exercise 17.3.

2. Use a text editor such as Notepad to create a simple standard HTML (extension .HTM) file in a different directory from the directory where your application's project files are stored. The contents of the file should look like Listing 17.6 shown earlier in this chapter. After you have finished creating the file, save the file with the name MYFIRSTIIS.HTM.

3. In the VB IIS project, make sure that the WebClass designer is open and right-click on HTML Template WebItems to bring up the shortcut menu, as shown in Figure 17.4.

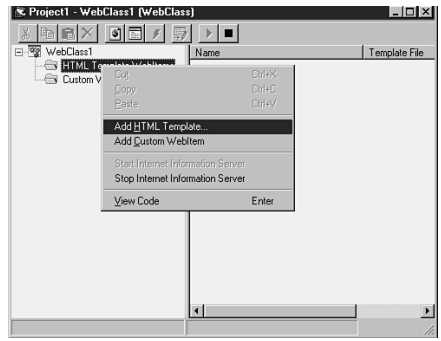


FIGURE 17.4▲
The HTML Template WebItems shortcut menu.

4. Choose Add HTML Template from the shortcut menu to bring up the File Browse dialog box, and then navigate to and select the *.HTM file that you created in step 2. A new item will appear in the Designer object tree under the HTML Template WebItems folder. Right-click the item and choose Rename from the shortcut menu. Rename the item as tmpMyFirst.

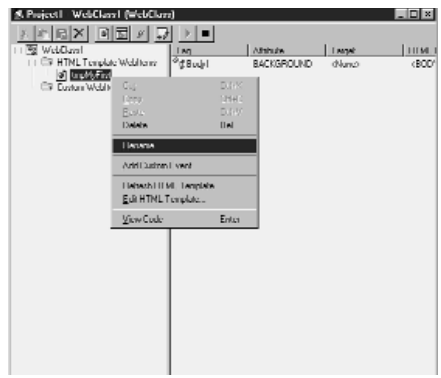


FIGURE 17.5▲
Renaming an HTML Template WebItem.

APPLY YOUR KNOWLEDGE

5. Navigate to the code for the [illegible] event procedure of the WebClass. Add the following line at the beginning of the [illegible] event procedure's code:

   [illegible]

   Rerun the application and observe the page's appearance in the Web browser. Stop the browser and the application in VB and experiment with different strings for background and font colors.

6. With the browser and the application stopped, right-click the template object in the Designer and choose Edit HTML Template from the shortcut menu. Change the contents of the HTML page to look like Listing 17.7 shown earlier in this chapter. Make sure that you save and close the text editor that you used for the HTML page before going on to the next step.

7. Enter code in the Template item's [illegible] procedure as follows. If you want to, substitute your own values for the three [illegible] values (the background and foreground colors and the greeting message).

   [illegible]

8. Run the application and view its changed appearance in the browser.

9. Stop the browser and application. Experiment by inventing and inserting new tag pairs in the HTML file and then adding references to those tags in the [illegible] event procedure.

17.5 Implementing Custom WebItem Respond Events

This exercise builds on the results of the preceding exercise to illustrate the use of the [illegible] event for custom WebItems.

Estimated Time: 40 minutes

1. Edit the HTML page for the HTML Template WebItem tmpMyFirst so that it looks like the following (changes are in italic):

   [illegible]

   ```
   <TITLE>SURVEY RESULT</TITLE>
   ```

   [illegible]

   ```
   <H1>SURVEY RESULT</H1>
   ```

   [illegible]

   Note that the only changes are the [illegible] and the addition toward the middle of the file of the header line with the text [illegible]. Make sure to save the file and close the editor after you have finished.

2. Right-click the Custom WebItems folder in the IIS designer window's left-hand pane and choose

APPLY YOUR KNOWLEDGE

Add Custom WebItem from the shortcut menu, as shown in Figure 17.6

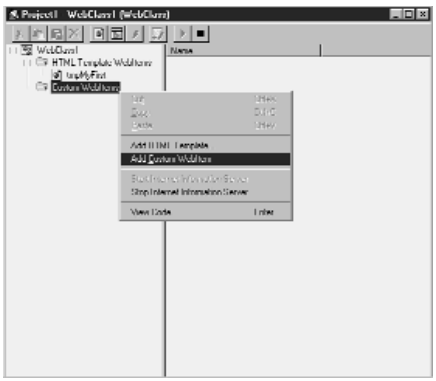


FIGURE 17.6
The Custom WebItems shortcut menu.

3. After adding the WebItem, type the name SvcAverage and press Enter.
4. Repeat Steps 2 and 3 four more times, adding WebItems named, respectively, SvcExcellent, SvcFair, SvcGood, and SvcPoor.
5. Completely change the event procedure so that it now reads as follows:

```
```

6. In the General Declarations section of the WebClass, declare some Classwide constants and variables, as follows:

```
```

7. Put code in the event procedures of the WebItems that you created in steps 3 and 4, as follows:

```
```

APPLY YOUR KNOWLEDGE

```
[illegible]
    [illegible]
    [illegible]
    [illegible]
[illegible]

[illegible]
    [illegible]
    [illegible]
    [illegible]
[illegible]
```

8. Modify the contents of the [illegible] event procedure as follows (modified lines are in italic):

```
[illegible]
    [illegible]
        [illegible]
        [illegible]
            [illegible]
            [illegible]
➥[illegible]
                TagContents =
➥strServiceDescription
            [illegible]
➥[illegible]
                [illegible]
➥[illegible]
            [illegible]
➥[illegible]
                TagContents = strServiceColor
        [illegible]
        [illegible]
[illegible]
```

Note that you modified the lines that set the tags for greeting and background color. The values of these tags will now derive from the Classwide variables and constants that you defined in step 6. The values of these variables, in turn, were set in one of the [illegible] event procedures that you programmed in step 7. Recall that each of the [illegible] event procedures calls the [illegible] method of the [illegible] object, and thus force the [illegible] to display and therefore also cause the [illegible] event procedure to run.

9. Run the application. When the initial page appears in the browser, select one of the choices that correspond to the custom WebItems. Note the behavior of the application. Press the browser's Back button to return to the initial page and make a different choice.

17.6 Implementing Custom Events for WebItems

This exercise builds on the results of the preceding exercise to illustrate how you can create custom events for WebItems.

Estimated Time: 40 minutes

1. Use the project of exercise 17.5 as the basis for this exercise.
2. Add a custom WebItem and name it Service.
3. Add five events to the WebItem (either right-click the new WebItem and choose Add Event, or choose the lightning-bolt icon from IIS Designer toolbar). Name the events Average, Excellent, Fair, Good, and Poor.
4. Write code in the event procedures as follows. Note that it is the same code that you wrote in the [illegible] event procedures of the individual WebItems in step 7 of exercise 17.5, so you can paste it from those event procedures if you like.

```
[illegible]
    [illegible]
    [illegible]
➥[illegible]
    [illegible]
[illegible]

[illegible]
    [illegible]
    [illegible]
➥[illegible]
    [illegible]
[illegible]
```

APPLY YOUR KNOWLEDGE

```
[illegible]
    [illegible]
    [illegible]
    [illegible]
[illegible]

[illegible]
    [illegible]
    [illegible]
    [illegible]
[illegible]

[illegible]
    [illegible]
    [illegible]
    [illegible]
[illegible]
```

5. Modify the WebClass' [illegible] event procedure as follows (changes are in italic):

```
[illegible]
    [illegible]
    [illegible]
        [illegible]
        [illegible]
        [illegible]
➥[illegible]
        [illegible]
➥[illegible]
        [illegible]
                    URLFor(Service,
➥ÒPoorÓ) & _
                    [illegible]
        [illegible]
                    URLFor(Service, ÒFairÓ)
➥& _
                    [illegible]
        [illegible]
                    URLFor(Service,
➥ÒAverageÓ) & _
                    [illegible]
        [illegible]
                    URLFor(Service, ÒGoodÓ)
➥& _
                    [illegible]
        [illegible]
                    URLFor(Service,
➥ÒExcellentÓ) & _
                    [illegible]
        [illegible]
        [illegible]
    [illegible]
[illegible]
```

Note that the difference from the preceding exercise consists in the fact that the calls to [illegible] now use two arguments rather one. The first argument is the name of the WebItem, [illegible]. The second argument is a string that tells which custom event of the WebItem to fire.

6. Run the application and note that it functions the same as before.

17.7 Implementing Dynamic WebItem Events

This exercise builds on the results of the preceding exercise to illustrate how you can implement completely dynamic WebItem events whose names are defined at runtime.

Estimated Time: 20 minutes

1. Use the project of exercise 17.6 as the basis for this exercise.
2. Delete the custom events for the [illegible] WebItem. Note that their event procedures' code still remains in the project under General Procedures.
3. Run the application and note that clicking the various anchors in the browser now yields blank screens for each item.
4. Add the following code to the [illegible] procedure of the [illegible] WebItem:

```
[illegible]
[illegible]
    [illegible]
        [illegible]
            [illegible]
            [illegible]
➥[illegible]
            [illegible]
```

APPLY YOUR KNOWLEDGE

```
[illegible]
    [illegible]
    [illegible]
➥[illegible]
    [illegible]

[illegible]
    [illegible]
    [illegible]
➥[illegible]
    [illegible]

[illegible]
    [illegible]
    [illegible]
➥[illegible]
    [illegible]

[illegible]
    [illegible]
    [illegible]
➥[illegible]
    [illegible]

[illegible]

[illegible]
```

Note again that the code in each [illegible] branch is the same as the code in the old event procedures.

5. If you run the application again, this time you will note that clicking on the items in the browser calls up the appropriate display. Although the events no longer exist in the project, the [illegible] procedure receives the customized event name formed by the calls to [illegible] in the [illegible] event procedure and can correctly detect which event has fired.

17.8 Creating and Modifying a Web Page with the DHTML Page Designer

In this exercise, you use VB's DHTML Page Designer to create a Web page.

Estimated Time: 45 minutes

1. Begin a new DHTML application project.
2. In the Project Explorer, open the Designers folder and double-click the [illegible] object to bring up its Designer.
3. In the right-hand pane of the Designer, type the text shown in Figure 17.7.

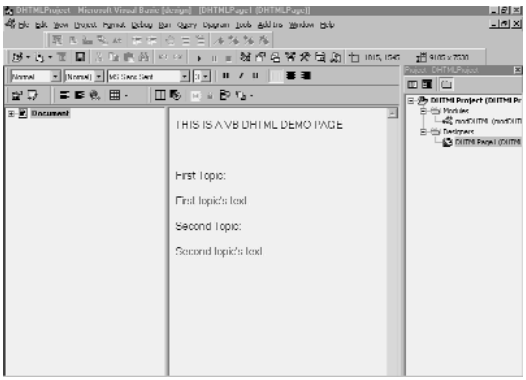


FIGURE 17.7
Text to type in DHTML Designer for Exercise 17.8.

4. Select the first line ("This is a VB DHTML Demo Page") and use the formatting tools at the top of the screen (see Figure 17.8) to assign a font size of 4 to the text and to make it appear bold.

   Assign a font size of 3 and bold to the lines that read "First Topic" and "Second Topic."

   Assign a font size of 2 to the lines that read "First Topic text" and "Second Topic text." Do not make them boldface.

5. Expand the left-hand pane of the Designer and fully expand all the nodes of the tree under the [illegible] object to examine their contents, as shown in Figure 17.9.

APPLY YOUR KNOWLEDGE

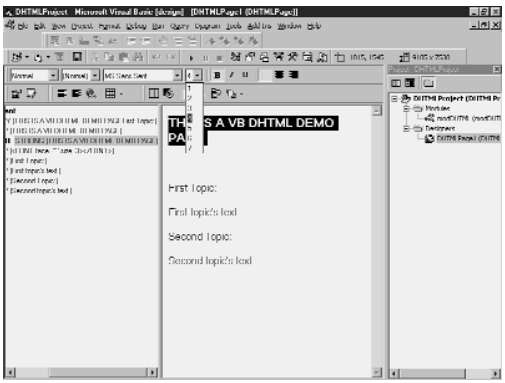


FIGURE 17.8▲
Using the formatting toolbar to assign attributes to text with the DHTML Page Designer.

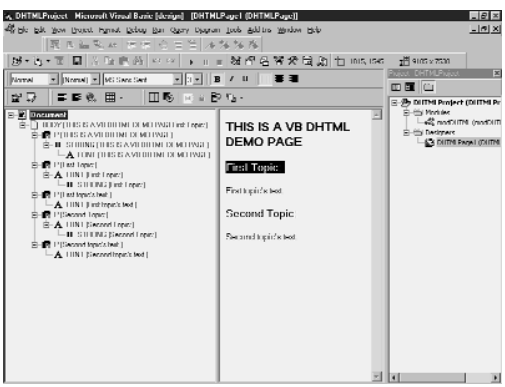


FIGURE 17.9▲
The objects contained in the Document of the DHTML page.

6. Assign a unique ID to the main title by selecting it, pressing F4 to bring up the Properties window, and typing M as the D property's value, as shown in Figure 17.10. Assign IDs to the other paragraph elements as follows:

   P , P ,

   P , P .

FIGURE 17.10▲
Assigning a unique ID to an element of the DHTML page.

7. Add two p objects and two f objects to the DHTML page, as shown in Figure 17.11.

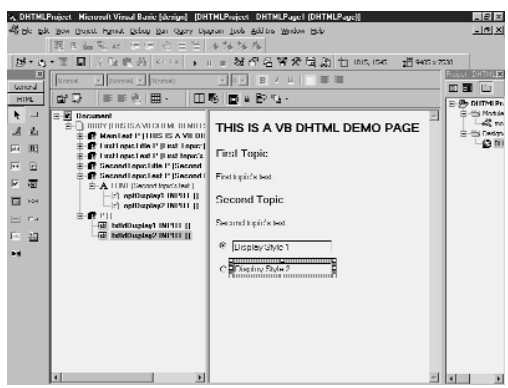


FIGURE 17.11▲
Adding elements from the Toolbox to the DHTML page.

8. Assign property values to the new elements as shown in Table 17.1. Note that the two p controls take the same name. Giving them the same name will include them in the same Option group and so will guarantee that only one Option in the group can be selected at any given time.

APPLY YOUR KNOWLEDGE

TABLE 17.1

VALUES TO ASSIGN TO CONTROL PROPERTIES IN EXERCISE 17.8

Control type	*Property*	*Value*
Option	ID	optDisplay1
	Name	optDisplay
	Checked	True
Option	ID	optDisplay2
	Name	optDisplay
	Checked	False
TextField	ID	txfldDisplay1
	ReadOnly	True
	Value	Display Style 2
TextField	ID	txfldDisplay2
	ReadOnly	True
	Value	Display Style 2

9. Write the following code in the DHTML page's event procedures:

```
[illegible]
    [illegible]
    [illegible]
[illegible]

[illegible]
[illegible]
    [illegible]
    [illegible]
[illegible]
    [illegible]
[illegible]

[illegible]
[illegible]
```

```
    [illegible]
    [illegible]
[illegible]
    [illegible]
[illegible]
```

10. Save and run the project. Accept the default Debugging options for the project (the project will run in your default browser). Click back and forth on the [illegible] controls and note the behavior. Note the lack of borders around the runtime instances of the [illegible] objects (caused by adjusting the [illegible] property in the [illegible] event procedure of the page). Experiment with the code and rerun.

17.9 Navigating Between Pages in a DHTML Project

In this exercise, you code a DHTML application to navigate between Web pages.

Estimated Time: 25 minutes

1. Use the same project that you created for Exercise 17.8.
2. From the VB ToolBox in the DHTML Designer, add a [illegible] control to the bottom of the form. Change the [illegible] property of the button to [illegible] and the [illegible] property to [illegible] (see Figure 17.12).
3. Double-click the button to view the Code window for its [illegible] event procedure. Enter the following code:

```
[illegible]
[illegible]
    [illegible]
[illegible]
    [illegible]
[illegible]
```

APPLY YOUR KNOWLEDGE

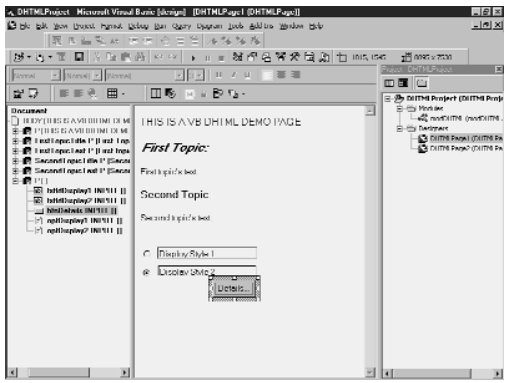


FIGURE 17.12▲
The DHTML form for Exercise 17.9 with the added Details button.

4. Add the following code to the event procedure (recall that the first two lines already exist from the preceding exercise):

5. Notice that the code of step 3 refers to a DHTML page that doesn't exist yet. Add the DHTML page from the Project menu and allow it to keep the default name of . Navigate to the new DHTML page.

6. Position your cursor in the right-hand pane of the DHTML Designer and press the Enter key twice to produce two Paragraph elements. Change the ID property of the first paragraph to and the second Paragraph's ID property to .

7. Select the second paragraph element and add a control to the page from the ToolBox (the new control will be included in the second paragraph element). Change the Button's ID property to and its to . The completed visual interface should look like Figure 17.13.

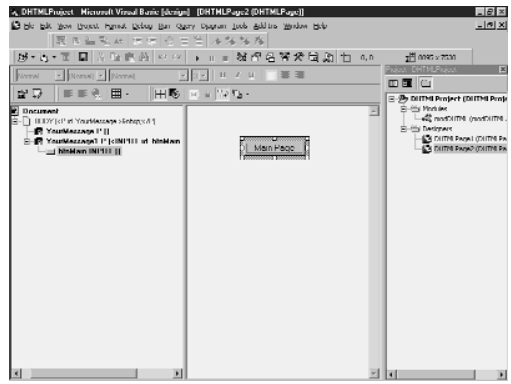


FIGURE 17.13▲
The second DHTML page for Exercise 17.9.

8. Enter the following code into the button's event procedure and the event procedure:

APPLY YOUR KNOWLEDGE

[illegible]

9. Test the application running it. In the browser, click the button to bring up the second page and observe the message. Click the button on the second page to return to the first page. Change the display state and repeat these actions. Note how state is preserved with the [illegible] and [illegible] functions.

Review Questions

1. Does an IIS (WebClass) application run on a Web client or on a Web server? A DHTML application?
2. What is the pair of tags that demarcate ASP code in a Web page?
3. What does the [illegible] event do?
4. Where can you detect the firing of a dynamically named WebItem event in an IIS application?
5. Which object member of a DHTML page element must you refer to in code to manipulate most of the element's visual formatting properties?

Exam Questions

1. Assuming that the default Tag ID for a WebClass is [illegible], the proper syntax for referring to a WebClass tag in an HTML template for that WebClass would be:

   A. [illegible]

   B. [illegible]

   C. [illegible]

   D. [illegible]

2. For an IIS (WebClass) application to run on an end-user's machine with an Internet browser, the following files must be deployed to the user's machine:

   A. MSWCRRUN.DLL, MVBVM60.DLL, and compiled application's DLL

   B. Application's DLL only

   C. MSWCRRUN.DLL and compiled application's DLL

   D. MSWCRRUN.DLL only

   E. No extra files

3. Calling the [illegible] method will possibly fire:

   A. The [illegible] event

   B. The [illegible] event

   C. The [illegible] event

   D. The [illegible] event

4. An HTML file that you use as an HTML template in an IIS application:

APPLY YOUR KNOWLEDGE

A. Should be created in a directory separate from the project's development directory and will be saved in the development directory.

B. Should be created in the development directory and then saved in a separate directory.

C. Should be created in and saved in a directory separate from the project's development directory.

D. Should be created in and saved in the project's development directory

5. You specify the substitution tag prefix for an [redacted] WebItem:

A. Within the [redacted] event

B. With the [redacted] property

C. Before the [redacted] section of the HTML template

D. Within the [redacted] section of the HTML template

E. Individually for each substitution tag that you use within the HTML template

6. The [redacted] event's first parameter specifies:

A. A delimited string containing the names of all the WebClass tags embedded in the HTML template's HTML code

B. An array containing the names of all the WebClass tags embedded in the HTML template's HTML code

C. A collection containing the names and values of all the WebClass tags embedded in the HTML template's HTML code

D. The name of a single tag

7. To enable the firing of a custom WebItem's [redacted] event:

A. In the [redacted] event of the WebClass, put code such as this:

[redacted]

B. In the [redacted] event of the WebClass, put code such as this:

[redacted]

C. In the [redacted] event of the WebClass, put code such as this:

[redacted]

D. Insert into the HTML template itself the HTML code:

[redacted]

E. Insert into the HTML template itself the HTML code:

[redacted]

8. The [redacted] procedure:

A. Runs for all events fired for WebItems.

B. Runs only for events defined by the programmer within the WebClass project.

C. Runs only for events whose names were generated on-the-fly in HTML code.

D. Runs for events defined by the programmer within the WebClass project and for events whose names were generated on-the-fly in HTML code, but not for any other events.

APPLY YOUR KNOWLEDGE

9. You can manipulate a background color of a DHTML page [redacted] element named [redacted] with the syntax:

   A. [redacted]

   B. [redacted]

   C. [redacted]

   D. [redacted]

   E. [redacted]

10. You want to manipulate HTML text within a DHTML page without disturbing its visible formatting. The segment of text that you want to manipulate has the name [redacted] and the ID [redacted]. You can change the text by referring to:

    A. [redacted]

    B. [redacted]

    C. [redacted]

    D. [redacted]

Answers to Review Questions

1. An IIS application runs on a Web server. The Web server invokes an instance of an IIS application to modify HTML pages that it sends to clients. Clients have no awareness of the IIS application. See "IIS (WebClass Designer) Applications in VB."

   A DHTML application runs on the same machine as the end-user's browser (a Web client). The DHTML application is in the form of an ActiveX DLL that downloads with a Web page to the browser. See "DHTML Applications."

2. The tag pair [redacted] demarcates ASP code within a Web page file. See "Creating a Simple ASP Page."

3. The [redacted] event fires each time that the IIS application encounters a pair of substitution tags in an HTML template. See "Substitution Tags."

4. The [redacted] event of a WebClass fires when a dynamically named event fires. The [redacted] procedure's parameter gives the name of the event that just fired. See "Dynamic Events for WebItems."

5. You often must refer to the [redacted] property of a DHTML page element to make visible formatting changes to a DHTML page element. For example, you must write the code

   [redacted]

   to change the [redacted] property of a [redacted] object. See "Changing DHTML Element Style."

Answers to Exam Questions

1. **C**. The proper syntax for referring to a WebClass tag in an HTML template is

   [redacted]

   assuming that [redacted] is the [redacted] for that Web Template item. You must use an HTML tag pair that includes the full name of the tag. The tag pair surrounds the default initial value of the tag. The tag name includes the Web Template item's [redacted]. For more information, see the section titled "Substitution Tags."

2. **E**. No extra files are needed on the user's machine for an IIS application. Recall that IIS applications run server side to help the Web browser prepare a standard HTML page. DHTML applications,

APPLY YOUR KNOWLEDGE

of course, are a different story; they implement an ActiveX DLL that runs client side. The files mentioned in the other choices are either required on the Web server only (MSWCRRUN.DLL and the application's compiled DLL) or not at all for an IIS application (VBVM60.DLL). For more information, see the section titled "IIS Applications in VB."

3. **D**. Calling the [illegible] method of an HTML template page in an IIS application will possibly cause the [illegible] event of the [illegible] object to fire. Note, however, that [illegible] would only fire in some cases, and then only as an indirect result of the [illegible] method (when the Web server reads the tags in the template). A is incorrect because the [illegible] event has already fired (you often place a call to [illegible] in the [illegible] event procedure). B is incorrect because, once again, [illegible] fires earlier. C is incorrect because the [illegible] event procedure is the other main place (aside from the [illegible] event) from where you would call the [illegible] method. For more information, see the sections titled "Programming with an HTML Template" and "Creating and Programming Custom WebItems."

4. **A**. An HTML file that you use as an HTML template in an IIS application should be created in a directory separate from the project's development directory. When you run, save, or compile the project for the first time after adding the template to the project, VB makes a copy of the template in the project's directory (the directory where the project's VBP file resides). For more information, see the section titled "Programming with an HTML Template."

5. **B**. You use the [illegible] property to specify the substitution tag prefix for an [illegible] WebItem. For more information, see the section titled "Substitution Tags."

6. **D**. The [illegible] event's first parameter specifies the name of a single tag. The fundamental problem with the assumption behind the three incorrect answers (besides the fact that they are just plain wrong) is that in reality [illegible] fires separately for each substitution tag pair, and so only handles one tag at a time. For more information, see the section titled "Substitution Tags."

7. **A**. To enable the firing of a custom WebItem's [illegible] event, you can place you can put code such as

```
[illegible]
```

in the [illegible] event of the WebClass. For more information, see the section titled "Creating and Programming Custom WebItems."

8. **C**. The [illegible] procedure runs only for events whose names were generated on-the-fly in HTML code. You can create such calls with syntax such as

```
[illegible]
[illegible]
```

where [illegible] is a variable name that will cause the firing of a like-named event. You then can write event-handling code to detect the custom event names in the [illegible] procedure of the WebItem.

APPLY YOUR KNOWLEDGE

A, B, and D are incorrect because events defined by the programmer within the WebClass project get their own event procedures (and so [illegible] doesn't fire), and the WebItem's default event (which you can implement by calling [illegible] with only one argument) is the [illegible] event. For more information, see the sections titled "Creating and Programming Custom WebItems," "Custom Events for WebItems," and "Dynamic Events for WebItems."

9. **A**. You can manipulate a background color of a DHTML page [illegible] element named [illegible] with the syntax [illegible]. The exact names of properties in such syntax can be a bit tricky to remember because the property names for DHTML control objects are not always the same as the corresponding standard VB control property names. For more information, see the section titled "Changing DHTML Web Page Element Style."

10. **A**. You can refer to the [illegible] property of a standard HTML object to change its displayed text in a DHTML application without affecting the visible format. Answer B ([illegible]) would perform the text replacement, but it would possibly change the visible formatting of the element, because it would replace the HTML tags surrounding the element. Answer C is syntactically incorrect, because this property's name in DHTML applications is not [illegible]. Answer D is syntactically incorrect, because the property's name is not [illegible], and moreover the [illegible] object does not give access to the text of standard HTML objects. For more information, see the section titled "Changing DHTML Web Page Element Content."

OBJECTIVES

The Debug window and the associated Debug object are VB's main built-in features for debugging your code at design time.

This chapter discusses many features of both the Debug window and the Debug object here, although only three of them—Watch expressions and the Immediate and Locals windows—are mentioned in the exam objectives. It is likely that the certification exam will require knowledge of some of VB's other built-in debugging features as well; that is why they are included here.

This chapter helps you prepare for the exam by covering the following objectives:

Set Watch expressions during program execution (70-175 and 70-176).

. Watch expressions enable you to monitor the changing values of data in your application.

Monitor the values of expressions and variables by using the Immediate window (70-175 and 70-176).

- **Use the Immediate window to check or change values.**
- **Explain the purpose and usage for the Locals window.**

. This chapter discusses the Immediate window, which enables you to interrogate and manipulate your application's environment while it is paused during design-time execution. This chapter also discusses the Locals window, which enables you to monitor and set the values of variables that are local to the currently running procedure.

Given a scenario, define the scope of a Watch variable (70-175 and 70-176).

. Watch scope enables you to refine exactly what information you monitor and when you monitor it.

CHAPTER 18

Using VB's Debug/Watch Facilities

OUTLINE

Preventing Bugs **846**

Using Watch Expressions and Contexts **848**

Creating a Watch Expression 849
Types of Watch Expression 850
Watch Contexts 852

Using Break Mode **854**

Entering Break Mode Manually 854
Stepping Through Your Code 855
Using the Watch Window 859
Entering Break Mode Dynamically 861

Using Quick Watch **863**

Watching on Demand **864**

Immediate Window and the Debug Object **864**

Displaying the Debug Window 864
Displaying Messages Programmatically with the Debug Object 865

Using the Print Method **866**

Formatting Debug.Print Messages 867
Displaying Data Values 869

Using the Debug.Assert Method **871**

Interacting with the Immediate Window **873**

Querying or Modifying Data Values 874
Testing and Executing VB Procedures 875

Using the Locals Window **877**

Using the Immediate Window in Place of Breakpoints **880**

Using the MouseDown and KeyDown Events 880
Using the GotFocus and LostFocus Events 880

Levels of Scope **881**

Local Scope 882
Module Scope 883
Global Scope 884

Scope Considerations **885**

Striving to Narrow the Scope 885
Performance Concerns 886

Chapter Summary **887**

STUDY STRATEGIES

- The best way to prepare for the exam topics discussed in this chapter is to complete the exercises at the end of the chapter.
- Although Exercises 18.1, "Using the Call Stack," and 18.2, "Using the `Debug.Print` Command," don't deal with explicit exam objectives, they do illustrate important VB debugging skills that you will need to be familiar with.
- Exercise 18.3, "Modifying Values in the Immediate and Locals Windows," addresses the exam objective for `Watch` variables and its subobjectives for Immediate and Locals windows.
- Exercise 18.4 illustrates the Watch Scope objective.

INTRODUCTION

Bugs are a fact of life. Even the most finely honed programming discipline can't guarantee error-free code. Because you can't absolutely prevent bugs from being written in the first place, you have to settle for second best by trying to stamp them out before they inadvertently get released into a product.

Fortunately, VB gives you a lot of opportunities to find your bugs before you put your software into the hands of your users. The primary debugging tools in the Visual Basic programming environment are the following three debugging windows:

- The Watch window
- The Immediate window
- The Locals window

This chapter covers the following topics:

- Bug prevention
- Watch expressions, types, and contexts
- Break mode
- The Immediate window and the `Debug` object
- Interacting with the Immediate window
- Locals
- When to use the Immediate window in place of breakpoints
- Levels of scope
- Scope considerations

PREVENTING BUGS

This discussion starts by testing your debugging skills by looking at a simple problem in a loop that assigns values to an array of string variables. The array is defined in a module as follows:

```
Public astrName(9) as String
```

Because arrays are zero-based by default, this creates an array of 10 variables in the astrName array. Assume that each of the variables is properly assigned a value, but that you later try to display the names in the array using this code:

```
Dim i as Integer
For i = 1 to 10
    MsgBox ÒName #Ó & i  & Ò is Ò & astrName(i)
Next
```

As this loop executes, you begin to see a series of message boxes for Name #1, Name #2, and so on. When you reach Name #10, however, a Òsubscript out of rangeÓ error message appears (assuming, that is, that you haven't used an advanced optimization feature to disable array bounds checking). Obviously, something is wrong, but what is it? After all, you know there are supposed to be 10 variables in that array, and you are only asking to see items 1 to 10. This hardly seems like it should give you any trouble.

This is a simple example, and if you already understand arrays, you can spot the problem without bothering with a watch. The array does have 10 elements, but because it is zero-based, the index values range from 0 to 9, not 1 to 10. When the loop tried to display the value of astrName(10), it was outside the bounds of the array.

The same person who wrote the buggy display loop might also make this mistake in assigning values to the array:

```
For i = 1 to 9
    astrName(i) = ÒSome name valueÓ
Next
```

This is a little more insidious than the display loop. This assignment loop doesn't generate any error messages, but it is wrong nonetheless. If the default Option Base 0 has not been changed, the first element of the array is astrName(0), to which this loop never assigns a value.

A programmer with some VB experience will understand these problems almost at once, but that doesn't help the beginner who wrote the buggy code in these loops. You can use certain techniques to help prevent these problems by making your intent clearer to others who may later need to maintain your code. Even if no one else but you will ever touch your code, you may benefit from these techniques yourself if you ever have to resume a project that you haven't touched in a long while.

First, the array might have been defined by explicitly specifying its upper and lower bounds:

```
Public astrName(0 to 9) as String
```

This makes it clear that the range of values begins with 0 and ends with 9, so it should be less likely that the flawed assignment loop above will be repeated. It also ensures that there will still be 10 elements in the array even if someone later adds Option Base 1 to the code to change the default lower boundary of an array.

Second, additional care could be taken in the display loop by finding out both the upper and lower boundaries of the array before trying to access its elements:

```
Dim iLower as Integer, iUpper as Integer
iLower = LBound(astrName)
iUpper = UBound(astrName)
For i = iLower to iUpper
    MsgBox ÒName #Ó & i  & Ò is Ò & astrName(i)
Next
```

This code is guaranteed to access each element of the array from beginning to end. Nothing gets skipped, and the code can't step outside the bounds of the array. Remember that you will need to use this technique for safety's sake if you use the advanced optimization switch that disables automatic array checking (see Chapter 20, "Compiling a VB Application").

You can (and you should!) take preventive measures like these to guard against potential bugs. Even simple safeguards have great value. For instance, it is hard for working VB programmers to imagine coding without being forced to declare variables via `Option Explicit`, for example, and `On Error Resume Next` is reserved only for extremely brief routines with their own internal error checks. When you run into a new problem that isn't so obvious, and you can't understand what's wrong just by reading the code, you will develop an instant appreciation for VB's debugging aids. The first type you will examine is the Watch expression.

Using Watch Expressions and Contexts

. Set Watch expressions during program execution.

A *Watch expression* is essentially what its name implies; it is an expression whose value you want to watch change while your program runs. The expression may be simple or complex.

A simple expression might consist of a single variable. A complex expression might perform calculations on multiple variables, or even call functions. The only requirement is that a Watch expression must be a legal VB expression. In other words, if you can't use an expression in a line of code without generating a compiler error, you can't use it as a Watch expression either.

Creating a Watch Expression

A watch enables you to observe the value of an expression while your code executes. To create a Watch expression, open a project and pull down the Debug menu to choose Add Watch, as shown in Figure 18.1.

FIGURE 18.1▲
Use the Debug menu to create a Watch expression.

After you click on the Add Watch option, you will see the dialog box shown in Figure 18.2.

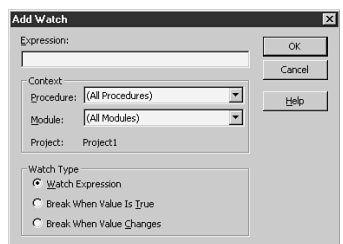
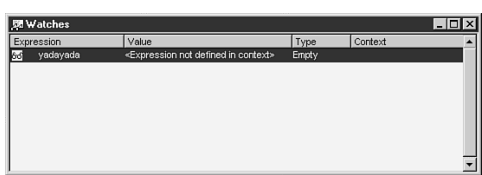


FIGURE 18.2▲
The Add Watch dialog box.

When you create a watch, you can specify any valid VB expression. The expression may range in complexity from the name of a single variable to a calculation involving a series of nested function calls. As long as it is a legal VB expression, you can monitor its value while your code executes.

If you make a mistake when you type the expression, however, VB won't tell you about it. There is no Watch expression equivalent of the VB syntax checker to tell you that you have made a mistake after you enter a line. Consequently, it is up to you to pay close attention to ensure that you type what you intended. If you type nonsense that VB can't evaluate at runtime, the Watch window will cheerfully display it for you in the Watch window without complaint, as shown in Figure 18.3.



, FIGURE 18.3
If your expression can't be evaluated, the Watches window says so.

The closest thing you will get to an error message is the text in the Value column of the Watch window that says Expression not deÞned in context , but you will see that message under normal circumstances anyway. That is because the mechanism that enables you to set watches isn't omniscient. The same rules that apply to standard VB code also govern the evaluation of a Watch expression. That

means that the expression being watched must be in the scope of the currently executing code for the Watch window to report its value.

That is why you don't get an error message in the Watch window even if you try to set a watch for nonsense. The Watch expression is evaluated only during runtime when VB is in Debug mode. You can, however, enter Watch expressions when VB is in Design mode.

You don't need to type the expression you want to watch. You can double-click on a variable name to select it, and then drag it to the Watch window. Likewise, if you highlight an expression, the expression can be dragged to the Watch window. Whatever you drag into the Watch window is automatically used to create a new Watch expression.

If you need to edit an existing watch to fix a mistake, select it in the Watch window, and then pull down the Debug menu to choose Edit Watch. You can also right-click the Watch window to add, edit, or delete a watch. Another way to delete a watch is to select it in the Watch window and press the Delete key.

As usual, VB gives you a lot of different ways to manage the Watch window itself. You can let the Watch window float or drag it to a location on your screen to dock in a fixed position. Grab its title bar with the mouse to drag it from one place to another. Double-clicking the title bar toggles its status between docked and floating.

Types of Watch Expression

Visual Basic offers three Watch types. Each Watch type determines a slightly different behavior for the Watch expression when you run your code at design time.

Remember that you choose the Watch type when you add or edit a Watch expression. The following three sections discuss these types of Watch expression.

Watch Type Watch Expression

The default Watch type is Watch expression. This Watch type will just list the expression's name and value (when in scope) in the Watch window. Whenever you want to know an expression's current value, you can access the Watch window with View, Watch Window.

Watch Type Break When Value Changes

If you select the Break When Value Changes Watch type in the Add Watch dialog box, Visual Basic will enter Break mode whenever the value of the expression changes during execution.

A Watch type of Break When Value Changes automates the process of monitoring the variable. When you set up a watch of this type, the Watch window (with a row listing the variable's current value) will come up automatically every time the variable's value changes.

Watch Type Break When Value Is True

This Watch type causes Visual Basic to enter Break mode when the value of the expression becomes True during execution. This type is handy when you're uncertain about the exact behavior of the expression and there is complicated code involved.

Unless the variable you're monitoring is of Boolean type, you typically won't stipulate a variable's name by itself in the Expression field of the Add Watch dialog box. Instead, you will put some type of comparison expression involving the variable, because what you're trying to do here is refine how often VB will break to the Watch window.

For example, your code includes a variable that is changed inside a loop. You are not, however, interested in having your code break every time the variable's value changes. You need only to know when it takes on a certain range of values, say values greater than 100. In that case, you might specify the following expression:

variablename > 100

Now, instead of being interrupted on every pass through the loop, VB notifies you only when the expression evaluates to True .

Of course, observing the expression's current value (which will be either True or False) in the Watch window doesn't give you much information about the variable's precise value. Therefore, you will typically use this type of watch in tandem with a simple nonbreaking watch—that is, a Watch type of Watch expression.

As an example, imagine a loop with this line:

```
intAccumulator = intAccumulator + SquareInt(intCounter)
```

You want to make sure that intAccumulator never approaches a value that could max out the carrying capacity of the Integer type, that is 32KB. To test this, you could create two Watch expressions:

- The first Watch expression's type would be Watch expression and the expression would be just the name of the variable, `intAccumulator`.
- The second Watch expression would be of the type Break When Expression Is `True`, and the expression would be something like `intAccumulator >=15000`. (If you waited for exactly 32KB, it would be too late!)

Therefore, when you run your application from the VB design environment with these two watches set, you will see nothing unless `intAccumulator` reaches 30,000 or more. If that ever happens, VB will go into Break mode, the Watch window will come up, and you will see the exact value on a line in the Watch window.

Watch Contexts

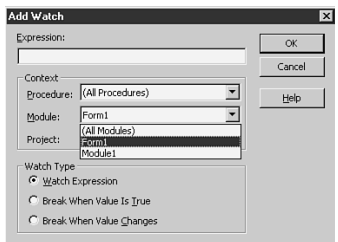
After you have entered an expression to watch, you need to tell VB about the context in which the watch should be active. Remember that the watch expression must be in the scope of the currently executing code for VB to tell you its value. If you specify a context in which the expression isn't valid, you will see the `Expression not defined in context` message. Again, that message doesn't necessarily mean that you have entered a bad expression. It does pay, however, to check the entry just in case.

If you don't have a form or module open when you create your watch, the context options for your watch defaults to all procedures and all modules.

If you accept the default context options, the expression is evaluated constantly throughout the entire project. Such a broad setting may sometimes make sense (as for a global variable, for instance), but you will make VB work harder if you monitor Watch expressions in every possible context. The narrower the scope, the faster you see the results.

Bear in mind that the goal of a watch is to locate a problem in your code, such as a failure to modify a variable or assigning it a bad value.

If you have some idea of where the problem is, it makes sense to limit the watch to a more appropriate context. When you set a watch, VB will change the module context setting for you to default to the form or module you are currently viewing, as shown in Figure 18.4.



, FIGURE 18.4
You can limit the scope of your Watch expression to a single module.

The default procedure selection is also context sensitive. If the cursor is in a particular procedure, it will be used as the default scope, as shown in Figure 18.5.

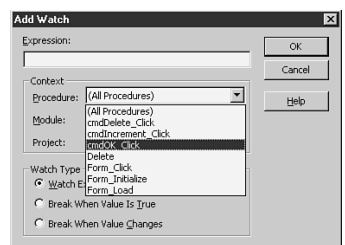
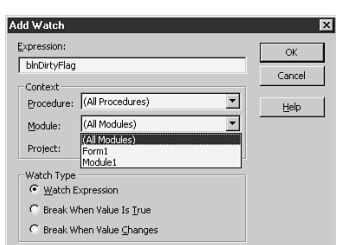


FIGURE 18.5▲
You can also limit the scope of a Watch expression to a particular procedure.

If you have an expression selected in a code window, the value of the Watch expression defaults to it too. If you want to watch a variable, you don't need to highlight its entire name—the word in which the cursor appears will be used by default. Of course, you aren't locked into the defaults. You can type a different expression for the watch if the default isn't what you want. If you need to select a different scope, you can select from a list of the modules and procedures in the current project, as shown in Figure 18.6.



, FIGURE 18.6
Use the combo boxes to change the module or procedure scope of a Watch expression.

Using Break Mode

You can't actually use the Watch window until you are in Break mode. Break mode is sort of a programmatic limbo: Your program is still active, but its execution is temporarily suspended while the programmer pokes around. All your variables retain their values, as do the properties of any objects in your project. During Break mode, you can inspect any of these values. Remember, however, that only those variables within the scope of the currently executing code are available at any given moment.

Entering Break Mode Manually

You can manually enter Break mode in several ways:

- Press Ctrl+Break
- Choose Break from the Run menu
- Click on the Break button on the toolbar

If you aren't sure what part of your code may be responsible for a problem, but you see it when it happens, the ability to manually enter Break mode can be handy. Because it is possible for so many Windows events to occur in rapid succession, however, the few hundred milliseconds it takes for you to click your mouse or press a key after you spot the problem may put you into a part of your code that has nothing to do with the problem.

If you can localize your problem to some extent, you can also specify places in your code at which to enter Break mode automatically:

- Set a breakpoint
- Use the Stop command

You can toggle a breakpoint either from the Debug menu, as shown in Figure 18.7, or by pressing the F9 key. If the current line has no breakpoint set, F9 sets one; if the line already has a breakpoint, F9 turns it off. If you want to remove all breakpoints from your project, you can either use the Debug menu to do so or press Ctrl+Shift+F9. The color of the line in the VB code editor reflects the status of any line with a breakpoint.

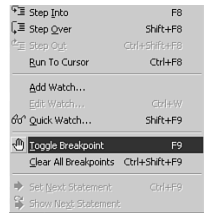


FIGURE 18.7
Setting a breakpoint from the Debug menu.

Once set, your program will enter Break mode the moment that it reaches the line of code with the breakpoint. The program's execution is suspended just before that line of code executes.

You can also toggle individual breakpoints on or off by clicking in the bar to the right of a line of code in Code window.

One problem with breakpoints is that they are not preserved between programming sessions. That is, if you close a project after setting a breakpoint and then reopen the project, your breakpoints will be gone. If you want to set a breakpoint that persists between sessions, use the Stop command. The moment the Stop line is reached in your code, the program enters Break mode.

Stepping Through Your Code

Setting breakpoints usually isn't an exact science. While tracking down a bug, you may observe that problems occur after clicking on a certain command button. That doesn't necessarily mean that the `Click` event code in the command button is at fault, of course. You don't have to wait to pin down the problem more closely before setting a breakpoint, however. Because you know that the problem occurs at some point after that procedure executes, go ahead and set a breakpoint there so that you are at least that much closer to the problem when you are in Break mode.

WARNING

Avoiding Inadvertant Stop Commands If you're worried about inadvertently leaving a Stop command in a project before you deliver it to a client, remember that you can use a conditional compiler switch:

```
#Const DEBUGMODE = 1
#If DEBUGMODE Then
    Stop
#End If
```

To be extra safe, set the value of `DEBUGMODE` on the command line. Then you won't have to worry about forgetting to remove the definition of the compiler constant from your code either.

Because program execution is suspended during Break mode, you may wonder how this helps you get closer to the bug. After all, you haven't reached the bug yet, and the program has stopped. How do you get to the problem?

Besides enabling you to watch variable and property values during Break mode, VB also enables you to continue to execute the program on an incremental basis. The process is typically called *stepping through* your code, and it enables you to execute a program a line at a time. You may not be right at the point at which a problem occurs, but you can step through your code to sneak up on the bug and catch it in the act.

The following four stepping commands are available in Break mode:

- Step Into
- Step Over
- Step Out
- Step to Cursor

FIGURE 18.8
The Debug toolbar.

The stepping commands are available in several ways: from the Debug menu, from the keyboard via the F8 key (by itself, or in conjunction with the Shift, Control, and Alt keys), or from the Debug toolbar—as shown in Figure 18.8. If the Debug toolbar is not already visible, you can pull down the View menu and choose Toolbars, where you can toggle it off and on. The Debug toolbar can be docked or permitted to float.

Here is a simple example to illustrate how the step commands behave after a program is in Break mode. Consider a project containing one form with a single command button. The code looks like this:

```
Sub cmdCommand_Click()
    Dim strName as String
    Msgbox ÒAfter this message, break mode begins.Ó
    Stop
    strName = MyFunction()
End Sub
Function MyFunction() as String
    Dim strPrompt as String
    strPrompt = ÒPlease enter your nameÓ
    MyFunction = InputBox (strPrompt)
End Function
```

When you run this project and click on the command button, the program enters Break mode when it reaches the Stop command. The code editor will appear with the Stop command highlighted in the cmdCommand_Click function. Program execution is suspended, and nothing else happens until you issue a step command.

Step Into

If you select Step Into, the highlight moves to the next line of code, where strName is assigned the value returned by MyFunction. If you select Step Into again, the code editor will display the code for MyFunction with the highlight in its first line (recall that the highlighted line hasn't executed yet; it runs after you perform the next stepping action). If you continue to select Step Into, you will see where the command gets its name: Step Into enables you to step into any code that is called in your project, executing every procedure a line at a time.

Step Over

Step Over is similar to Step Into. Within the current procedure, Step Over continues to execute the code one line at a time. The difference is how it handles calls to other procedures. If you run this project again and select Step Over when you reach the strName = MyFunction() line in the Click event, you won't step into the MyFunction code. Instead, it runs all at once and you are returned to the function that called it. When you aren't interested in observing the behavior of another procedure called from within the one you are debugging, choose Step Over.

Step Out

What if you selected Step Into, only to realize that you really don't need to watch the execution of every step of the procedure you just stepped into? No, you don't have to lean on the F8 key to force the execution of every line. That's what Step Out is for; it causes the rest of the current procedure to run and returns to the calling procedure. That is, you could select Step Out as soon as you entered MyFunction, and you would find yourself back in the Click event just after MyFunction had completed its work.

Step to Cursor

You may step into a procedure that only has a few lines that you want to watch execute. If that happens, and there are a lot of lines from your point of entry to the lines that interest you, you don't need to wade through the boring parts with Step Into or Step Over. Instead, move the cursor to the first line you want to watch, and then select Step to Cursor. All the code will execute between the line at which you originally entered Break mode and the line where you placed the cursor, then you can begin to step through the interesting parts again.

Setting Stepping Options

You will also notice a couple of other options at the bottom of the Debug menu. If you want to prevent a few lines of code from executing, but otherwise let your program continue to execute normally, use Set Next Statement.

While in Break mode, this enables you to move the cursor to the next line that you want your program to execute. Because this skips the intervening lines, use this feature carefully. If you inadvertently skip some lines that modify values in your program, you may not catch the bug you are after. If you aren't sure what line is supposed to execute next, use Show Next Statement .

The Set Next statement option only works with lines of code that are inside the currently running procedure. In other words, you can't set the next statement to a line of code in a different procedure from the currently running procedure.

To exit Break mode and resume the normal operation of your program, choose Continue from the Debug menu or toolbar. (Its keyboard equivalent is F5.) If you have seen all that you need to see, you can also halt the program by choosing End, which is available on the Debug menu or on the toolbar.

Using the Watch Window

While you are busy stepping through your code, you can watch the values of your Watch expressions in the Watch window. For instance, consider a loop such as this:

```
Dim i as Integer, j as Integer, k as Integer
Dim astrAlphabet(0 to 25) as String
j = LBound(astrAlphabet)
k = UBound(astrAlphabet)
For i = j to k
    astrAlphabet (i) = Chr$( i + 65)
Next
```

This loop populates a 26-element array with the letters of the alphabet. If you set a breakpoint and create a watch on the astrAlphabet array, you can watch the values of each element of the array as they are assigned values in the loop. However, you won't see the array build a string such as ÒABCDEFGHIJKLMNOPQRSTUVWXYZÓ. In fact, if you only watch the single line containing the name of the array, you won't see much of anything useful except for the range of the array index values (in this case, 0 to 25). That's because an array is a data structure that contains more than one value, and each line in the Watch window can only display an individual value.

Watching Arrays

Don't worry. You don't need to set a separate watch for each element of the array. When dealing with a data structure such as an array, the Watch window will display the name of the array with a boxed plus sign next to it, as shown in Figure 18.9. It works in the same way that Windows Explorer displays a disk drive's directory structure in a tree control. Just as Windows Explorer uses plus signs to indicate that nested subdirectories remain to be displayed, the VB Watch window uses plus signs to indicate that more data elements are nested within the selected structure.

FIGURE 18.9.
When a watch is set on an array, you will see a boxed plus sign next to it.

If you click the plus sign, the structure unfolds to display each individual data element, as shown in Figure 18.10. The plus sign changes to a minus sign, which you can click on to hide the elements contained in the structure again.

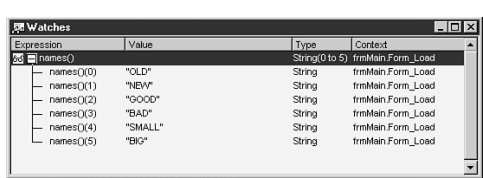


FIGURE 18.10.
Click on the plus sign to monitor the values of the elements contained in the array.

Watching User-Defined Types

The same nesting principle applies to user-defined types. If you create a type to store an employee's name, ID number, and Social Security number, it might look like this:

```
Type tEmployee
    strName as String
    iIDNumber as Integer
    lSocSecNum as Long
End Type
```

If you create a variable of type tEmployee , and then set a watch on that variable, the Watch window will display the name of your variable and a plus sign. As with the array, you need to click on the plus sign to unfold the strName , iIDNumber , and lSocSecNum elements of the data structure.

It is also possible to have multiple levels of nested structures in the Watch window. If you were to create an array of tEmployee variables called atMyEmployees , for example, you would need to click on the plus sign associated with the atMyEmployees array to display the tEmployee elements. To see the values contained for each tEmployee , you would in turn have to click on the plus sign associated with each element in the array.

Entering Break Mode Dynamically

If you're waiting to see where a watch value changes, the process of manually executing each individual line of your code can get to be awfully tedious, particularly if you're executing a lengthy loop. Waiting for the 26 letters of the alphabet to be assigned is no big deal, but what if you're processing hundreds or thousands of records? Rather than forcing you to manually execute the code until the change occurs (or until the F8 key on your keyboards wears out), VB gives you a couple of more convenient options.

Breaking on True

One of your options is to set a watch that causes your program to enter Break mode only when the value of the watch is True . This kind of watch spares you the bother of manually stepping through each individual line of code, taking you directly to Break mode when the program gets to a point that is really interesting.

Here's a simple procedure to illustrate the point:

LISTING 18.1

AN ILLUSTRATIVE PROGRAM

```
Private Sub Form_Click()
    Dim iCounter As Integer, iTotal As Integer
    Dim fIncrementTotal As Boolean
    Print ÒFrom 1 to 100, the even multiples of 7 are:Ó
    For iCounter = 1 To 100
            Ô set ßag if value is an even multiple of 7
            fIncrementTotal = Not CBool(iCounter Mod 7)
            If fIncrementTotal Then
                Print iCounter
                Ô sum even multiples of 7
                iTotal = iTotal + iCounter
            End If
        Next
    Print ÒThe sum of these values is Ò & iTotal
End Sub
```

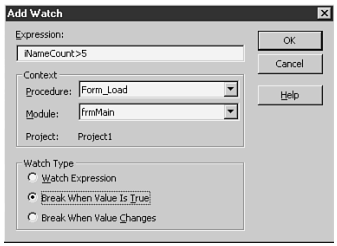


FIGURE 18.11
Select Break When Value Is True to enter Break mode automatically when a value is True .

If you set a simple watch on the iTotal variable, you would have to set a breakpoint and then manually execute every subsequent line to observe changes in iTotal 's value. If you instead set a watch that causes the program to enter Break mode only when fIncrementTotal is true, however, you will save yourself some drudgery. To set this kind of watch, just select the appropriate option in the Watch dialog box, as shown in Figure 18.11.

When you hit the first breakpoint, you can inspect the values of the program to make sure your calculations are correct. Instead of stepping through each individual line, you can select Continue and the program will run normally until the Watch expression evaluates to True again, at which point you are back in Break mode.

Breaking on Change

Another way to evaluate this procedure is to set a watch so that the program enters Break mode only when the value of a Watch expression changes, as shown in Figure 18.12. In this case, you could set this type of watch on the iTotal variable. The program would only enter Break mode when iTotal changes. This accomplishes the same thing as the preceding example, of course, because iTotal is incremented only when fIncrementTotal is True .

When you are working with different kinds of watches simultaneously, VB makes it easy to see what kind of watches are in play by displaying a different icon for each type in the Watch window, as shown in Figure 18.13. An eyeglasses icon denotes a simple Watch expression.

Break When Value Is True and Break When Value Changes both use an open palm to indicate that these watches use VB's internal breakpoint traffic cop to force your program into Break mode. The Break on Change icon also includes a small triangle, which you may also recognize as the Greek letter "delta," that is often used to indicate changes in a value. Break on True displays a small blue rectangle in addition to the open palm, which is perhaps meant to call the expression "true blue" to mind.

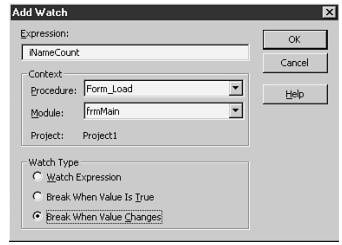
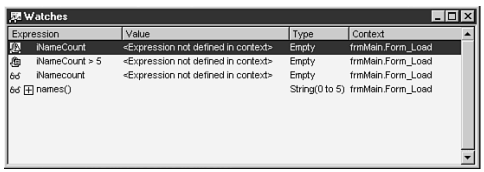


FIGURE 18.12▲
Select Break When Value Changes to enter Break mode automatically when a value changes.



, FIGURE 18.13
Each type of watch is associated with its own icon in the Watches window.

If you find that setting a watch that enters Break mode is more convenient than using the simple Watch expression you have already entered, remember that you can always change a watch from one type to another via the Debug menu's Edit Watch selection. A right-click of the mouse in the Watch window also displays this option on the pop-up menu.

USING QUICK WATCH

When you are in Break mode, you may want to determine the value of an expression in your code for which a watch has not already been set. You can use the Quick Watch feature to display the value of any expression, as shown in Figure 18.14. Select the expression, and then pull down the Debug menu and choose Quick Watch. This displays the value of the selected expression in a Quick Watch dialog box. The dialog box also gives you a chance to add the expression to the Watch window.

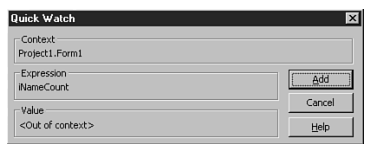


FIGURE 18.14▲
The Quick Watch dialog box.

WATCHING ON DEMAND

Finally, you can also evaluate any expression in Break mode by selecting it and positioning the mouse cursor over the expression for a moment. The expression and its value are displayed like the ToolTips that appear when you momentarily hold the mouse pointer over a toolbar button. You don't get a chance to add the expression to the Watch window this way, but you can take advantage of this technique to minimize clutter in your Watch window. If you don't need to constantly monitor a variable, just point at it when you need to know its current value.

IMMEDIATE WINDOW AND THE DEBUG OBJECT

. Monitor the values of expressions and variables by using the Debug window.

The Immediate window is so called because it gives the VB developer an opportunity to interact with a program during Break mode. By displaying almost instantaneous responses to your questions, you get a degree of immediate gratification that isn't available in many contemporary Windows development environments.

Displaying the Debug Window

To open the Immediate window, pull down the VB View menu and choose Immediate Window. The Immediate window will appear.

The Immediate window enables programmers to do the following three things:

- Display messages programmatically with the Debug object
- Query or modify data values *on-the-fly*
- Test and execute VB procedures

NOTE

Displaying the Immediate Window
Unlike some options that can be turned on and off via the same command, be aware that neither the menu option nor the shortcut serve as a toggle for the Immediate window; they can only be used to display the Immediate window when it is not currently displayed.

The first two capabilities are only available when a program is in Break mode. Messages can be programmatically displayed in the Immediate window by embedding certain statements in a program. This entails use of a built-in VB object called the Debug object.

Unlike the fixed response in the other windows that can be used to monitor the state of variables and properties, the Immediate window also permits programmers to interactively inquire about these values, or even to change a value at will.

The last capability doesn't require that a program be running in Break mode, or even that a VB program be running at all. At design time, the Immediate window can be used to test code or execute commands without actually running the application under development. If you want to find out what value a function returns, for example, you can enter and run the function in the Immediate window.

The following sections take a closer look at each of these capabilities.

Displaying Messages Programmatically with the Debug Object

If you want to display messages about the status of your program during a debug session, but you don't want to enter Break mode, you can do so by calling the Debug object and its methods in your code. The Debug object is built in to VB itself, so you don't need to declare an object variable or do anything special to use it.

The Debug object has two methods: Print and Assert . Each displays information in the Immediate window, but the methods have different applications.

USING THE PRINT METHOD

Like any other VB method, you must use the dot operator syntax to call the method from its object. Because you don't need to declare an object variable to use the Debug object, you just invoke the method from the object itself, like this:

```
Debug.Print ÒEat at JoeÕs.Ó
```

If this statement were in your code, the message ÒEat at JoeÕs.Ó would appear in the Immediate window when this line executed in your code. More useful messages might tell you that a particular function has been entered, or into which branch of a conditional test your code has entered. For instance:

LISTING 18.2

MORE USEFUL MESSAGES

```
Select Case iConditionTest
    Case 1
        Debug.Print ÒBranched into Case 1Ó
        Ô real case 1 code follows
        Case 2
        Debug.Print ÒBranched into Case 2Ó
        Ô real case 2 code follows
    Ô Case etc.
End Select
```

Although you could also find out this kind of information by single-stepping through your code in Break mode and observing the path of execution, using the Immediate window to display *signpost* messages saves you the bother of manually stepping through the code.
It is especially handy when all you wanted to know was the information displayed in the signpost itself. If you just want to know whether your code branches into condition A rather than condition B, and the details of execution are otherwise unimportant, a message in the Immediate window is much more convenient than single-stepping.

Because this is a debugging technique, you may recall a lesson from the chapter on conditional compilation that explains how to keep your debug code from inadvertently making its way into a product release. One way to prevent your Debug.Print statements from appearing in the release version of your programs might be to wrap in a conditional compiler block like this:

```
#CONST DebugConstant = 1
#If DebugConstant Then
    Debug.Print ÒThis message only appears when
➥DebugConstant is TrueÓ
    #End If
```

Fortunately, Microsoft did everyone a favor that saves the bother of writing a conditional compiler block every time you want to use Debug.Print. Statements involving the Debug object are effectively stripped out of your program when it is compiled.

> **NOTE**
> **When Debug Messages Appear** The Debug.Print message appears only when testing an application in the debugging environment. When a user runs an application, it is not running in the debugging environment, so there is no Immediate window to display the message.

Formatting Debug.Print Messages

The Debug.Print examples so far have been quite simple—Debug.Print followed by a message statement—but the syntax of the message can be somewhat more elaborate than simple text.

For one thing, the position in which the message appears in the Immediate window can be specified by preceding the message text with VB's Spc() or Tab() functions. If you want to indent a message by 10 spaces, you could do so in either of two ways:

```
Debug.Print Spc(10) ÒThis message is preceded by ten
➥spaces.Ó
```

or:

```
Debug.Print Tab(11) ÒThis message begins on column eleven,
➥which is functionally identical.Ó
```

Obviously, Spc() inserts spaces in the output, and Tab() sets the position where the next message will appear.

It is also possible to use more than one text expression on the same line:

```
Dim sDebugMsg as String
sDebugMsg = Ò will self-destruct in Þve seconds, Mr.
➥Phelps.Ó
Debug.Print ÒThis messageÓ & sDebugMsg
```

When you need to combine multiple text expressions for use with a single Debug.Print statement, the syntax starts to get a bit cluttered. After each text expression, you can tell VB where to put the next expression. There are three ways to do this.

First, you can place a semicolon after a text expression. This puts the insertion point immediately after the last character displayed.

NOTE

How VB Treats Consecutive Debug.Print Text Expressions If you try to place two Debug.Print text expressions immediately after one another, VB will insert a semicolon between them for you. In other words, if you type:

```
Debug.Print ÒMessage #1Ó
➥ÒMessage #2Ó
```

As soon as you press Enter to move to the next line, VB will automatically change your debugging message into this:

```
Debug.Print ÒMessage #1Ó;
➥ÒMessage #2Ó
```

That is, the first character in the next expression that prints will be immediately after the last character in the expression preceding the semicolon. For all practical purposes, this behavior makes the semicolon act just like the "&" concatenation operator. In fact, the last line of the preceding example could have been written like this:

```
Debug.Print ÒThis messageÓ; sDebugMsg
```

Second, you can use the Tab() function to move the insertion point to a specific column. If you want some space between your messages, you might try something like this (if you have an exceptionally wide screen):

```
Debug.Print ÒThatÕs one small step for manÓ; Tab(100); Ò;
➥one mighty leap for VBÓ
```

What happens if you specify a Tab position that would cause part of the previous text expression to be overwritten? The first text expression in the preceding example is 29 characters long, for example. What happens if you enter this?

```
Debug.Print ÒThatÕs one small step for manÓ; Tab(11); Ò;
➥one mighty leap to the next line for VBÓ
```

You might expect the output in the Immediate window to look like ÒThatÕs one;one mighty leap to the next line for VBÓ , starting the second text expression in column 11 of the same line. In fact, VB saves you from such mistakes by moving the second expression to column 11 of the next line.

Finally, you can use Tab with no argument (remember to omit the parentheses too; otherwise, VB will generate a syntax error) to position the insertion point at the beginning of the next print zone. (On average, a print zone occurs about every 14 columns.)

Remember that all the semicolons and Tabs are optional. If you don't specify where to place the next character, it will print on the next line. You can use these formatting rules to produce output in a variety of ways. However, this example should give you some idea of how to combine them. Figure 18.15 shows the result:

LISTING 18.3

THE Debug.Print MESSAGE

```
Private Sub cmdOK_Click()
    Dim sRef As String, sMsg1 As String, sMsg2 As String
    sRef = Ò1234567890123456789012345678901234567890123456789 0Ó
    sMsg1 = ÒI am a string exactly 40 characters longÓ
    sMsg2 = Òfollowed by more text at column 46.Ó
    Debug.Print sRef
    Debug.Print sMsg1; Spc(5); sMsg2
    Debug.Print sMsg1; Tab(46); sMsg2
    Debug.Print sMsg1; Tab(Len(sMsg1) + 6); sMsg2; vbCr
    Debug.Print ÒThe End!Ó
End Sub
```

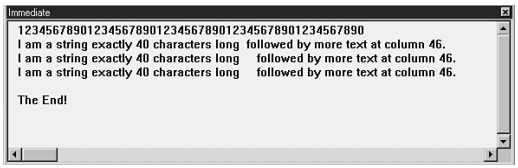


FIGURE 18.15
The output of the Debug.Print formatting example looks like this.

Among other things, notice how it is legal to use a function call and calculations in the Debug.Print line. The Len() function determines how long the sMsg1 string is, and then you add another six columns to that to duplicate the output of the other lines. Why is there an empty line just before ÒThe End!Ó ? That's because there isn't a position specifier after the vbCr constant (yes, built-in constants are available, too). You didn't specify where to place the next character, so it printed on the next line following vbCr .

Displaying Data Values

So far, you have only been using Debug.Print to print simple text messages to yourself, but you can also use it to display data values. (Because some of your text messages have used string variables, perhaps that is obvious.) A few rules apply to data variables when they are displayed via Debug.Print . Most of these rules apply equally well to every other aspect of VB, so they are pretty straightforward and should not cause you any difficulty even if you don't remember them.

The Immediate window is aware of any locale settings you have established for your system. Consequently, if you use Debug.Print to display numeric data, the values will be displayed with the appropriate decimal separator, and any keywords will appear in your chosen language. Date variables will be displayed in the short date format recognized by your system. Boolean values are displayed as either true or false. These are the same default behaviors you should see anywhere in VB.

If you are testing variables to see whether they contain values, remember that *empty* is not the same state as null. If the value of an empty variable is displayed, nothing is printed (not the word *nothing*, but literally nothing). If the variable is null, the word *Null* is printed in the Immediate window.

Here are some examples of how data may be output to the Immediate window:

LISTING 18.4

OUTPUT OF DATA TO THE IMMEDIATE WINDOW

```
Sub DebugPrintExamples()
    Ô Display a decimal value
    Dim s As Single
    s = 3.14159
    Debug.Print ÒThe value of s isÓ; s
    Ô Display a Boolean
    Dim fMakesSenseToMe As Boolean
    Debug.Print ÒThe default value of a Boolean is Ò;
➥fMakesSenseToMe
    Ô Display a date
    Dim d As Date
    d = Date
    Debug.Print ÒToday is Ò; d
    Ô Empty vs. Null
    Dim var1 As Variant, var2 As Variant
    Debug.Print ÒBefore any assignment, a Variant is Ò; var1
    Debug.Print ÒCanÕt see that?  Of course not; it really IS
➥empty!Ó
    Debug.Print ÒOK, True or False.  The statement Ôvar1 is
➥EmptyÕ is Ò; IsEmpty(var1)
    Debug.Print Òand the statement Ôvar1 is NullÕ is Ò;
➥IsNull(var1)
    Ô After an assignment, can var be reassigned a value of
➥Empty?
    var1 = 1: var1 = Null
    Debug.Print ÒNow, var1 is Ò; var1;
    Debug.Print Ò, so the value of IsEmpty(var1) is Ò;
➥IsEmpty(var1)
```

```
    Debug.Print ÒNot surprisingly, the statement Ôvar1 is NullÕ
➥is Ò; IsNull(var1)
    var1 = Empty
    Debug.Print ÒA Variant can become Empty again by
➥assignment: Ò; IsEmpty(var1)
    var1 = 1: var1 = var2
    Debug.Print Òor by being assigned the value of another
➥empty Variant: Ò; IsEmpty(var1)
End Sub
```

It used to be that after a Variant had been assigned a value, it could become empty again only by being assigned the value of another empty Variant. Now you can just reassign the value *empty* to the variable.

USING THE DEBUG.ASSERT METHOD

As seen in previous sections in this chapter, you can cause your code to enter Break mode in various ways. With the exception of the Stop statement, however, none of the automated techniques previously discussed (toggling a breakpoint or setting a Watch expression whose type is Break When Value Changes or Break When Expression Is True) can be stored between sessions of your VB development environment. The Watch expressions or breakpoints you set up during a VB development session are lost as soon as you exit VB or exit the VB project.

The Debug.Assert method enables you to save breakpoints and break conditions with your code so that they persist from one development session to the next.

Debug.Assert causes your code to enter Break mode at design time if a specified condition is False .

The format for a call to Debug.Assert in your code is this:

```
Debug.Assert logical condition
```

where *logical condition* is any expression that evaluates to a True or False value. For instance, if you wanted your application to enter Break mode whenever the variable intEmployees were 0, you would write the line:

```
Debug.Assert intEmployees <> 0
```

In other words, you are asserting that intEmployees is not 0. If it is, the Assert method forces the application to enter Break mode.

What if you always wanted your application to enter Break mode at a particular point, regardless of conditions? The line

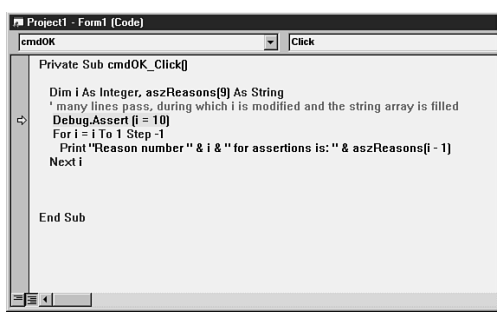
```
Debug.Assert False
```

would do the trick, because False is, of course, always false.

Like the Debug.Print method, Debug.Assert has no effect when the compiled executable file runs. This means that you can also permanently leave Debug.Assert statements in your production code without any problems for your executable file.

Figure 18.16 shows what the result of an assertion failure looks like in the debugging environment.

FIGURE 18.16
An assertion failure.



Notice how the code executed normally until it reached the Debug.Assert line. The expression Òi = 10Ó evaluates as False, because i has just been declared and VB initializes Integer variables to 0 by default. Because this expression is False, the Debug.Assert on this line caused the application to enter Break mode. The line with the assertion is highlighted, and an arrow immediately to the left of the assertion indicates the program is still running, but execution is suspended at the indicated line.

So far, this discussion has focused on the mechanics of the Assert method, but has not said much about when to use them. Consider using an assertion whenever your code depends on an expression satisfying certain criteria.

If you have already gone to great lengths to ensure that your code is trouble free, why not take an extra step to verify that you haven't let something slip through the cracks? Even after you have applied every reasonable test that you can imagine, an assertion may lead you to some things you had not imagined. If you plug the expression into an assertion only to trigger an assertion failure, maybe your original tests weren't as good as you thought.

By alerting you to conditions that are contrary to your expectations, assertions help you to create tests that more completely reflect the actual conditions under which your code must perform. When you trigger an assertion failure, you know at once that your assumption is wrong, meaning that you must either do something to ensure that it continues to be valid or reframe the assumption.

Generally speaking, it is a good idea to get into the habit of using assertions. Remember, however, that an assertion is not a substitute for an error handler. It can show you where there may be a problem, but an assertion can't resolve any problems in your code at runtime because it isn't part of the compiled program.

INTERACTING WITH THE IMMEDIATE WINDOW

So far, you have been using the Immediate window as a sort of global message box. Although Debug.Print sends useful messages during program execution, it is also possible to type directly into the Immediate window. What can you type? For starters, you can use the same Debug.Print commands that you have been using so far in code. Figure 18.17 shows an example. The syntax is as follows:

```
Debug.Print ÒInsert Your Message HereÓ
```

FIGURE 18.17
Debug.Print commands can be entered directly into the Immediate window.

Because the Debug object is the default object for the Immediate window, you can take advantage of some shortcuts. As this example shows, Debug.Print can be abbreviated either to Print or just ?. The method is invoked just as if you typed it completely, so you may as well save yourself that extra typing.

Besides printing messages, the Immediate window can perform a few other useful tricks when you use it interactively. You will look at those next.

Querying or Modifying Data Values

Besides printing text messages, you can use the Print method to display the value of a variable. It is also possible to change the value of a variable inside the Immediate window. The only catch is that the program must be in Break mode to perform either of these actions.

If you think about it, one reason for this constraint is simple: A local variable exists only in the context of the function in which it occurs. If the function has already finished, the variable no longer exists. Forcing the program to enter Break mode during the execution of a routine ensures that the variables in that routine still exist for you to view or modify. Global variables and form-level variables persist beyond particular functions, of course, but the rules apply to all interactive uses of the Immediate window during program execution: Your program must be in Break mode. Figure 18.18 provides an example.

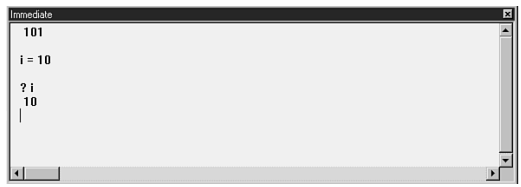


FIGURE 18.18
Querying and changing a variable in the Immediate window.

In this case, the variable *i* is incremented in a For loop. When the loop finishes, the Debug.Print statement in the code displays the current value of the variable in the Immediate window, and then the embedded Stop command throws the program into Break mode. At this point, the programmer changes the value of *i* interactively in the Immediate window, and then the value change is confirmed via the

Debug.Print method (which is invoked via the question mark). Sure enough, the value has been changed from 101 to 10.

Besides modifying variables interactively, you can also change the values of properties. If you want to hide the form whose code is currently running, for example, you can type Òme.Visible = FalseÓ in the Immediate window. The form will vanish as soon as you press Enter. To restore the form's visibility, of course, you can type Òme.Visible = TrueÓ. Again, your program must be in Break mode for this to work.

The capability to modify variables and properties in the Immediate window gives you a chance to test a variety of conditions in your debugging sessions that your program will face when it is released. Want to see what happens if a variable value is (pick one) a specific value/very small/very large/empty/null? Modify the variable and see what happens. Want to see what happens if you change a property of a control or form? Go ahead. If you can do it in code, you can do it in the Immediate window.

NOTE

Ending Value of a Loop Counter Variable In the example in Figure 18.18, did you expect *i* to be 100 at the end of the loop? Remember that 100 is the last legal value for the loop. When *i* equals 100, it is incremented one last time by the Next statement in the final iteration. Thus when the loop exits, *i* equals 101.

Testing and Executing VB Procedures

Although most of the uses to which you have put the Immediate window so far have required that a program be suspended in Break mode, you can do some things with the Immediate window that don't require a program at all. If you want to execute a command or run a procedure that you have written, you can just type the command or procedure name (along with any necessary parameters) into the Immediate window, and then press the Enter key.

Assume, for example, that you want to delete a file using the Kill statement, but you aren't sure what will happen if you put the command in your program and the file doesn't exist. You can enter the command in the Immediate window twice in succession to see what it will do, as shown in Figure 18.19.

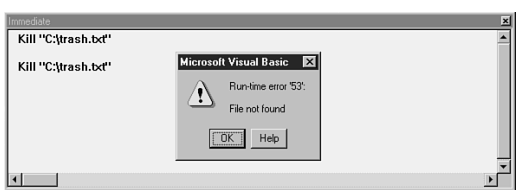


FIGURE 18.19
Deleting a file with the Kill statement.

As you can see, nothing special happens in the Immediate window to indicate that the command succeeded. If you try to execute the `Kill` command again, however, you will generate a runtime error (#53: `File not found`) because the file was successfully deleted the first time.

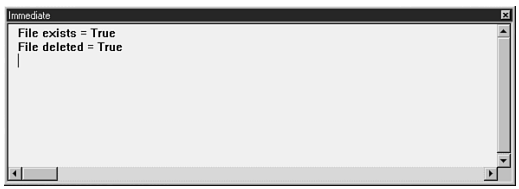
Perhaps it would be nice to have a more informative way to delete a file. There is no way to modify the built-in `Kill` statement to provide extra information, but you can write your own file deletion function that does. Here is one that uses the `Kill` command to do its dirty work, but that uses the Immediate window to display its status along the way. Most important, the new function also provides a return value to indicate whether it succeeded:

LISTING 18.5

A File Deletion Function

```
Function Delete(sFilename As String) As Boolean
    Ô use the return value in the calling function rather than
➥trap the error here
    On Error Resume Next
    Dim fReturn As Boolean
    #Const DEBUGGING = True
    Ô see if the Þle exists to delete
    If Dir(sFilename) <> ÒÓ Then fReturn = True
    #If DEBUGGING Then
        Debug.Print ÒFile exists = Ò; fReturn
    #End If
    If fReturn Then
        Ô Þle exists, so kill it
        Kill sFilename
        Ô if couldnÕt delete, set return value
        If Dir(sFilename) <> ÒÓ Then fReturn = False
        #If DEBUGGING Then
            Debug.Print ÒFile deleted = Ò; fReturn
        #End If
    End If
    Delete = fReturn
End Function
```

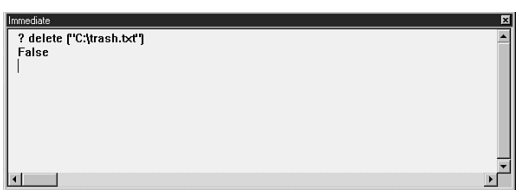
If you run this function twice in succession, the Immediate window looks like Figure 18.20.



, **FIGURE 18.20**
Results of the Delete function displayed in the Immediate window.

The Delete function is more informative than the Kill statement at debug time because it uses Debug.Print to keep you posted about its progress. Aside from the debugging output, Delete also gives you more to work with in your program. Because it provides a Boolean return value, you can test for the success of the file deletion and take appropriate measures in your program.

Because Delete is a function, it also enables you to display useful output in the Immediate window without sprinkling Debug.Print statements throughout its body. If you remove the Ò#Const DEBUGGING = TrueÓ line and run Delete from the Immediate window, you can use the Debug.Print statement to display its return value, as shown in Figure 18.21.



, **FIGURE 18.21**
Results of the Delete function displayed in the Immediate window with the Debug.Print method.

This gives you two ways to run a procedure from the Immediate window: with or without prefacing it with Debug.Print. If you just want to execute a procedure and disregard its return value (if it has one), don't use the Print method. If you want to display the return value of a function, preface it with a leading question mark to invoke Debug.Print.

NOTE

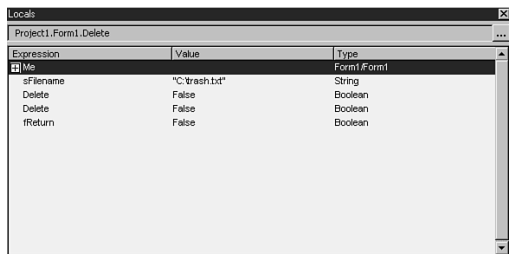
Functions Versus Subs in the Immediate Window You can treat a function like a sub in the Immediate window, but you can't treat a sub like a function. If you inadvertently put a leading question mark in front of a sub that you try to run in the Immediate window, you will get this error message: Compile error: expected function or variable.

USING THE LOCALS WINDOW

One of VB's debugging windows can save you from endlessly embedding Debug.Print statements in code to examine the value of variables: the Locals window, which displays the current value of variables within the scope of the currently executing procedure.

Because what it displays depends on the current scope, the contents of the Locals window changes whenever a different procedure executes. To display the Locals window, pull down the View menu and choose Locals Window. Figure 18.22 shows the Locals window.

FIGURE 18.22
The Locals window.



Calling it a "Locals" window is somewhat misleading, because it isn't limited to viewing only those variables that are local in scope. You will see all the local variables, of course, but form-level variables are also visible in the Locals window. (Remember that you will only see the form-level variables for the current form.) For module-level variables, you have to use the other methods covered in this chapter (for example, the Watch window and Debug.Print). The Locals window can't display module-level variables.

Here is an example. Module_1 contains nothing but a global variable, declared like this:

```
Public g_Test As Integer
```

In Form1, a form-level variable is declared in the General Declarations section:

```
Public f_Test As Long
```

Here is the rest of the Form1 code:

```
Private Sub Form_Initialize()
    Dim i as integer
    i = 1
    g_Test = 10
    f_Test = 1000
End Sub
Private Sub Form_Click()
    Dim k as integer
    k = 5
    Stop
End Sub
```

Values are assigned to the form-level variable and the module-level variable in the form's Initialize event. When the form is clicked, the program enters Break mode. This is necessary because the contents of the Locals window depend on a specific local scope. After the program enters Break mode, you can use the Locals window to examine the status of the variables.

When the program enters Break mode, the Locals window looks like Figure 18.23.

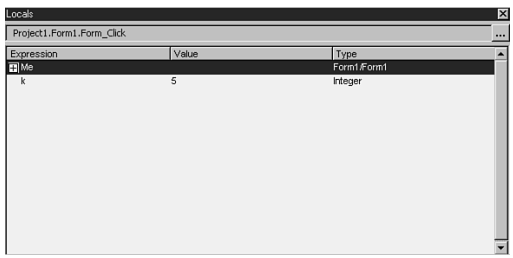


FIGURE 18.23
The local variable and ME are available in the Locals window.

Because program execution was broken in the form's Click event, the value of *k* is available because it is local to that event. Notice what isn't available: You can't see the value of the *i* variable from the Initialized event because it isn't local to the current procedure. You also can't see the form-level variable *f_Test* or the module-level variable *g_Test*. You do see something called ME, however, which has a boxed plus sign immediately to its left.

You may recall that ME is VB shorthand for the form in which code is currently executing. If you click on the plus sign, the ME object will unfold, displaying all the current form's properties in alphabetic order. Look toward the bottom of the window displayed here and you will see that *f_Test* is displayed, but *g_Test* is not. A form-level variable is essentially a property of a form, and so it is available here. The module-level variable, however, is not. What happened to *k*? It is still there, but when you clicked on ME to display its contents, it scrolled to the very bottom of the window.

At the top of the Locals window, notice that the name of the currently executing procedure is identified. To the right of the procedure name, the button with the ellipsis can be used to display the Call Stack window. The Locals window also enables you to do one more trick: You can use it to change the value of a property or a variable. If you click on the value displayed for an item, you can type a new value to replace it. The change won't take effect until you press Enter.

Using the Immediate Window in Place of Breakpoints

Many of the techniques that have been used in this chapter to display data require a program to be in Break mode. Sometimes, however, that can cause problems. If you know when Break mode can give you fits, you can use Debug.Print to get the information you need in the Immediate window instead of setting watches or breakpoints.

Using the MouseDown and KeyDown Events

The first potential problem occurs in association with key events and mouse events. If you enter Break mode during a MouseDown or KeyDown event, you are probably going to release the key or mouse button while you are still in Break mode. After all, you need to use the keyboard and the mouse yourself to carry out your debugging tasks. If you break execution during either of these events and then resume program execution, your application doesn't know enough to trigger a MouseUp or KeyUp event for you. After all, when it *went to sleep* the button or key was down; and now that it is awake, it thinks that's still the case.

To get the MouseUp or KeyUp event, you need to press the button or key again and release it. But that's where the program goes into Break mode again (a veritable "Catch-22"), so you never really get to the MouseUp or KeyUp events.

Solution: Don't enter Break mode in these events. Instead, use Debug.Print to display the necessary data in the Immediate window.

Using the GotFocus and LostFocus Events

The problem here doesn't pose the same logical difficulty presented by the MouseDown and KeyDown events. Instead, a breakpoint in GotFocus or LostFocus can just throw off the timing of system messages as they are sent. Windows is so sensitive to the timing of system messages that the API even includes several different ways to dispatch them.

SendMessage will send a message to the target object, for example, waiting until the message has been successfully sent before returning a value. PostMessage , on the other hand, returns at once after adding its message to the queue; it doesn't wait for the results. Function calls seem to occur so quickly that it may seem strange to think that the difference would be so important as to require a different function. Remember that a computer's memory access is measured in nanoseconds, however, and perhaps it will make more sense.

Timing is critical, and a breakpoint in the Focus events can mess things up. If you need to display values during a GotFocus or LostFocus event, use Debug.Print to send the data to the Immediate window.

LEVELS OF SCOPE

. Given a scenario, define the scope of a Watch variable.

Just as the scope of a variable depends on where and how it is declared, the scope of a watch depends on the manner of its definition. Specifying the scope of a watch determines the context in which its behavior is observed. If you know where in your program the observation of a variable is significant, you can make more efficient use of your debugging time by specifying the appropriate Watch context.

You are already familiar with the issue of scope as it pertains to data variables. In a block-structured programming language such as Visual Basic, the visibility of a variable depends on where it is declared. A variable may exist at any one of the following three levels of scope:

- Local
- Module
- Global

A variable declared in the context of a particular procedure is visible only in the context of that procedure. It is said to be local in its scope, which means that it is invisible to all other procedures in the program. Consider the following subprocedures, for instance:

```
Sub MyProcedure
    Dim iCount as Integer, sName as String
    Ô pretend that something useful happens later in the
➥procedure
End Sub

Sub YourProcedure
    Dim iCount as Integer
    Ô pretend that something else useful happens here, too
End Sub
```

The variables declared in these routines are all local in scope. The variable sName declared in MyProcedure can be accessed only by the code in MyProcedure. It is not visible to YourProcedure, nor is it visible to any other routine in the program. The two iCount variables are each local to the procedures in which they are declared. They are permitted to have the same name because each is unique in the context of the procedure in which it is declared.

A variable may also have module scope if it is declared in the General Declaration section of its code module using the keyword Private. In this case, the variable is visible to all procedures contained in the module, but it is invisible to all other procedures in the program.

Finally, a variable is global in scope if it is declared in the General Declaration section of its code module using the keyword Public. Such a variable is visible to all procedures contained throughout the program.

If you already understand how variables may have their scope defined at these different levels, you will find the scope of a watch fairly easy to understand. Just as a variable's scope may be at the local, module, or global level, the scope of a watch may be defined at these levels too.

Local Scope

Although the scope of a variable depends on where it is declared in the source code, watches are all created using the same Add Watch dialog box, and edited using the same Edit Watch dialog box. The key to differentiating among the various levels of scope lies in the controls contained in the Context group of the Watch dialog boxes.

The Context group contains three controls: Procedure, Module, and Project. The control pertaining to Project is a simple label that displays the name of the current project. Because it is a label, it can't be edited. It serves as a reminder that only those procedures and modules will be displayed by the other controls in the group.

The real work of determining the scope of a watch is done by the other two controls. If you want to define a watch at the procedure level, that means that you only want to assess the value of the Watch expression in the context of a particular procedure. To do this, use the Module combo box to select the appropriate module. The Module combo box displays the names of all modules contained in the current project.

After a module name is selected, use the Procedure combo box to choose the procedure for which the watch should be active. The Procedure combo box displays only those procedures contained in the currently selected module. If the procedure you want isn't in the list, check the Module combo box again. You probably selected the wrong module by mistake.

Figure 18.24 shows how to use the Add Watch dialog box to create a procedure-level watch.

Remember that if you need to modify the settings for a watch, you can use the Edit Watch dialog box to do so. The following section shows you an example of this.

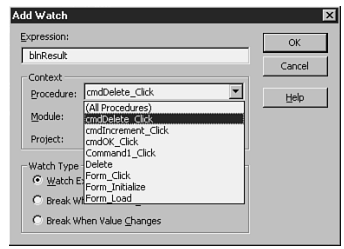


FIGURE 18.24
A procedure-level watch.

Module Scope

If you want to monitor an expression as it changes throughout an entire module, you don't need to create individual procedure-level watches for each routine contained in the module. Instead, you can set a single watch that applies to an entire module.

To define a watch at the module level, use either the Add Watch dialog box (to create a new watch) or the Edit Watch dialog box (to modify an existing watch). Just as with a procedure-level watch, you still need to select the appropriate module from the Module combo box.

As before, the next step is to select an item from the Procedure combo box's drop-down list. Instead of selecting a particular procedure name, however, select the item that says (All Procedures). (All Procedures) is the first item contained in the drop-down list. By selecting (All Procedures), your watch is automatically activated for every procedure contained in the currently selected module.

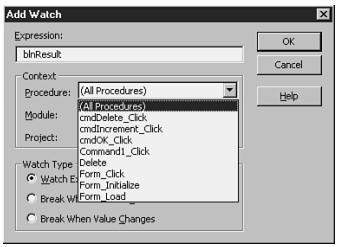


FIGURE 18.25▲
A module-level watch.

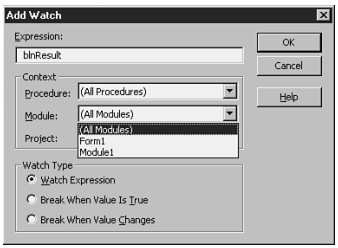


FIGURE 18.26▲
A global watch.

Figure 18.25 shows how to use the Edit Watch dialog box to change the local watch you set in the preceding section into a module-level watch.

Global Scope

What if you want to monitor an expression throughout your entire program? Just as you can set a single watch pertaining to an individual module, you can also set a single watch that applies to an entire program.

To define a global watch, you once again can use either the Add Watch dialog box or Edit Watch dialog box. First, select All Modules from the Module combo box. (As with the Procedure combo box, the All selection is the first item in the drop-down list.) This will also automatically set the Procedure watch to All Procedures in the Procedure combo box. Figure 18.26 shows an example of this.

The Watch expression in Figure 18.26 will be evaluated in all procedures in all modules as the program executes, making it global in scope.

As you set watches at various levels of scope, you will notice that the Watches dialog box displays the context of each expression. As shown in Figure 18.27, a procedure-level watch is indicated to apply to a particular procedure by the ModuleName.ProcedureName syntax in the Context column of the Watches dialog box. A module-level watch has the name of its module displayed in this column. A global watch is indicated when the entry is blank, denoting that there is no contextual limitation on the watch.

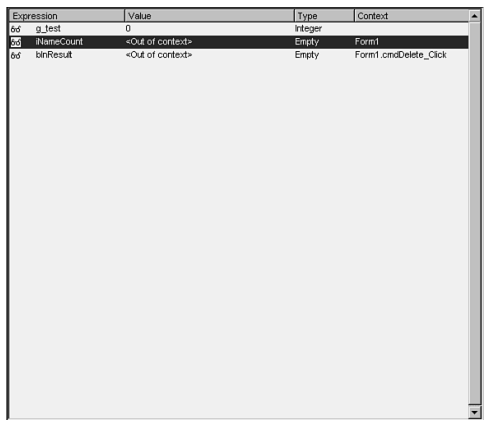


FIGURE 18.27.
The Watches dialog box indicates the level of scope for each Watch expression.

SCOPE CONSIDERATIONS

As you have seen, a watch may be set at any of three levels of scope. This section considers how to decide which level of scope is best suited to solve a given problem.

Striving to Narrow the Scope

The rules governing intelligent declaration of variables largely apply to watches too. It generally makes sense to limit the scope of a variable, for instance. If a variable must be used in just one procedure, it should be declared locally within that procedure; declaring it at a higher level is a needless complication. In particular, global variables should be used sparingly. Because they can be modified anywhere in a program, bugs involving global variables are harder to track down.

In essence, the task of debugging is vastly simplified as the number of places in which data can be modified diminishes. This rule also applies to watches. Limiting the number of places in which a Watch expression is evaluated cuts down on distractions during the debugging process. After all, if you monitor a variable in 100 procedures when you only need to monitor it in 10, you expend about 10 times more effort on it than it is worth.

You will recall from the section "Types of Watch Expression" that you can do three things with a watch:

- Monitor the value of the Watch expression
- Break when the Watch expression is `True`
- Break when the value of the Watch expression changes

Bearing these options in mind, this discussion begins to focus on some strategies and constraints regarding watches set on different kinds of variables.

Global Variables

A global variable can be evaluated in a watch at any level of scope. Because it is available throughout the program, its value can be detected in a watch of module-level or procedure-level scope as easily as in a global watch.

If you suspect that a global variable is being incorrectly modified, you will probably want to begin with a global watch. The value of the variable can be changed anywhere in your program. If you set the watch to break on change or when the Watch expression is true, however, you will be able to tell which procedure has modified the value. As you find likely suspects for the bug in question, you may wish to narrow the scope of your watch, or to set additional watches of more limited scope.

Module-Level Variables

A module-level variable can be evaluated in a watch at either of two levels: module level or procedure level. Naturally, a procedure-level watch makes sense only for procedures found in the same module in which the variable is defined.

The same search pattern suggested for global variables makes sense for module-level variables. Until you have reason to focus the search on a particular procedure or set of procedures, you will initially want to set the watch at the highest scope available. Use watches to trigger a program break to determine which procedure(s) is the source of your problem.

Local Variables

Obviously, only a procedure-level watch is sensible for local variables. Its value can be evaluated only in the context of the procedure in which it is defined. It literally does not exist anywhere else, so there's no point in looking for it elsewhere.

The point to limiting the number of places in which your watches are active is so that you can spend less time looking at meaningless changes. After you have pinned down the likely sources of a bug, it is only logical to limit the search to the likely problem areas.

Performance Concerns

Aside from the logical issue, performance concerns are also a motivation to limit the scope of a watch. Broader watches cause VB to devote more resources evaluating watch expressions than do watches of more limited scope.

Does the performance degradation produced by a watch that is broader in scope than necessary really matter? Most of the time, probably not. If you are working with anything particularly time sensitive, however, the time you may save by limiting the scope of your watch may make a difference. Even a few milliseconds may be important to a real-time system, a callback function, or a program relying on the sequence in which a series of Windows messages is processed.

If you encounter problems along these lines during a debug session in which you are using watches, try deactivating the watches or limiting their scope before you decide that you have found more bugs in your program. If the problems go away after you have changed the watches, it is likely that the watches slowed execution just enough to cause problems.

CHAPTER SUMMARY

KEY TERMS

- Assertion
- Call stack
- Immediate window
- Locals window
- Scope
- Watch

In summary, this chapter covered the following topics:

‡ Using Watch expressions

‡ The meaning and use of Break mode, including the various Step options, the use of the Watch window, and different methods for entering Break mode

‡ Using the Immediate window and the `Debug` object

‡ Using the Locals window

‡ Scope of watches

APPLY YOUR KNOWLEDGE

Exercises

18.1 Using the Call Stack

Most programs will have procedures that call other procedures that call still other procedures. VB has a tool to show the call stack to help you keep track of the calling chain. You can use the Call Stack window any time you need to trace the interactions of procedures in a program. This exercise demonstrates its use.

Estimated Time: 30 minutes

To create this exercise, follow these steps:

1. Create an application consisting of a single form with a command button. The form should resemble that shown in Figure 18.28.

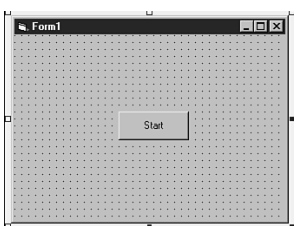


FIGURE 18.28
The call stack demonstration form.

2. Enter the following code for the Click event of the command button:

```
Private Sub cmdStart_Click()
Cls
    Print ÒBeginning Start button code.Ó
    Print ÒCalling Procedure AÓ
    ProcedureA
    Print ÒEnding Start button code.Ó
End Sub
```

3. Create three additional procedures in the form, as follows:

```
Private Sub ProcedureA()
Print Tab(5); ÒBeginning Procedure A.Ó
    Print Tab(5); ÒCalling Procedure BÓ
    ProcedureB
    Print Tab(5); ÒEnding Procedure A.Ó
End Sub

Private Sub ProcedureB()
    Print Tab(10); ÒBeginning Procedure B.Ó
    Print Tab(10); ÒCalling Procedure CÓ
    ProcedureC
    Print Tab(10); ÒEnding Procedure B.Ó
End Sub

Private Sub ProcedureC()
    Print Tab(15); ÒBeginning Procedure C.Ó
    Ô Stop
    Ô Print ÒEntering break modeÓ
    Print Tab(15); ÒEnding Procedure C.Ó

End Sub
```

4. Run the exercise program and click on the command button. The form will display text in the form indicating the sequence in which the program's procedures have been called.

5. Close the form. Open the form's code module and remove the comment characters from the ProcedureC code. ProcedureC should now look like this:

```
Private Sub ProcedureC()
    Print Tab(15); ÒBeginning Procedure C.Ó
    Stop
    Print ÒEntering break modeÓ
    Print Tab(15); ÒEnding Procedure C.Ó
End Sub
```

6. Run the application again. When the program enters Break mode, pull down the VB View menu and choose the menu option titled Call Stack (notice that Ctrl+L is a keyboard shortcut for the call stack).

APPLY YOUR KNOWLEDGE

7. Experiment with the Call Stack window. Notice how you can use it to view the point in any active procedure where a subsequent procedure call was issued. You can use information about the call stack to guide your decisions about how to use the Debug menu's stepping commands to step into, out of, or to a particular line via step to cursor.

18.2 Using the Debug.Print Command

In this exercise, you practice using the Debug.Print command to display formatted output in the Immediate window.

Estimated Time: 15 minutes

To create this exercise, follow these steps:

1. Create a new VB application containing a single form.
2. Code the form's Initialize, Load, Resize, Activate, GotFocus, Paint, Unload, and Terminate events to display a message in the Immediate window notifying you of the current event. Progressively indent the messages in the order of their occurrence until the form paints, and then unindent them as the form exits.
3. Experiment by inserting similar messages into the form's other events. Interact with the form—change its size, obscure it with another program's window, and so on. Which events occur only once? Which ones continue to occur?

18.3 Modifying Values in the Immediate and Locals Window

In this exercise, you use the debugging tools to modify program values at runtime.

Estimated Time: 30 minutes

To create this exercise, follow these steps:

1. Create an application that contains at least one module and one form. Create Public integer variables both in the form and the module.
2. Add code to the form's Click event that increments the value of the variables and displays them on the form with the form's Print method. (Hint: You also may want to use the Cls method when the form's CurrentY property approaches its height.) Be sure to set a breakpoint so that you can use the debugging windows.
3. Set the form's AutoRedraw property to True so that its contents will be preserved when you return to it from the debugging windows.
4. Click on the form a few times to confirm that the variables are incrementing and displaying properly. After you have displayed a few values, use the Immediate window to change the form's Forecolor property. (Unless you have specific hexadecimal values memorized, this may be most easily accomplished via the RGB function, or by using VB's built-in named Color constants, such as vbRed, vbMagenta, and so on.) Clear your breakpoint and click on the form again to confirm that the printed output has changed color.
5. Reset the breakpoint and try changing the form's Forecolor property again using the Locals window. If you have trouble selecting a particular value, try using the RGB function again or one of the color constants. Notice how the property value changes to reflect the return value of the function. Continue to experiment with changes in other properties by using both the Immediate window and the Locals window.
6. After you have finished experimenting with prop-

APPLY YOUR KNOWLEDGE

erties, try modifying the variables by using both windows. Which way do you prefer? Does either window permit you to do anything that the other window can't do?

18.4 Setting the Scope of a Watch

This exercise shows how to use the Edit Watch dialog box to create watches at different levels of scope.

Estimated Time: 15 minutes

To create this exercise, follow these steps:

1. Create a new project.
2. Add a code module to the project. Enter the following code into the General Declarations section of the module:

```
Public g_iCounter as Integer
Private m_strModuleString as String
```

3. Create a new procedure in the code module as follows:

```
Public Sub MyProcedure
    Dim iAlphaCode as Integer
    m_strModuleString = ÒÓ
    For iAlphaCode = 65 To 96
        m_strModuleString = m_strModuleString &
➥Chr(iAlphaCode)
    Next
    Ô count the number of times this procedure
➥runs
    g_iCounter = g_iCounter + 1
End Sub
```

4. Enter the following code for the `Click` event of the default form VB provides:

```
Call MyProcedure
```

5. Set watches for each of the variables declared in the code module (`g_iCounter`, `m_strModuleString`, and `iAlphaCode`). Use the `Context` controls in the Watches dialog box to experiment with different scopes as described in this chapter.
6. The test code gives you one sample each of a global variable, module-level variable, and local variable. What difference does the scope make in the evaluation of each variable type?

18.5 Changing the Scope of a Watch

This exercise shows how to use the Edit Watch dialog box to modify the scope of a watch.

Estimated Time: 20 minutes

To create this exercise, follow these steps:

1. Create a new project.
2. Add a code module to the project. Enter the following code into the General Declarations section of the module:

```
Public g_iTestVariable as Integer
```

3. Create a new procedure in the code module as follows:

```
Public Sub TestProcedure
    On Error Resume Next
    Ô count the number of times this procedure
➥runs
    g_iTestVariable = g_iTestVariable + 1
    If g_iTestVariable > 32767 Then
        g_iTestVariable = 0
    End If
End Sub
```

4. Enter the following code for the `Click` event of the default form VB provides:

```
Call TestProcedure
```

5. Set a global watch that will break when the value of `g_iTestVariable` changes. Run the application and pay attention to which module the break occurs in. Change the scope of the watch so that

APPLY YOUR KNOWLEDGE

it is active only in that module. Run the application again. This time pay attention to which procedure the break occurs in. Change the scope of the watch again so that it is active only in that procedure.

Using this general approach, you can narrow the scope of a watch so that it encompasses only that scope relevant to the variable being watched. In a live application, of course, you will want to exercise the program more to see whether the variable changes anywhere else before jumping to the conclusion that it needs to be watched only in a single module or procedure.

Review Questions

1. Identify the various kinds of watches available in VB6.
2. How are arrays and user-defined types displayed in a Watch window?
3. What is Break mode?
4. What method of the Debug object can be used to display a value in the Immediate window during a debugging session?
5. A boxed plus sign appears in the Locals window next to the name of a variable being processed in a loop. What does this mean?
6. What happens when a procedure name is entered into the Immediate window?
7. At what three levels may the scope of a watch be set?
8. How does the scope of a watch affect its calculation time?
9. What choices must be selected in the Context group for a watch to have global scope?

Exam Questions

1. What cannot be changed in the Watch window?

   A. Watch expression

   B. Watch on demand

   C. Watch, break on change

   D. Watch, break on true

2. If you no longer need to monitor the values in the current procedure, but you want to continue single-stepping through another procedure, which options can you select to continue single-step execution in the other procedure?

   A. Step Into

   B. Step Over

   C. Step out Of

   D. Step to Cursor

3. You entered Break mode in the current debugging session by pressing Ctrl+Break. To resume program execution, you should do what?

   A. Select Continue

   B. Toggle the current breakpoint from on to off

   C. Select Stop to exit Break mode

   D. Press Ctrl+Break again

APPLY YOUR KNOWLEDGE

4. Your program enters Break mode when the value of an integer (iVariable) Watch variable is assigned a value of 25. As the program continues, iVariable is later assigned a value of 0, but the program does not enter Break mode this time. The type of watch set on the variable is

   A. A simple Watch expression, iVariable >= 25

   B. Watch, break when expression changes

   C. Watch, break when expression is True

   D. A quick watch

5. If a plus sign appears next to a Watch variable used in a loop, you can click on the plus sign to:

   A. Increment the loop counter by one and process the next iteration of the loop

   B. Increment the value of the Watch variable

   C. Toggle the status of the watchpoint from active to inactive

   D. Display the elements contained in the user-defined type or array being watched

6. If you type an invalid expression for use as a Watch expression, VB will

   A. Display a compiler error

   B. Immediately display a watch error

   C. Display a watch error the first time it evaluates the expression in Break mode

   D. Do nothing

7. You have distributed your application to a client in the form of an EXE file. How can the client recover from a Debug.Assert failure that occurs when he runs the program?

   A. Press the F9 key to deactivate the breakpoint, and then press F5 to continue.

   B. Use the Locals window or the Immediate window to modify the bad value that triggered the failure.

   C. Nothing can be done. The client must exit and restart the program.

   D. This scenario cannot occur.

8. Using the Immediate window, you assign an integer variable a value of 32768. What happens?

   A. The program continues to execute normally.

   B. The Immediate window closes and the program shuts down.

   C. A runtime overflow error occurs.

   D. The variable's value wraps to the first legal value, Ð32768.

9. An application has two forms: Form1 and Form2. Form2 is displayed only when a command button on Form1 is clicked. While Form2 is displayed, you use the Immediate window to change its Visible property to False and type **Unload Me**. What happens when you click on the button on Form1 that should display Form2?

   A. The program crashes.

   B. Form2 cannot be displayed because it is invisible.

   C. Form2 is displayed and its code is active, but the form is invisible.

   D. Form2 is displayed normally.

10. To place the insertion point immediately after the last character in an expression displayed in the Immediate window by Debug.Print , what must you place after the expression?

APPLY YOUR KNOWLEDGE

A. Colon

B. Semicolon

C. Ampersand

D. Comma

11. The Immediate window can be used interactively to debug a program

    A. Whenever a call is made to Debug.Assert or Debug.Print

    B. When a program is waiting for an event to occur

    C. When a program is in Break mode

    D. At any time

12. Debug.Print can be represented by symbol(s) when typed directly into the Immediate window.

    A. ?

    B. Print

    C. >>

    D. D.P

13. The scope of a watch should be narrowed when

    A. The program is time sensitive.

    B. The variable is static.

    C. The maximum number of watches of that scope is exceeded.

    D. The expression doesn't need to be evaluated in certain contexts.

14. The scope of a Watch expression is determined by

    A. The scope of the variable(s) used in the expression

    B. The Watch class module

    C. The Watch type library

    D. The Watch dialog box settings

15. A global variable is visible to a watch of which type of scope?

    A. Global

    B. Module-level

    C. Procedure-level

    D. Unbound

16. A module-level variable is visible to a watch of which type of scope?

    A. Global

    B. Module-level

    C. Procedure-level

    D. Unbound

17. The highest Watch scope meaningful to a variable declared in a form using the Public keyword is

    A. Global

    B. Module-level

    C. Procedure-level

    D. Unbound

18. The highest Watch scope meaningful to a variable declared in a module using the Private keyword is

    A. Global

    B. Module-level

    C. Procedure-level

    D. Indeterminate

APPLY YOUR KNOWLEDGE

Answers to Review Questions

1. Simple watch, Watch (break when expression changes), Watch (break when expression is True), quick Watch, and Watch on demand. See "Using Quick Watch," "Watching on Demand," and "Entering Break Mode Dynamically."

2. Arrays, user-defined types, and other objects appear in Watch windows with a boxed plus sign to the left of their name. Their data elements can be selectively displayed or hidden by toggling the boxed symbol between "+" and "–". See "Using the Watch Window."

3. In Break mode, a program is temporarily suspended during execution so that the programmer can inspect the program state. See "Using Break Mode."

4. The Debug object's Print method displays values in the Immediate window. See "Using the Print Method."

5. A boxed plus sign indicates that a variable contains subelements not currently displayed. This might indicate that the variable is an array, a user-defined type, or some other type of complex object. See "Using the Locals Window."

6. Procedures can be executed by typing them into the Immediate window. See "Testing and Executing VB Procedures."

7. A watch may be set at three different levels: the procedure level, module level, or globally. See "Levels of Scope."

8. The greater the scope of a watch, the slower it can be calculated. A watch set at the procedure level executes more quickly than a watch set at the global level. See "Scope Considerations."

9. The Context group enables you to select from among the modules and procedures in the current project. Global scope is specified by selecting All Modules in the Module combo box. See "Global Scope."

Answers to Exam Questions

1. **B**. The Watch on demand isn't available in the Watch window; it appears like a ToolTip when the mouse pointer lingers over an expression. For more information, see the section titled "Watching on Demand."

2. **A, C**. Both will continue execution of code one statement at a time, either in a procedure called from the current code (Step Into) or in the procedure which called the current code (Step Out). Step Over executes a called procedure in its entirety, but continues to single-step through the current procedure's code. Step to Cursor works only within the current procedure. For more information, see the section titled "Stepping Through Your Code."

3. **A**. Continue is the only way to resume execution. Deactivating a breakpoint in code doesn't automatically resume program execution. The other means of entering Break mode are one way; they don't toggle it on and off. For more information, see the section titled "Setting Stepping Options."

4. **C**. It isn't A or D, because neither a simple watch expression nor a quick watch affects Break mode. It isn't B because then the program would enter Break mode on any change to the variable. Remember that 0 is equivalent to False . For more information, see the section titled "Breaking on True."

APPLY YOUR KNOWLEDGE

5. **D**. The plus sign can be used to expand or collapse the variable being watched. For more information, see the sections titled "Watching Arrays" and "Watching User-Defined Types."

6. **D**. If an expression can't be evaluated, it is just as if it were out of scope. No error occurs. For more information, see the section titled "Creating a Watch Expression."

7. **D**. Assertions are not compiled into an executable program; they are only available in the debug environment. For more information, see the section titled "Using the Debug.Assert Method."

8. **C**. The value 32768 exceeds the bounds of an integer, so the assignment generates a runtime error. For more information, see the section titled "Querying or Modifying Data Values."

9. **D**. Setting ÒForm2.Visible = FalseÓ applied only to the current instance of the form. The next time the Form2 is loaded and displayed, the original defaults are used, so the form displays normally. For more information, see the section titled "Querying or Modifying Data Values."

10. **B**. The semicolon puts the insertion point immediately after the prior text. For more information, see the section titled "Formatting Debug.Print Messages."

11. **C**. A program must be in Break mode to use the Immediate window. For more information, see the section titled "Querying or Modifying Data Values."

12. **A, B**. Either ? or Print is shorthand for Debug.Print when entered into the Immediate window. For more information, see the section titled "Interacting with the Immediate Window."

13. **A, D**. Performance considerations may require you to narrow the scope of a watch so that it can calculate more quickly. If you don't need to observe a variable in certain contexts, it is safe to exclude it from the watch. For more information, see the section titled "Scope Considerations."

14. **D**. The Context group of controls on the Watches dialog box determines the scope of the watch. For more information, see the section titled "Procedure Scope."

15. **A, B, C**. Global variables are visible to watches of all scope levels. For more information, see the section titled "Global Variables."

16. **B, C**. Module-level variables are visible to watches set at either the module level or procedure level, assuming that the module is the same one in which the variable is defined and that the procedure is contained in that module. For more information, see the section titled "Module-Level Variables."

17. **A**. A Public form variable is essentially a property of the form, making it globally accessible throughout the program. For more information, see the section titled "Global Variables."

18. **B**. Private makes it a module-level variable. For more information, see the section titled "Module Scope."

OBJECTIVE

This chapter helps you prepare for the Visual Basic 6 exam by covering the following objective and subobjectives:

Implement project groups to support the development and debugging process.

- **Debug DLLs in process.**
- **Test and debug a control in process.**

. Since version 5 of VB, developers have been able to use *project groups*. A project group enables you to manage more than one VB project in a single session of the VB IDE.

. The most obvious advantage of project groups is that you can quickly move between related VB projects and manage them as a unit.

. The most powerful consequence of this facility for working with more than one project at a time is the possibility of fully testing and debugging VB components that require a client application to run—in other words, COM components such as ActiveX DLLs and ActiveX controls.

. Since neither ActiveX DLLs nor ActiveX controls are intended to run standalone, you can't test them by themselves: They always need some client application to run and exercise their features. You can therefore create a VB project group that contains your ActiveX DLL or ActiveX control project, as well as one or more test client applications.

. When you run a test client project at design time, the IDE automatically makes the ActiveX DLL or ActiveX control in the same project group available in the test application's environment. You can therefore program the test client to use the ActiveX DLL or ActiveX control as if it had already been compiled and distributed in the client's environment.

CHAPTER 19

Implementing Project Groups to Support the Development and Debugging Process

OUTLINE

Understanding Project Groups **899**

Creating Project Groups 900
Building Multiple Projects 902

Using Project Groups to Debug an ActiveX DLL **903**

Setting Up a Sample Group 903
Debugging Features in Project Groups 906

Using Project Groups to Debug an ActiveX Control **907**

Chapter Summary **909**

STUDY STRATEGIES

- Know how to create a project group that contains more than one VB project (see Exercise 19.1).
- Know how to change a project group's startup project and why this is useful (see Exercise 19.2).
- Know how to debug an ActiveX DLL in a project group (see Exercise 19.3).
- Know how to debug an ActiveX control in a project group (see Exercise 19.4).
- Be familiar with how to handle errors when testing ActiveX projects in the design-time environment (see Exercise 19.5).

INTRODUCTION

The concept of a project group enables you to work with multiple projects open at the same time. In versions of VB before versions 5 and 6, you could only accomplish this by using multiple instances of Visual Basic, making it difficult to work with the projects as one complete system. Working with multiple design-time projects at once was actually impossible in versions 3 and below, because these versions did not enable you to have more than one instance of VB running at a time!

As many of the systems built with Visual Basic 6 are component based, using multiple ActiveX projects, it is very important to be able to work with all the components in one development environment.

This chapter covers the following topics:

- What exactly project groups are, and how you can use them.
- How to use project groups in component-based development.
- How to debug multiple-project applications using project groups.

NOTE

Other ActiveX Control Debugging Techniques Not Discussed Here
Because this chapter deals with project groups, it does not discuss other techniques available in VB6 for debugging ActiveX controls. For more discussion of ActiveX control debugging, see the section titled "Testing and Debugging an ActiveX Control" in Chapter 13, "Creating ActiveX Controls."

UNDERSTANDING PROJECT GROUPS

Developers often work with multiple projects simultaneously. The two most common reasons to group projects together are

- The projects depend on each other (such as an application and an ActiveX control that it uses).
- The projects are multiple, independent parts of a large development project (such as an order-entry system and an HR application both being built for the same company).

Project groups have become necessary because each Visual Basic project does not exist in a vacuum, but rather must coexist with other applications. As the software industry continues to mature, you will see more and more development with multiple, interdependent projects.

Project groups provide several benefits over the more traditional single-project model:

- All the individual projects you have placed into a group are opened and closed together, reducing the time spent getting ready to work and the time getting ready to quit.
- Opening all the projects together enables you to work with them inside a single instance of Visual Basic.
- If the projects are directly related (through references), you can execute and debug them together (as discussed in more detail in the section titled "Using Project Groups to Debug an ActiveX DLL").

The individual projects are not modified just by being part of a project group. They are still stored individually on disk. The group file (*.VBG) merely contains links (using relative paths) to each member project (*.VBP). The contents of a sample *.VBG group file are shown here:

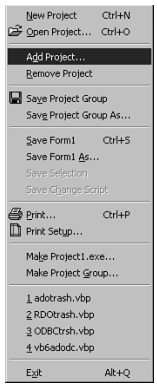
```
VBGROUP 5.0
StartupProject=project1.vbp
Project=Project2.vbp
```

Each project can belong to a number of groups, in combination with other projects, and can still be opened directly, as an individual project, at any time. The only limit on combining projects is that a project can only be in a project group once; multiple copies of a given project within the same project group are not possible.

Creating Project Groups

There is no special procedure for creating a project group on its own: You just start with a standard Visual Basic project, and then add other projects to it. By adding a second project, you implicitly create a new group. You add and remove projects in the group with the File menu's Add Project and Remove Project options, as shown in Figure 19.1.

Selecting the Add Project command brings up the same style of dialog box as Visual Basic 6 shows at startup, enabling you to create a new project. Select one from a list of recently opened projects or select any other existing project through a standard directory browser, as shown in Figure 19.2.



, FIGURE 19.1
Visual Basic 6 includes Add Project and Remove Project commands for working with project groups.

The Remove Project command disassociates the currently selected project with the project group, and is also available through the right-click context menu in the Project Explorer. Even if you remove projects until only one project remains, it is still contained within a project group. If, on the other hand, you remove all the projects in a project group, it is the equivalent of closing the group completely. Exercise 19.1 takes you through creating a simple project group containing two new projects.

When you exit Visual Basic or attempt to load another project (as opposed to adding a project), the current group and its associated projects must be closed. If the group has not previously been saved, this is when you can specify a name and location for the file, as shown in Figure 19.3. You can also save the entire project group through the two menu options: Save Project Group and Save Project Group As. These commands are not available when you do not have multiple projects loaded. It is not necessary to save the group in any particular location, relative to its component projects, but it can make moving the entire set of your development files easier if you store them all in the same area.

After a project group has been saved, it will appear in the Recent tab in any open project dialog boxes, as shown in Figure 19.4, as well as in the Recently Opened Files list near the bottom of the File menu.

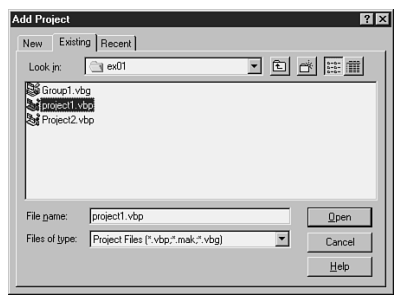


FIGURE 19.2▲
Using a standard directory browsing dialog box to open an existing project.

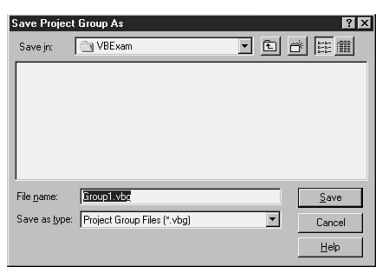


FIGURE 19.3▲
Saving a project group file (*.VBG).

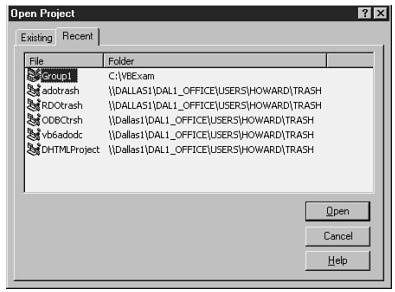


FIGURE 19.4.
Group Files appear in Visual Basic's Recent tab.

Building Multiple Projects

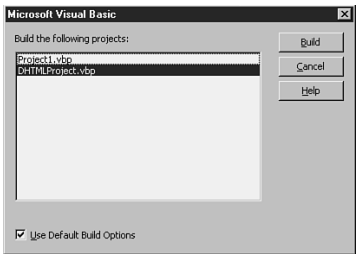


FIGURE 19.5▲
The Make Project Group dialog box enables you to select which projects to build.

Visual Basic makes one other important modification to its menus when a project group is loaded: the addition of the Make Project Group command under the File menu. This will perform a build of every project contained within the current project group, which is sometimes more convenient than selecting and building each project individually. Selecting the Make Project Group option brings up a dialog box, as shown in Figure 19.5, where you can select which projects you would like to build.

The option at the bottom of this dialog box, Use Default Build Options, is very important. If it is selected, Visual Basic will just go ahead and build each of the selected projects using whatever settings you used last, without displaying any dialog boxes to enable you to change these build settings. If you have never built a project before, and this option is selected, the system will use the standard defaults shown for a first build (save in current folder, name is same as project name, and so on). If you do not have this option checked, the standard build dialog box will come up for each project in turn, prompting you to either accept or change the current settings. If you are using the Make Project Group command with the intention of starting all your projects compiling while you go away and do something else, you should ensure that this option is checked.

Another important difference when you have multiple projects loaded is that you need to let Visual Basic know which one is the main, or *startup* project. This project is the one that will run when you press F5 or choose Start from the Run menu. All other projects are only run if they are *instantiated* by that main project. A project is considered instantiated by another project if it is started through some type of ActiveX call, such as CreateObject .

The first project placed in a group is, by default, the startup project, but this can be changed through the Project Explorer. If you right-click within any of the projects in the Project Explorer window, you will see the menu option Set as Start Up (see Figure 19.6). If you select that option, the project that was currently selected is now designated the startup project. To indicate its status, the startup project's name is shown in bold in the Project Explorer.

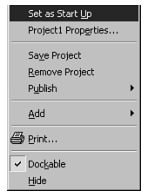


FIGURE 19.6
Set this project to be the startup project.

Generally, the choice of which project should be the main project is obvious to the developer. It is the one that calls or references all the others. It is sometimes necessary, however, to change the startup project while testing.

Any project that can be run on its own can be a startup project, which means any type of project other than an ActiveX control (refer back to Figure 19.6). Exercise 19.2 demonstrates the effect of changing the startup project, using the simple project group you created in Exercise 19.1.

NOTE

Setting the Startup Project Always ensure that your startup project is set correctly when dealing with ActiveX objects. ActiveX DLLs in particular, when accidentally set to be the startup group, can cause some puzzling results. Usually, because the DLL probably doesn't have any code that runs without a client application calling it, there are no visible results of starting the project group. You will see Visual Basic remain in Execution mode (Start button grayed, Pause and Stop buttons enabled), but nothing else will happen. This might cause you to frantically look for errors in your main project (the one you intended to be the startup), without finding anything. Remember to always look for the bold project name as proof of the current startup project.

USING PROJECT GROUPS TO DEBUG AN ACTIVEX DLL

In real development, project groups will usually be used with some form of components. When you are creating a system that uses one or more components, especially components that are still in development, a project group can greatly simplify your work. After a component is stable, or when using third-party components, there should be no need to have their code available and a simple reference to the component should suffice. To illustrate the use of project groups when developing with components, this section walks you through the creation and debugging of an ActiveX DLL.

Setting Up a Sample Group

For the purposes of this demonstration, you will set up two projects: one standard EXE that consists of a simple form with one command button, and one ActiveX DLL project.

To create the sample projects, follow these steps:

NOTE

To Which Project Am I Adding? Other than the previously mentioned options, the Project Explorer behaves the same when dealing with a project group as it does for a single project. The project affected by the various Add commands, such as for forms and modules, is always the currently selected one.

STEP BY STEP

19.1 Setting Up Multiple Projects

1. Start Visual Basic. The Startup dialog box appears, as shown in Figure 19.7.

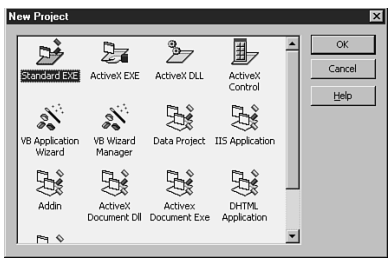


FIGURE 19.7.
Selecting a new Standard EXE project through the Visual Basic 6 Startup dialog box.

2. From the dialog box, select the Standard EXE icon.
3. This project will include a single form, named Form1. Change the following properties of Form1 to the indicated values:
   - Change Name from "Form1" to "frmSample ".
   - Change BorderStyle to 3 - Fixed Dialog .
4. On the form, create a new button. Change the button's name to cmdGetStatus , and its caption to "Get Status ".
5. Place the following code into the form, replacing anything else that may be there:

```
Option Explicit

Private Sub cmdGetStatus_Click()
    Dim sTemp As String
    Dim objStatus As New clsStatus

    sTemp = objStatus.Status

    MsgBox sTemp, vbOKOnly, ÒStatusÓ

End Sub
```

The code itself is very straightforward: It takes a string value from the objStatus object and displays it in a standard Visual Basic message box.

NOTE

No Startup Dialog Box Appears at the Beginning of a VB Session If the dialog box shown in Figure 19.7 does not appear when you start Visual Basic 6, you have most likely chosen to turn it off, and a default, empty, project is created instead. For the purposes of this example, the project created by default will work just fine. If you wish to have the Startup dialog box appear when you start Visual Basic in the future, you can turn it back on by selecting Prompt For Project under the Environment tab in the Visual Basic Options dialog box, as shown in Figure 19.8.

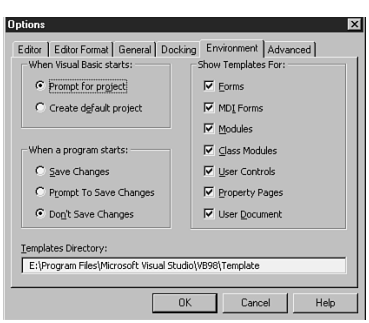


FIGURE 19.8▲
Control whether the Startup dialog box appears through this Option dialog box.

6. Change the name of the project to "Sample".

7. Add a second project by choosing Add Project from the File menu. Select ActiveX DLL from the dialog box that appears, as shown in Figure 19.9.

8. The new project already contains one class module, Class 1, so you won't need to add anything. Change the class's name to clsStatus in its Property sheet; it is referred to it by that name in the cmdGetStatus_Click routine in step 5.

9. Change the new project's name to "DateTime".

10. Place the following code into clsStatus, once again replacing any other code that may already be there:

```
Option Explicit

Public Property Get Status() As String
Dim sTemp As String

    sTemp = Format(Date, ÒLong DateÓ)
    Status = ÒThe Current Date is Ò & sTemp

End Property
```

This property procedure, like the preceding code sample, is simple. It puts together a simple string and returns it to the calling program.

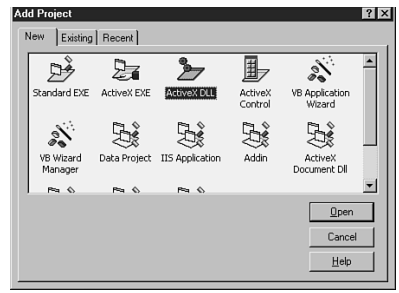


FIGURE 19.9▲
Adding a new ActiveX DLL project to a project group.

After you have completed all the preceding steps, the group is almost complete. The code in your Get Status button won't work, however, if you stop now. (Try it, a User-Deþned type not deþned error will appear when you click on the button.) You still need to add a reference to the DLL to your main project so that you can create instances of your new class.

Adding a reference to a project that is in the same project group is not much different from adding any other reference. It is, in fact, a little easier. If you select the main project in the Project Explorer, and then choose the References command from under the Project menu, the standard Visual Basic References dialog box will appear, as shown in Figure 19.10.

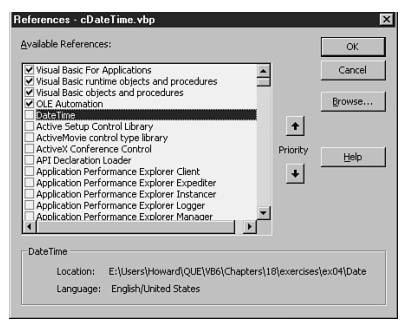


FIGURE 19.10▲
ActiveX objects in the same project group appear at the top of the unselected References list.

Note that the entry for your new project, DateTime , appears directly under the existing, selected references and not in regular alphabetic order. This is not a mistake. Microsoft expects you to commonly be making references to other projects in the same project group. Because this is the main reason why this feature exists, Microsoft has made it a little easier. Another important fact here is that the DateTime DLL was never compiled, but still appears in the list. This would not occur if you didn't have the projects in a group. After you check the DateTime reference and close the dialog box, the main project will be ready to run.

Debugging Features in Project Groups

With the project group properly set up, you are now ready to explore some of the debugging features you can take advantage of when using project groups. When working with multiple components, it is possible to treat them all as independent projects and completely finish one before ever working on another. In such a scenario, you would test and debug each of your components separately. Then, after you have built all the components in to actual DLLs and EXEs, you would combine them with your main application and begin integration testing. If you were always able to do things this way, and there were never any bugs missed, there would be no need to test or debug in project groups. However, that doesn't happen often.

WARNING

Scope of VB IDE Options In general, when dealing with project groups, it is important to realize whether a setting applies to only the current project or to the entire Visual Basic environment. Settings found in the Options dialog box (under the Tools menu), including the Error-Trapping setting, apply to Visual Basic. Those found in the Project Properties and Compile Options dialog boxes are stored with each individual project.

Debugging within a project group is not any different from regular debugging. The same techniques are used—breakpoints, error-trapping, and stepping through code—but they apply across all the projects in the group. You can set breakpoints in any of the projects, not just the startup. If that line is about to be executed, it will pause at that point, just as you would expect. This can be useful in determining exactly when an object is created, initialized, and terminated, and will even work across multiple instances of one project. The Debugging Error-Trapping setting, which can be set to Break on All Errors, Break in Class Module, or Break on Unhandled Errors, behaves as it does with a normal project, but applies across all the projects in a group. It is not possible to set different debugging options for individual projects because these options are really environment-level options, not project-level ones.

Stepping through the executing code is where the true power of project groups becomes clear. It is possible to step, line-by-line, through the code of one project, into the code of another. When dealing with ActiveX components, this could mean hitting an object call in one project that takes you immediately to the code of that object's property or method, possibly passing through some initialization code on the way.

Exercise 19.3 takes you through all three of the debugging options discussed so far, using the sample projects you created in the section titled "Setting Up a Sample Group."

Using Project Groups to Debug an ActiveX Control

Another feature of Visual Basic, the capability to develop your own ActiveX controls, naturally works well with the concept of project groups.

Whenever you develop an ActiveX control, it is a good idea to add some other project to your project group—before you run, test, or debug your control. You really don't need to have anything more than a single form in the test project, but this testing method, of course, always requires the use of a group. This is because the test project has to be separate from the ActiveX control's project.

As in the previous DLL examples, you can use all the debugging features of Visual Basic when working with your ActiveX control in a project group. Using breakpoints, you can cause Visual Basic to pause execution at any point in your control, or in the code of the host project. When stepping through code, the entire project group is treated as one complete, uninterrupted collection of code. You can step directly from a line of code in the host project to a line in your control, transparently. The Error-Trapping settings also function as in the DLL examples: The settings always affect all projects in the group. Exercise 19.4 illustrates the various debugging methods with an ActiveX control.

Overall, there are only a few important differences about working with ActiveX controls rather than DLLs in project groups:

NOTE

New ActiveX Debugging Possibilities With VB6 In VB5, a multiproject group as described in this chapter was the only way you could test and debug an ActiveX control inside the VB IDE. With the release of VB6, you now have other options for the design-time testing and debugging of an ActiveX control or other ActiveX component, as described in the section titled "Testing and Debugging an ActiveX Control" in Chapter 13.

- You do not need to create a reference to the control in your other projects in the same project group. The control is always available on the ToolBox, and a reference is automatically created for you when you place the control on to any of your forms in the test project.
- Although the ActiveX control you're working on becomes immediately visible in the VB ToolBox, you can't use it in the text project unless you have closed the ActiveX control's designer. It will remain disabled on the ToolBox until you close the ActiveX designer.
- Breakpoints placed in key startup code (such as the UserControl's Initialize event) can prevent proper display or functioning of the control during design time.

The first difference is not a problem; it is merely a convenience not provided for ActiveX DLLs, and saves you a few seconds of effort. Because you have to load the control into another project before you can run it, every project in the group references the control by default.

The second and third points are a bit more of an issue, and they exist because of the differences between DLLs and controls. ActiveX controls have a visual interface at design time, so their code is loaded into memory and executed even when you are not running the project.

Visual Basic is not really in Execution mode at this point, however, so it is not possible to debug what happens when the code is run at this point.

If you attempt to open a form containing your control, and the control has an error, you will receive an error message, as shown in Figure 19.11, and the form will not be opened. Fortunately, you have the option to enter Debug mode at this point, enabling you to fix the problem. Exercise 19.5 takes you through a series of steps to demonstrate the design-time effects of breakpoints and errors in an ActiveX control project.

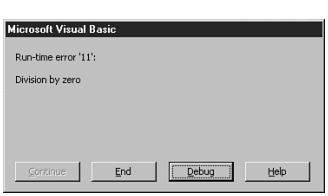


FIGURE 19.11
Errors in your ActiveX control project can cause error messages when placing an instance of your control on a form in the test project.

Because of the obvious benefits of using project groups when you work with any ActiveX object, it is important to understand and begin using them as soon as you can. As it is such a common feature, project groups are almost guaranteed to appear on the certification exam in some form or other. The exercises in this chapter help prepare you for those questions.

EXAM TIP

Project Groups Project groups are extremely likely to be addressed in some form or fashion on the exam. Make sure that you not only understand them conceptually, but that you have some hands-on experience with them. Make sure you work through the exercises in the "Apply Your Learning" section of this chapter.

CHAPTER SUMMARY

In summary, this chapter covered the following topics:

- The meaning of a project group in VB6
- How to create a project group containing multiple projects
- How to debug an ActiveX DLL using a project group
- How to debug an ActiveX control using a project group

KEY TERMS

- ActiveX component
- Design time
- IDE
- Project group
- Runtime

APPLY YOUR KNOWLEDGE

Exercises

19.1 Creating a Simple Project Group

In this exercise, you create a project group consisting of two new standard EXE projects.

Estimated Time: 5 minutes

To create this exercise, follow these steps:

1. Start Visual Basic. The Startup dialog box displays.
2. From this dialog box, select Standard EXE, double-click on the icon or click on the OK button.
3. You should now have one project, named Project1, with a sample form displayed. Choose Add Project from the File menu.
4. The Add Project dialog box appears (refer to Figure 19.9), enabling you to choose between a new project, an existing project, or a recent project. Select the New Project tab.
5. Select the Standard EXE icon. Double-click on that icon or click the OK button.

The exercise is complete. You now have a project group consisting of two projects. Remember that in this case, because the two projects do not reference each other in any way, running this project group is really no different from running the projects on their own. Keep this project group open if you do not want to re-create it before running the next exercise.

19.2 Demonstrating the Effects of Changing the Startup Project

For this exercise, you will be starting with the project group created in Exercise 19.1. If you didn't complete that exercise previously, please do so before beginning this one.

Estimated Time: 10 minutes

To create this exercise, follow these steps:

1. Start with the sample project group (containing two new standard EXE projects) from Exercise 19.1.
2. Select the only form contained in Project1.
3. Change the form's caption from "Form 1" to "First Project".
4. Select the form in Project2.
5. Change its caption to "Second Project".
6. At this point, Project1's name should be displayed in bold (in the Project Explorer) to indicate that it is the startup project. If it is not, make it the startup project by right-clicking on it and choosing Set as Start Up from the context menu that appears.
7. Run the project group by choosing Start from the Run menu, clicking on the Play icon on the toolbar, or pressing the F5 key.
8. A Visual Basic form appears onscreen, with a caption of "First Project". This is because Project1 is the startup project. No matter what you do, the form with the caption "Second Project" will not appear during this run because it is not referred to in any way from the first project.
9. Stop the execution by closing the visible form.
10. Make Project2 the startup group. Right-click on the second project in the Project Explorer and choose Set As Start Up in the context menu. Project2 should be displayed in bold.
11. Run the project group again. This time the form that appears shows the caption "Second Project" because now Project2 is running.

APPLY YOUR KNOWLEDGE

19.3 Debugging an ActiveX DLL (in a Project Group)

This exercise requires the use of the sample projects created in Step by Step 19.1. After you have this project group created and open, you can follow these steps to experiment with various debugging techniques.

Estimated Time: 45 minutes

First you step through a project group, as follows:

1. If you plan to complete Exercise 19.4, "Debugging an ActiveX Control," you should make a second copy of the project created in Step by Step 19.1 before you go ahead with this exercise. You can use this second copy of the project as the starting point for Exercise 19.4.
2. With your sample project group open, ensure that the project Sample is set as the startup.
3. From the Debug menu, choose Step Into, or press the F8 key.
4. The Visual Basic form "frmSample" appears. Click on the button labeled Get Status on this form.
5. At this point, you will enter Break mode as Visual Basic encounters the first executable line of code in this project.
6. Using the Debug menu command Step Into again, step through the lines in the button's Click procedure. When you reach the objStatus.Status call, the next step will take you into the property procedure of your ActiveX DLL.
7. If you continue to step through the code, you will reach the end of the GetStatus procedure and return back to the first project.
8. After you have completed stepping through the cmdGetStatus_Click procedure, and dismissed the message box that is displayed, stop the project execution by closing the form. You may have to minimize the VB IDE window or move it out of the way to see your form running.

Add some code into the Class_Initialize() and Class_Terminate() procedures of clsStatus. It doesn't matter what code, so put something simple such as a Debug.Print command. Step through the project group again and watch what happens. The class's initialization code doesn't fire until you first attempt to actually use the object (by calling its Status property), not when the object is declared (in the Dim objStatus as line).

This happens because Visual Basic is always attempting to conserve memory and other system resources, so it doesn't even create an instance of your DLL until it is referenced. This fact can become extremely important if you place code into the Initialization event of any class. The termination of the object happens later, when objStatus goes out of scope at the end of the cmdGetStatus_Click procedure.

To experiment with the Termination event, try playing with it when the DLL reference is lost. If you move the line declaring objStatus into the Global Declarations area of the form's code, the initialization will occur at the same time but the termination will not happen until the entire form is closed because the object has broader scope. You can explicitly cause the object reference to be destroyed by inserting a Set objStatus = Nothing line at the end of the button's Click event; clicking the button more than once will result in an error.

Next you use breakpoints, as follows:

1. With your sample project group open, ensure that the project Sample is set as the startup, and that it is not currently executing.

APPLY YOUR KNOWLEDGE

2. Open clsStatus by double-clicking on it in the Project Explorer. Find and select the line sTemp = Format(Date, ÒLong DateÓ). Right-click on this line, select the Toggle submenu, and click on the BreakPoint command. The line should be highlighted in red, indicating a breakpoint is on.
3. Run the project group. (Do not step through it, just run it normally.) Once again, the main project's form will appear onscreen.
4. Click on the Get Status button. Almost immediately, Visual Basic will enter Break mode and the code from clsStatus will display.
5. Stop the execution of the project.

Breakpoints are very powerful in testing ActiveX components. It is not unusual, in a large system of DLLs and other objects, to be interested in a procedure buried deep inside a program. Stepping through all the code up until that point would work, but would take much longer than it needs to. Setting a breakpoint right where you need to check some values or pay special attention to the flow of execution can save you a great deal of time.

Now use the Break on Errors settings, as follows:

1. With your sample project group open, ensure that the project Sample is set as the startup, and that it is not currently executing.
2. Open clsStatus by double-clicking on it in the Project Explorer. Find and select the Class_Initialize procedure. Put code into this event that will cause an error, and an On Error Resume Next statement at the beginning of the procedure (example code, as follows):

```
Private Sub Class_Initialize()
On Error Resume Next
Dim j as Integer

    j = 1/0

End Sub
```

3. Remove the breakpoint added in the preceding portion of the exercise, if you haven't already done so.
4. Right-click anywhere in your code and select the Toggle submenu. Select the Break on All Errors setting.
5. Run the project. Visual Basic will break on the line j = 1/0, with a "Division by Zero" error, despite the On Error Resume Next statement. Try the other two settings in the Toggle submenu, restart the project, and note what happens. Next, comment out the On Error Resume Next statement and run the project again with each of the Toggle submenu options.

The important point to remember, when changing the Error-Trapping settings in Visual Basic, is that they affect all projects equally.

19.4 Debugging an ActiveX Control

In this exercise, you create a project group containing an ActiveX control, and then demonstrate various debugging techniques on that control. First create a test project. This exercise builds on the sample projects created Step by Step 19.1. After you have this project group created and open, you can follow these steps to add an ActiveX control to that project group.

Estimated Time: 30 minutes

To create this exercise, follow these steps:

1. Choose Add Project from the File menu. A dialog box displays asking you to choose what type of new project to create. Select the ActiveX Control icon by double-clicking on it.
2. A new blank control project has now been added to your project group.

APPLY YOUR KNOWLEDGE

3. Change the new project's name to "ctrlDateTime", and the UserControl's to "ctrlStatus".
4. Place a Label control on to the control. Position and size do not matter.
5. Rename this label to "lblStatus".
6. Add the following code to the control, replacing any other code that may already be there:

```
Option Explicit

Private Sub UserControl_Initialize()
Dim sTemp As String


    sTemp = Format(Date, ÒLong DateÓ)

    lblStatus.Caption = ÒThe Current Date is
➥Ò & sTemp
    lblStatus.Left = 0
    lblStatus.Top = 0


End Sub

Private Sub UserControl_Show()

    UserControl.Width = lblStatus.Width
    UserControl.Height = lblStatus.Height

End Sub
```

7. Close the UserControl window. Switch to the main project in your group (Sample) and open its single form.

> **WARNING**
>
> **Don't Leave ActiveX Control Design Windows Open When Testing** Always close the control's Design window before switching to other projects. If it is left open, you cannot place the control anywhere else, and the toolbox icon will be disabled.

8. Select the new icon from your toolbox, as shown in Figure 19.12. It will be the very last icon, so you may need to expand the toolbox to see it. This is your new control, all ready to place on this form.

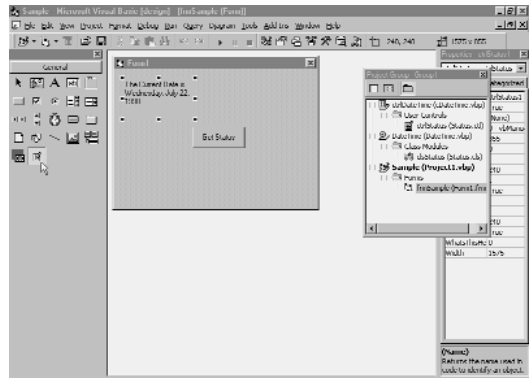


FIGURE 19.12
Selecting your new User Control item from the form toolbox.

9. Place the control on to your form, just under the command button that is already there. Notice that it is already displaying your status string, meaning your control is working.
10. Ensure that the main project, Sample, is set as the startup, and then step into the project group by choosing Step Into from the Debug menu.
11. Almost immediately (before frmSample appears), Visual Basic will break at the UserControl's Initialize procedure. From this point, you can step through the control, moving back and forth between it and the main project.
12. Continue to experiment with the control in Debug mode.

APPLY YOUR KNOWLEDGE

19.5 Errors in User Controls During Design Time

In this exercise, you continue using the project group created for the preceding exercise.

Estimated Time: 15 minutes

1. Ensure that the project group is not running by choosing Stop from the Run menu.
2. Open the code for ctrlStatus . Add a breakpoint inside the UserControl_Initialize() event procedure, at any point, and then close the Designer window for ctrlStatus .
4. Double-click on frmSample (from the main project) in the Project Explorer.
5. Notice how you immediately switch into Debug mode at this point, even though you were not really ever running your project. Continue the execution (F5).
6. Remove the breakpoint and open the form again to see how it should work.
7. Return to the control's initialization routine and add the following lines of code (anywhere):

```
Dim j as integer
j = 1/0 Ôwill Cause a division by zero error
```

8. Open frmSample again, noticing that an error message displays. You are given the option to debug your code.

In both cases, the problem is the same: Visual Basic hits a breakpoint or an error and cannot continue executing the control's code. This causes it not to reach the Show event procedure and not to display the status information.

Review Questions

1. You are building an ActiveX control for use in an existing Visual Basic system, and you have added the control's project to the same group as your existing program. Do you need to add a reference to the control before you can use it in your existing program?
2. What if, instead of an ActiveX control, you have a similar situation with an ActiveX DLL component—do you need to add a reference to the component before you can use it in your existing program in the same program group?
3. How do you temporarily register an ActiveX component that is running from the VB IDE in the Windows Registry so that you can test client applications against it?
4. You have a project group containing three projects, the main program (a standard EXE project) and two ActiveX DLLs. Is it possible to set up Visual Basic to break on errors in your main project, but not on errors in the DLLs?
5. In the project group described in question 2, what project begins to execute first if you choose Start from the Run menu?
6. You have a main project that creates two instances of the same ActiveX DLL while it runs. How many copies of the ActiveX DLL project do you need to have in your project group so that you can use breakpoints?

Exam Questions

1. What type(s) of projects can be startup projects? (Select all that apply.)

APPLY YOUR KNOWLEDGE

A. Standard EXE

B. ActiveX DLLs

C. ActiveX EXEs

D. ActiveX controls

E. ActiveX documents

2. When attempting to debug a project group containing project groups, which projects are effected by the Break on All Errors setting?

   A. All non-ActiveX projects

   B. The startup project

   C. Whichever project was selected when the option was set

   D. Any project except ActiveX controls

   E. All projects

3. Before you can make calls to an ActiveX DLL project in the same project group, what must you do? (Select all that apply.)

   A. Compile the DLL

   B. Check the DLL off in your project's reference list

   C. Make sure the DLL is the startup project

   D. Nothing

4. If you lose or delete your project group file (*.VBG), what information have you lost? (Select all that apply.)

   A. Projects which belonged to that group

   B. Compile information for each project

   C. The project that was the current startup project

   D. Any references to projects in the same group

5. If there is an error in the Initialize event of your ActiveX control (part of your project group), when will you receive the error message? (Select all that apply.)

   A. When you run the project and a form is displayed with your control on it.

   B. When you open the ActiveX control's project.

   C. When you open (in Design mode) a form containing the control.

   D. Never. ActiveX controls cannot raise error messages.

6. The design-time testing of ActiveX controls (Select the best answer.)

   A. Cannot be done in a project group. You must use another instance of VB.

   B. Can be done in a project group, among other options.

   C. Is not possible. ActiveX controls must be compiled to be tested.

   D. Can only be done in a project group.

Answers to Review Questions

1. No. References to ActiveX controls in the same project group are created automatically for all projects in the group. See "Using Project Groups to Debug an ActiveX Control."

2. Yes. References to ActiveX DLL components that you are debugging in the same project group must be created by using the Project, References menu option. See "Using Project Groups to Debug an ActiveX DLL."

APPLY YOUR KNOWLEDGE

3. Trick Question. You don't need to do anything to create a temporary Registry entry for an ActiveX DLL component that you are testing—VB automatically takes care of that for you. See "Using Project Groups to Debug an ActiveX DLL."
4. No. Visual Basic's Error-Trapping settings apply to all projects. See "Debugging Features in Project Groups."
5. Another trick question. You can't tell which project executes first; it will be whichever project was set to be the startup project. See "Understanding Project Groups."
6. One. You can create as many instances of your ActiveX components as you wish, but you can never include more than one copy of the same project in a project group. See "Understanding Project Groups."

Answers to Exam Questions

1. **A, B, C, E**. Any type of project can be a startup project, with the sole exception of ActiveX controls. ActiveX controls always need another project to host them before they can be executed. For more information, see the section titled "Understanding Project Groups."
2. **E**. The Break on All Errors setting is a Visual Basic environment setting and has no connection to any individual project. For more information, see the section titled "Debugging Features in Project Groups."
3. **B**. (Check the DLL off in your project's Reference list.) When running in the Visual Basic environment, you do not need to compile any of your ActiveX projects before you can run them. The calling project has to be the startup project, not the DLL itself. When working with ActiveX controls, you do not have to do anything (except place the control on your form), but ActiveX DLLs and EXEs need to have a reference set. For more information, see the sections titled "Understanding Project Groups," "Using Project Groups to Debug an ActiveX DLL," and "Using Project Groups to Debug an ActiveX Control."
4. **A, C**. Reference and compile information are all stored with each individual project and stay intact when the project(s) are opened outside of the project group. For more information, see the section titled "Understanding Project Groups."
5. **A, C**. ActiveX controls are executed when they are displayed, both in Design and Execution modes. For more information, see the section titled "Using Project Groups to Debug an ActiveX Control."
6. **B**. The design-time testing of an ActiveX control can be done in a project group, among other options. In VB5, project groups were the only method you could use, but this is no longer true. For more information, see the section titled "Using Project Groups to Debug an ActiveX Control."

OBJECTIVES

The exam objectives discussed in this chapter cover the two main ways in which you can manipulate the VB compiler's behavior when it creates your VB application from your source code.

Given a scenario, select the appropriate compiler options (70-175 and 70-176).

. You can specify general behavior of the compiler, to optimize your source code in particular ways. The VB IDE provides a dialog-driven interface that you can use to select among these options.

Control an application by using conditional compilation (70-175 and 70-176).

. You can direct the compiler to make logical decisions about optional compilation of different parts of your source code by using compiler variables that will control whether some sections of your code are compiled.

CHAPTER 20

Compiling a VB Application

OUTLINE

P-Code Versus Native Code **920**

Native Code 920
P-Code 924

Understanding When and How to Optimize **925**

Compiling to P-Code 926
Compiling to Native Code 928
Using Compile on Demand 941

Understanding Conditional Compilation **942**

Preprocessor Directives 943
Types of Expressions 945
Compiler Constants 947
Applications and Styles 952

Chapter Summary **956**

STUDY STRATEGIES

- Memorize the significance of each of the compiler options available in the Compile tab of the Project, Properties menu dialog box in the VB IDE.
- Experiment with the compilation of the same EXE with different compiler settings, as described in the section in this chapter titled "Results of Basic Optimization" and in Exercises 20.1 and 20.2.
- Experiment with compiler directives and conditional compilation as described in the section titled "Understanding Conditional Compilation" and in Exercise 20.4.
- Become familiar with the sample application included with your VB installation in the project "Optimize.VBP." Experiment with this application.

INTRODUCTION

Visual Basic includes a native-language compiler. This compiler can translate the Visual Basic source code that programmers can read and understand into executable programs consisting of native machine code that computers can read and understand.

Versions of Visual Basic before version 5 also produce executable files, but the code in those executables does not consist of native machine code. Until versions 5 and 6, executable program files produced by Visual Basic consisted of something generally known as *pseudocode*, or *P-Code*, rather than native machine code. Native code provides a performance advantage over P-Code that VB programmers have wanted for a long time.

As if it weren't enough to include a native code compiler, Microsoft also gave VB an optimizing compiler. In addition to the performance advantage inherent in native code compilation, the optimization switches enable discerning VB programmers to tweak their applications for even greater performance.

However, Microsoft still allows the programmer to choose P-Code compilation for any project.

Although the primary purpose of source code is to provide clear instructions for the CPU, it is occasionally useful to be able to talk to the compiler too. When necessary, this permits you to give the compiler special instructions about how it should produce the code that finally gets sent to the CPU.

Visual Basic enables the programmer to talk directly to the compiler itself by embedding instructions directed to the compiler in VB source code.

This chapter discusses the consequences of choosing different compiler options in a VB project and also discusses how you can manipulate the compiler directly by using compiler directives.

Specifically, this chapter covers the following topics:

- P-Code versus native code
- When and how to optimize
- Using compile on demand

- Conditional compilation defined
- Preprocessor directives
- Preprocessor constants
- Applications and styles

P-Code Versus Native Code

. Given a scenario, select the appropriate compiler options.

You should be able to explain the essential differences between P-Code and native code on the certification exam. You also need to know the advantages that each type of compilation provides.

Native Code

If you are familiar with other language compilers—such as Microsoft Visual C++, Borland Turbo Pascal, or any of the various assembly language compilers available for a given processor chip—you know that a language compiler turns a programmer's source code into native machine code that is linked into an executable program file.

Before version 5 of VB, the Visual Basic compiler did not generate the machine code common to other compilers. Instead, it could only create something called pseudocode, commonly abbreviated to P-Code.

To understand how P-Code differs from machine code, let's take a look at the difference between programs based on interpreted languages and programs based on compiled languages.

At some level, every computer program can be said to consist of nothing but the source code used to write it. Programmers write statements in a high-level language (such as Visual Basic, COBOL, or C++). If you assemble a series of these statements that collectively are supposed to do something useful, you have a computer program. Before the computer can do anything with a program, however, these high-level language statements must be translated into something the computer can understand: machine code.

Machine code instructions govern the most fundamental tasks performed by a CPU. For Intel chips, those basic operations include things such as data transfer, arithmetic, bit manipulation, string manipulation, control transfer, and flag control. The format of an instruction falls into two parts; One part identifies the operation code, while the other identifies the address in the computer's memory at which the data to be used in the operation is stored.

Machine code varies with each processor, however. The Motorola chips used in Apple computers, for example, don't respond to the same set of instructions used by the Intel chips found in computers that run Microsoft Windows. For a given program to run on a computer, it must be converted into the machine code appropriate for that computer. The code used by a particular processor chip is also known as its *native machine code*.

Until a program is translated from the language used by the programmer into native code, the computer can't do anything with the program. The difference between an interpreted program and a compiled program is the point at which this translation occurs.

With a compiled program, the process of producing an executable file from your source code takes two basic steps. The first step is the *compile step*. If you choose to compile to native code, the compiler produces a series of intermediate files from your source code. These intermediate files are commonly called *object files*, and many compilers (including the VB compiler) give these intermediate files an extension of OBJ.

Even though they consist of machine code, the object files themselves can't be used directly by your computer. Because your computer relies on an operating system (for example, Microsoft Windows 9x or Windows NT), another step—called the *link step*—is necessary to produce an EXE file. During the link step, the object files produced by the compiler are linked together with some *startup code* that tells your operating system where to load your program into memory. The result is written to disk in a form that your computer can use.

NOTE

Creation and Destruction of Object Files Ordinarily, you won't find OBJ files in your Visual Basic project directories. That's because VB cleans up after itself after it produces your executable file, DLL, or custom control. It still creates object files, but it deletes them after the link step. If you find an OBJ file in a VB project directory, it may be left over from an earlier attempt to compile the project that was disrupted.

If you are curious to see what usually happens to these object files, run Explorer in a window behind VB. Make sure that you have your project directory displayed in Explorer, and then compile a VB project. You can watch as the object files are created and destroyed just before the end product of your project is produced.

Interpreted Code

In an interpreted language, each individual language statement is converted to machine code on a line-by-line basis. A line is read and translated into machine code. The computer then reads the machine code for that line and executes its instructions. Then the next line is read, translated, and executed, and so on for every line in the source code.

Because this process must be repeated every time the program runs, interpreted languages are generally rather slow. The source code can be saved for re-use, but it must be re-interpreted every time the program runs.

If you have an old version of MS-DOS or PC-DOS, you probably have an example of an interpreted language. The GW-BASIC and BASICA implementations of BASIC that were included with DOS each provided an interpreted environment for running BASIC commands and programs. You could type a line of code and see just what effect it would produce before typing the next line. Despite the slow speed of execution, the real strength of an interpreted language is this capability to interactively test and debug portions of code within the development environment at any time.

A compiled program is executed differently than an interpreted program. When a program is compiled, the entire program is read and translated into machine code prior to execution. The translation from high-level language into machine code occurs just once, and the translation is saved for re-use. Because there is no overhead for the concurrent translation that occurs with an interpreter, programs produced in compiled languages generally execute faster than those produced in interpreted languages.

The speed at which one's code executes isn't always the defining characteristic of an application's speed at runtime, however. If an application is internally responsible for a lot of processing—for instance, if it performs complex mathematical calculations, or processes lengthy loops or large arrays—it certainly will benefit from compilation.

If an application depends on external resources—say, the speed at which a remote database searches and sorts its records—the benefits of compilation will be less apparent, as its internal code execution is less of a factor in the performance of the application as a whole. If it takes several seconds to return records from a remote database in Cleveland, it doesn't matter so much to the person running your program in Cincinnati that compilation to native code cut the time required to build the SQL statement by a few hundredths of a second.

Until the release of version 5.0, Visual Basic did not compile to native code, but it didn't depend on an interpreter, either. Instead, it produced P-Code, which stands somewhere in between.

Understanding Early Pseudocode

Even though programmers commonly refer to the code produced by VB as pseudocode, this term actually has another meaning that predates VB.

Pseudocode is a technique for expressing the operations of a computer program or an algorithm in a natural language, such as English. It was developed as an alternative to flowcharting.

A pseudocode representation of a program generally retains the flow control directives of the computer language actually being used to develop the program, but it replaces the rest of the program with high-level, natural language descriptions of the processes.

Because Visual Basic flow control uses reserved words such as DO, LOOP, WHILE, IF , and so on, here's the pseudocode for a program to read a book:

```
open book to first page
DO
        read page
        turn to next page
UNTIL end of book
```

The implementation details of reading a page and turning pages are not provided, but there is enough information to determine whether any tasks are omitted or out of sequence. It is easy to see that the read page step needs to occur before the turn page step. If these were reversed, the first page of the book would never be read.

This is called pseudocode because it looks sort of like programming code, but it really isn't (the prefix *pseudo* comes from the Greek word for *false*). This kind of pseudocode is much different from Visual Basic P-Code, which bears some resemblance to native machine code, but isn't fully translated.

In any case, the term is commonly used to refer to the pseudo–machine-code produced by VB, so you are stuck with it., If you talk to other programmers who don't work with VB, you may find that their notion of pseudocode differs from yours.

P-Code

As already mentioned, the executable files produced by earlier versions of VB don't consist of native code. For computers to run these programs, it is clear that they must entail some extra overhead for the computer to understand them. Because you have just taken a brief look at how interpreters work, you might fairly expect to discover that there is an interpreter at the heart of VB. VB does not, however, rely only on an interpretive mechanism to perform this decoding at runtime.

Every time you type a new line of code in the VB Code Editor, VB *tokenizes* it as a symbol that represents a series of machine instructions. That is, VB doesn't compile directly to machine code, but it produces a series of tokens that are a sort of shorthand for particular operations.

The process of tokenization occurs when an object is regularly represented by a particular set of signs or symbols. The most familiar form of tokenization is language. Words are not identical with the things for which they stand—for instance, the word *apple* isn't the same thing as an actual apple—but words are understood to be tokens for the things they represent.

A programming language such as Visual Basic is a special form of language. Unlike natural languages, the expressions formed by a programming language literally can be transformed into the things that they represent. Even though the computer can't understand words, programs can be thought of as instructions for a computer, and certain tools (compilers and linkers) turn words into actual computer instructions. Think about turning the word *apple* into an actual apple, and you begin to see how a programming language is distinct from a natural language such as English.

When you compile your program, the source code compiles to P-Code, which consists of a series of these symbols. When you turn your program into an EXE file, VB builds an executable file that contains the P-Code and the necessary executable header and startup code.

When VB code is compiled, however, you don't immediately get actual computer instructions. P-Code itself isn't executable. If a computer were told to take the contents of the P-Code literally, it wouldn't know what to do.

Because the P-Code symbols are not pure machine code, your VB program must look up these symbols at runtime to figure out what machine code corresponds to each symbol. That's the reason VB programmers have historically needed to distribute VB programs with a runtime DLL file. (for example, VB 3.0 programs relied on a file named VBRUN300.DLL.)

When a P-Code EXE file produced by VB runs, the runtime DLL loads to translate the program's P-Code into instructions a computer can understand. Because the Visual Basic language statements have been precompiled to P-Code, VB programs are faster than if they relied on pure interpretation. It still takes longer to translate P-Code into machine code, however, than it does to run native machine code in the first place.

Although P-Code has many advantages over interpreted code, native code offers yet another advantage: It can be optimized. You will need to compile to native code if you want to take advantage of VB's new optimization features. The following section discusses code optimization.

Understanding When and How to Optimize

To produce an executable, you may just select the VB File menu and choose Make. VB also gives you several choices when you are ready to produce an EXE file. These choices depend on whether you care to take advantage of the native code compiler.

In the following examples, you use one of the sample projects included with VB. The project file is Optimize.VBP. Assuming that you installed the sample files and that you used the default directory structure when you installed VB, you will find a separate directory containing the Optimize project under your VB Samples directory.

You can follow along with any project you like, of course, but this particular project also illustrates some of the optimization issues explored during the rest of this chapter. Therefore, it may be helpful to install it if you haven't already done so.

Compiling to P-Code

Yes, P-Code is still a compiler option. After you have loaded a project and are ready to produce an EXE, your File menu might look like Figure 20.1.

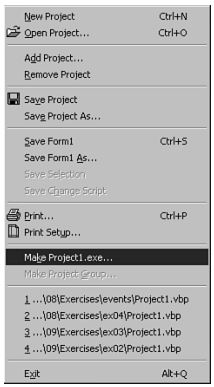


FIGURE 20.1▲
The menu command to produce your EXE file.

When you choose Make, you will see another dialog box that enables you to set various compiler options. To instruct the compiler to produce a P-Code EXE, all you need to do is select the P-Code option from the dialog box, as shown in Figure 20.2.

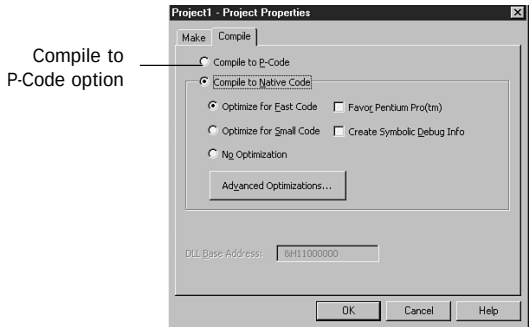


FIGURE 20.2.
Setting compiler preferences at compile time.

If you want to set your compiler options, but you aren't yet ready to compile to an EXE, you can also reach the compiler properties by pulling down the Project menu and choosing Optimize Properties. It looks like Figure 20.3.

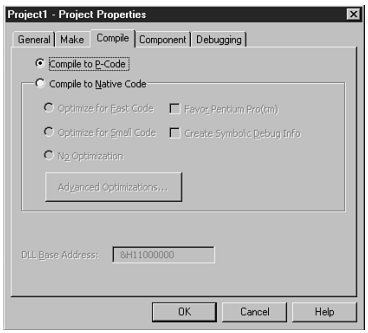


FIGURE 20.3▲
Setting compiler preferences from the Project menu.

You will work from the Project menu for the rest of this chapter. Notice that the Project Properties window, as shown in Figure 20.4, has two additional tabs that don't appear on the dialog box you get when you choose Make from the File menu. Because you are only concerned about setting compiler options now, the extra tabs don't matter.

When you select the Compile To P-Code option button, the other controls on the dialog box are disabled. That's because the other controls set optimization preferences, and P-Code doesn't get optimized in VB6. The EXE generated on compilation behaves just like the P-Code EXEs from earlier versions of VB.

If the capability to compile to native machine code is such a big deal, why does Visual Basic 6 enable you to choose not to take advantage of this great new feature? After all, if native code always executes faster, why would you ever want to bother with P-Code any more? Three reasons come to mind.

First, speed isn't everything. Because it doesn't have to bundle all the native machine code into the EXE (remember that with a P-Code EXE, the runtime file contains the native routines), compiling to P-Code produces smaller EXE files. The advantage of this smaller footprint increases as multiple VB applications are deployed on a single computer.

Theoretically, this can be a compelling factor in favor of P-Code, especially on a computer with a relatively small hard drive. In practice, however, this was more of an advantage for Visual Basic versions 3.0 and earlier, when the runtime files weren't nearly as big as they are now. The VB3 runtime file (VBRUN300.DLL) is about 390KB. The VB4 runtime files take up about 700KB. With VB6, the least you can get away with is about 1.3MB.

With the growth of the runtime files, the size of the EXE itself has become a proportionately less significant factor in distributing applications. Still, the runtime files need to be installed just once to support any number of applications. Therefore, generating a smaller P-Code EXE may still be important to those running short of disk space.

A second reason why P-Code is still an option may be because the native code speed advantage isn't always a huge factor. The degree to which P-Code incurs a performance hit depends on the granularity of the interpreted routines. If even the slightest action entails a call to the interpreter, the slowdown could be immense.

Fortunately, however, the VB P-Code system has evolved to the point that it uses many fairly large-scale instructions, each of which is executed by a big compiled subroutine. VB's built-in functions fall into this category, for instance. (Although some, such as IIF() , are still notoriously slow. Given the speed of the C/C++ ternary operator that IIF() emulates, this is surprising.) After they have been passed to the runtime files, these subroutines run at native speed, so the P-Code overhead need not be substantial.

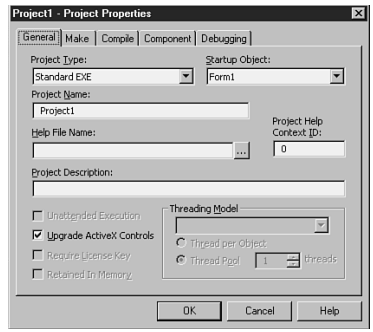


FIGURE 20.4
The Project Properties dialog box uses four tabs to set project options.

Third and finally, a P-Code engine certainly simplifies the distribution of common features among multiple programs. Why do you suppose Microsoft settled on this mechanism for building macro and programming capabilities into so many of its applications via Visual Basic for Applications? A native compiler for VBA would certainly have been interesting, but the convenience of the P-Code engine is compelling, especially when you consider that Microsoft applications such as Word and Excel need to run on Apple platforms in addition to Windows.

Compiling to Native Code

Although P-Code has its advantages, the capability to compile to native code is an advantage that most VB6 programmers put to good use, especially because it permits additional optimizations. This discussion focuses on the basic compiler options for native code.

Visual Basic versions 5 and 6 have the capability to compile native machine code into the EXE files it produces. Like the applications normally produced with other language compilers that produce Windows programs (for example, Visual C++), VB programs require library files. The difference is that a compiler such as Visual C++ can create completely independent EXEs—if the programmer is willing to write all the program's interface code from the ground up. VB can't do that.

Why is native code important? You already know the answer: speed. As hardware has become faster and operating systems have grown more sophisticated, programmers feel the need to produce applications that can keep pace. In a world in which desktop computers featuring 200+MHz Pentium processor chips and 32MB+ RAM are becoming commonplace, and developers routinely produce Internet-enabled programs, your applications absolutely *must* be fast if they are to be taken seriously.

To compile to native code, open the Project Properties dialog box and make sure that the Compile to Native Code option button is selected. When you do, additional choices on the dialog box become available. The following sections look at these choices.

Basic Optimizations

Two sets of optimizations are available. The basic optimization choices are all accessible from the Compile tab of the Project Properties dialog box (see Figure 20.5). A set of advanced optimizations is also available if you click on the Advanced Optimizations button.

Take a look at the basic optimization choices first. By selecting the appropriate option button, you can optimize for fast code, small code, or use no optimizations at all. It is also important to consider the impact these optimization choices have on a sample project.

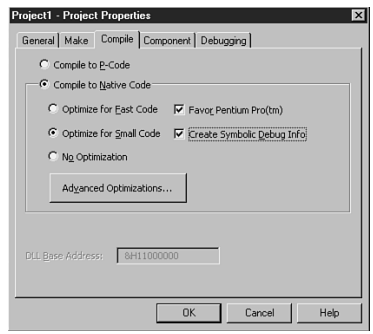


FIGURE 20.5
Compiling to native code basic optimizations.

Optimizing for Fast Code

After you have selected Compile to Native Code, select the first option button underneath it to generate the fastest code possible (the Optimize for Fast Code option). Even if the compiler decides that it needs to produce more machine instruction code to handle certain portions of your application, thereby resulting in a bigger EXE file, the end result ought to be faster than the smaller alternatives.

You may wonder how one set of instructions can be faster than another, if each accomplishes the same end result. Well, you can get to your next-door neighbor's house by walking about 25,000 miles around the earth or by walking a few steps the other direction and knocking on the door. You get the same result either way. A compiler that can optimize for speed just knows how to take those shorter routes.

The VB programmer can perform some kinds of optimizations. VB doesn't short-circuit expressions like C or C++, for example. That is, in a conditional expression such as the following:

```
If iConditionOne < 1 and iConditionTwo < 10 then
        Ô do something
End If
```

VB evaluates both parts of the conditional expression every time. Even if the value of `iConditionOne` were `5`, so that the overall expression must evaluate to `False`, VB would still evaluate the value of `iConditionTwo`. If a C or C++ compiler evaluated this conditional, it would know that the overall expression must evaluate to `False` as soon as it evaluated the first expression. This is called *short-circuiting*.

If a programmer knows that VB doesn't short-circuit logical expressions, it is simple to develop the more efficient habit of coding like this:

```
If iConditionOne < 1 then
    If iConditionTwo < 10 then
        Ô do something
    End If
End If
```

It takes two extra lines of code, but the second fragment executes more quickly than the first when the first condition is false. In this case, knowing how the language behaves makes it possible for you to write smarter code.

Optimization for performance generally occurs in two ways: globally and at the register level. If a compiler employs *global* optimization methods, it tries to change the order in which your program's instructions are executed. This can save time if an action is being repeated unnecessarily, as in a loop such as this:

```
Do
iBadlyPlacedVariable = 1
    Ô more processing occurs here, but
    ÔdoesnÕt change the value of the variable
Loop
```

In this case, the variable is assigned a value of 1 every time this loop repeats. If the loop iterates several thousand times, that's several thousand unnecessary assignments. Clearly, the assignment should have been done outside the loop, but the programmer made a mistake. If it uses global methods, an optimizing compiler can correct this mistake.

Register-level optimization tries to save time by putting data where it can be reached most quickly. Generally speaking, the data your computer needs can be found in one of just three places. In order of increasing speed, these are as follows:

- Physical storage
- Random access memory
- A CPU register

When possible, register optimization tries to put data into a register for quick access.

It takes a relatively long time to find data on a physical storage device such as a hard disk. Even fast hard disks have average seek times measuring in the millisecond range, which is an awfully long time compared to the nanoseconds used to measure RAM chips. Given a choice, it is always better to search RAM than to search a hard disk.

(That's why disk-caching programs are useful: They store recently accessed data from the hard drive in memory for faster access.)

Because they are part of the CPU itself, registers are even faster than RAM. If data or an instruction is in RAM, the CPU has to wait for it to be copied into a register to do anything with it. If it is already in a register, the CPU obviously doesn't have to wait on the RAM access operation. Register optimization occurs when a compiler can reduce the amount of register manipulation necessary to give the CPU what it needs to run the program.

You will see how well the basic speed optimizations work shortly; but first, this section looks at the other basic optimization choices. The explanations of the remaining basic optimizations will be brief; they're quite simple.

Optimizing for Small Code

Selecting the second button causes the compiler to minimize the size of the code it produces. You can easily guess the trade-off between the speed and size optimizations. If selecting fast code produces speed at the expense of a larger EXE, selecting small code may generate a more compact file at the expense of performance. As you have already seen, a shorter list of instructions doesn't necessarily correlate with greater performance.

No Optimization

If you select the third button, the compiler still generates native code, but it will no longer be optimized. Before you decide that Microsoft was asleep at the switch when this option was released, think about what it takes to develop an optimizing compiler. It isn't easy to determine precisely how to handle every conceivable combination of factors governing the use of an optimization. Detecting when it is safe to move an instruction out of a loop or when a particular register value ought to be retained is no mean feat, and it is possible that the optimizer may make a mistake.

In other words, the opportunity to optimize one's code is also another opportunity to introduce a bug. Generally speaking, it is certainly possible to introduce a bug via an optimization switch. The initial release of Visual C++ 5.0 had some problems with its speed optimizations, for example. (The problem was quickly identified and remedied with a service pack.)

Optimized or not, native code still ought to execute faster than P-Code. If you find that your program doesn't run properly when you compile it with optimization features activated, you should try compiling with no optimization. If your program still misbehaves, it is probably not the fault of the compiler—this bug belongs to you! If the program behaves properly after being compiled with no optimization, however, it is just possible that you have found a bug in the VB6 optimization routines.

Favoring Pentium Pro

Whereas the speed/size/no optimization switches are mutually exclusive, the remaining choices on the Project Properties dialog box can be selected for either the speed or size optimization. If you select Favor Pentium Pro, your code will run a little faster on computers with Pentium Pro processors.

It probably doesn't need to be said, but you shouldn't use this option unless you know that your program is likely to be deployed on a machine equipped with a Pentium Pro. Don't worry about breaking anything if you run the program on a computer with a slower CPU. The program will still run on a standard Pentium, or even a 486 or 386. As long as the computer can run a 32-bit Windows program, your program will still run—it just won't perform as well. If you aren't sure where your program will be deployed, don't use Favor Pentium Pro.

Creating Symbolic Debug Info

The last of the choices you can make on the basic optimization dialog box isn't really an optimization at all. If you check the Create Symbolic Debug Info box, the compiler will generate symbolic debug information for your project. This doesn't change the size or performance of your EXE file, but it does generate a PDB file containing the symbol information for the EXE. Bear in mind that the PDB file is of no value to you unless you have a debugger that can use CodeView-style debug information. That rules out the built-in VB debugger. If you want to use the Visual C++ debugger, however, you can.

Results of Basic Optimization

Use the sample Optimize project included with VB. The project file is named Optimize.VBP. Assuming that you installed the sample files and that you used the default directory structure when you installed VB, you will find the Optimize project in the `\VB\Samples\Pguide\Optimize` directory.

The Optimize project is particularly appropriate here because it assesses the speed at which certain VB operations run. The point of the "Real Speed" portion of the project is to show you how to write more efficient routines for string and file manipulation, variable access, and numeric data processing.

You will use the Real Speed tests from the Optimize project to assess the relative impact of VB6's basic optimization scheme in these areas. All you need to do is compile the project four times, producing a separate EXE corresponding to each of the basic options: one EXE each for P-Code, native code optimized for speed, native code optimized for size, and native code with no optimization.

Before you measure performance, take a look at the size of the EXE produced by each option, as shown in Figure 20.6.

Contents of 'H:\USERS\HOWARD\QUE\VB6\Chapters\18\Optimize'

Name	Size	Type	Modified	Attributes
p-code Optimz32.e...	480KB	Application	8/1/98 8:55 PM	A
small Optimz32.exe	528KB	Application	8/1/98 8:56 PM	A
fast Optimz32.exe	532KB	Application	8/1/98 8:56 PM	A
no Optimz32.exe	548KB	Application	8/1/98 8:57 PM	A

FIGURE 20.6
Comparative sizes of the Optimize project EXE.

The P-Code EXE is smallest, and native code with no optimization is largest. Optimizing for speed doesn't produce a file much larger than optimizing for size. Considering the total size of the files necessary to distribute the application, there isn't really a great size difference among the four files. This suggests that compiling for fast performance may be worthwhile as a matter of course. In any case, the price of speed doesn't seem to result in a large enough size penalty to worry about unless you are extremely pressed for disk space.

When you run the Optimize project, its main form looks something like Figure 20.7.

The first test shows how to improve the performance of string manipulation and file I/O. (Strings are built more efficiently outside a loop, and binary file access is faster than random file access.) You shouldn't see any deviation from these general conclusions in your comparison of the four EXE files, but you ought to learn something about the effect of the basic optimizations on string manipulation and file I/O. Because you are working from the same code base and running the tests on a single machine (in the author's case, a Compaq DeskPro SP with 64MB RAM running Windows NT Server 4), any significant differences in the timings assessed by the Optimize project can be attributed to the optimizations introduced by the VB6 compiler.

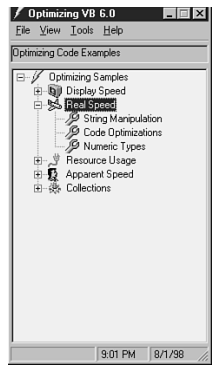


FIGURE 20.7▲
The Optimize application with all nodes closed except for the tests of Real Speed.

Figures 20.8, 20.9, 20.10, and 20.11 show the results of the string manipulation and file I/O tests.

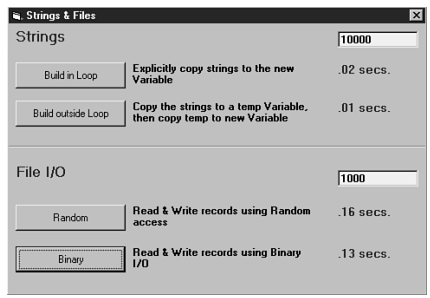


FIGURE 20.8.
Results of the string manipulation and file I/O test when compiling to P-Code.

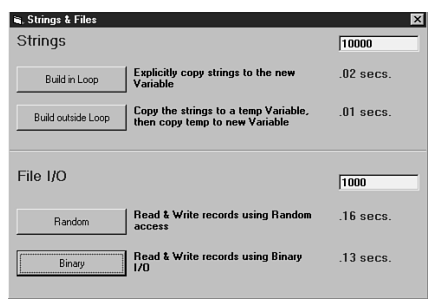


FIGURE 20.9.
Results of the string manipulation and file I/O test when compiling to native code optimized for speed.

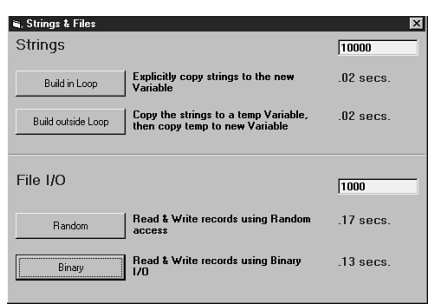


FIGURE 20.10.
Results of the string manipulation and file I/O test when compiling to native code optimized for size.

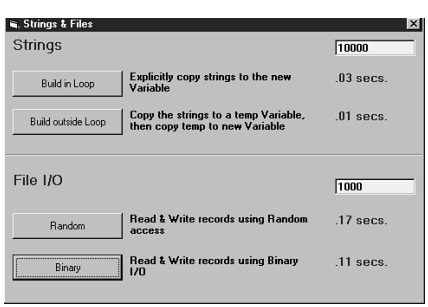


FIGURE 20.11▲
Results of the string manipulation and file I/O test when compiling native code with no optimizations.

Depending on your machine, you may want to specify a different number of iterations than the 100,000 string operations and 10,000 file operations used here. The goal is to run enough iterations to discern measurable differences among the behaviors of each optimization choice.

What conclusions can you draw here? In these tests, it looks as if there isn't a great deal of difference among the results for each optimization type. Because the measurements are all fairly close, even for P-Code, perhaps this merely indicates that the string manipulation code in the VB runtime is already pretty well optimized. The close timings on the file I/O tests are a reminder that other factors besides native code influence performance. No matter how fast the code executes in memory, you still depend on a hard drive (the fastest of which is still relatively slow compared to memory and processor throughput) for accessing data files.

The second test should be more instructive (see Figures 20.12, 20.13, 20.14, 20.15). As its name implies, the code optimization test is more a measure of how VB handles its own code. It is a test of internal factors that shouldn't be influenced either by runtime libraries that have been optimized in advance or by the speed of a physical device.

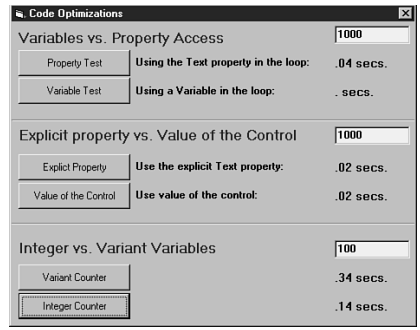
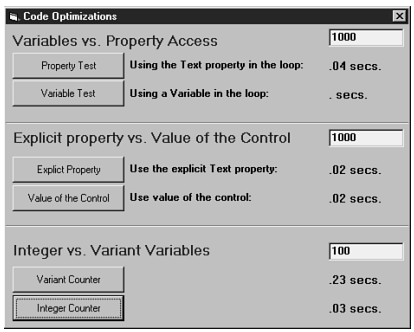


FIGURE 20.12▲
Results of the code optimizations test when compiling to P-Code.



, FIGURE 20.13
Results of the code optimizations test when compiling to native code optimized for speed.

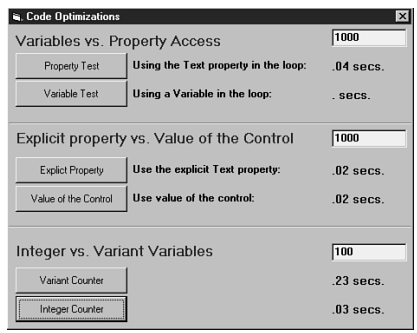
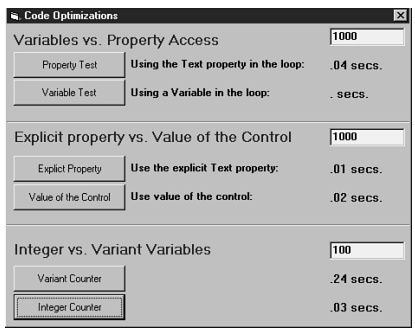


FIGURE 20.14▲
Results of the code optimizations test when compiling to native code optimized for size.



, FIGURE 20.15
Results of the code optimizations test when compiling to native code with no optimizations.

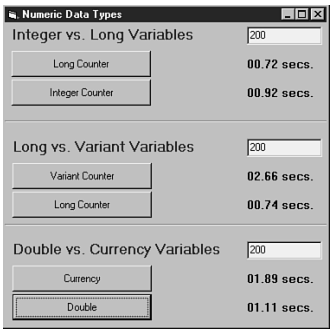


FIGURE 20.16▲
Results of the numeric data type test when compiling to P-Code.

In every test here, you clearly see that P-Code is slowest and optimizing for speed is fastest. The single biggest difference is in the handling of Variant data types, which native code does much more efficiently than P-Code. On the whole, it looks as if optimizing certainly makes a difference, although it doesn't make much difference whether one optimizes for speed or size; both are generally faster than P-Code or unoptimized native code.

The final test enables you to explore performance with numeric data types. The differences here are striking even over a small number of iterations, as shown in Figure 20.16, 20.17, 20.18, and 20.19.

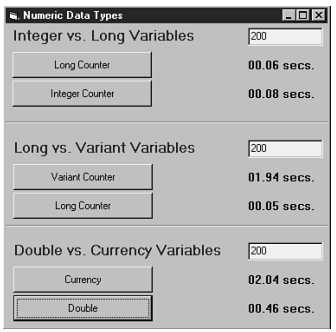


FIGURE 20.17.
Results of the numeric data type test when compiling to native code optimized for speed.

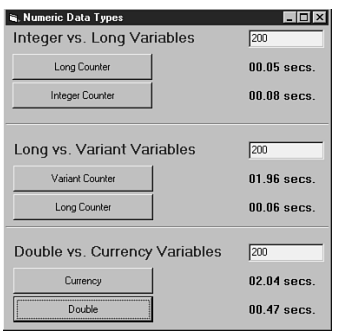


FIGURE 20.18.
Results of the numeric data type test when compiling to native code optimized for size.

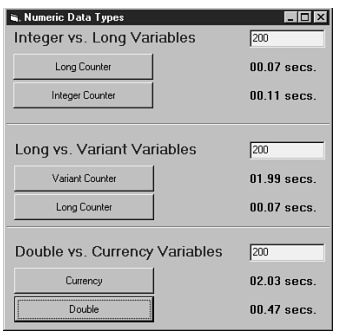


FIGURE 20.19▲
Results of the numeric data type test when compiling to native code with no optimizations.

Again, P-Code is slowest by a big margin in every test. Either optimization is an improvement on unoptimized native code. But once again, there doesn't seem to be a great deal to recommend optimizing for speed over optimizing for size; both seem to execute at fairly similar speeds.

On the whole, it looks as if some real performance benefits are available through the basic optimizations. The next section covers the advanced optimizations.

Advanced Optimizations

These advanced optimization choices appear if you click on the Advanced Optimizations button on the Compile Properties tab, as shown in Figure 20.20.

Next you will see what these optimizations are intended to do.

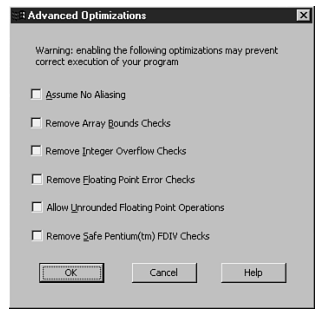


FIGURE 20.20
Options for advanced optimization.

Assuming No Aliasing

A programming *alias* is much the same as an alias used for a person's identity. Consider how you can declare a sub from a C++ DLL in your program. If necessary or desirable, you can give it an alias too:

```
Declare Sub MySub Lib Òz:\MyLibrary.DLLÓ Alias Ò_MySubÓ
➥(Arg1 as string, Arg2 as string)
```

In this case, the alias was necessary because the function name in the library begins with a leading underscore, which isn't legal in VB. The declaration enables you to call the _MyFunction routine in the library by the name MyFunction, which VB will accept. Even though you have two different names for the function, both names refer to the same function.

This optimization is concerned with the kind of aliasing that occurs when the same object in memory is referred to by more than one name. Assume, for example, you use the MySub routine (previously sketched) like this:

```
Dim szName as string
MySub szName, szName
```

According to the declaration statement, MySub receives two string variables as arguments. This example meets that requirement, even though the two variables happen to be the same. When MySub is called, the arguments are passed and MySub does whatever it needs to do with the argument variables. There is nothing special about that, is there?

Ordinarily, no. But recall that, by default, VB passes arguments by reference. Instead of passing by value, in which a routine receives copies of the arguments passed on the stack, the two arguments passed to MySub are actually the memory locations of the string variables.

WARNING

Advanced Optimizations Activating any advanced optimization turns off a built-in VB safety check. Although removing these checks will speed program execution, the extra speed certainly isn't worth it if your programs crash. The basic optimizations are probably safe enough, but you may want to heed the voice of conservatism when using the advanced optimizations:

- Don't use them unless you absolutely must. Any optimization that turns off a built-in safety check is another opportunity to ship a bug.
- You must remember to apply any necessary safety checks yourself. If you don't have time to conduct your own safety checks, refer to the punch line of the first rule.

Because it no longer is working with a copy of the argument variable, a routine that receives an argument by reference has the power to change the original variable by modifying the value stored at its location in memory.

The MySub example is a special case. The same variable is used for both of its arguments, and both arguments are passed by reference. This means that MySub receives the same memory location for each argument. Therefore if MySub modifies the value of one of its arguments, it will unknowingly also modify the other argument as well.

From the perspective of an optimizing compiler, this poses a problem. One of the general forms of optimization mentioned earlier is register optimization, in which values are kept as easily accessible to the CPU as possible. When multiple variables refer to the same memory location, only one instance of the variable needs to be copied to a CPU register. That is, if the address of the first instance of szName is already in a register, copying the address of the second instance of szName is redundant. But how is the compiler supposed to know that the two arguments are really identical?

Register optimization can be confusing when two arguments actually refer to the same thing, so VB ordinarily avoids this problem by not doing this kind of optimization. If you use Assume No Aliasing as an optimization, you are telling the compiler that each variable name you use refers to a value held in a memory location separate and distinct from the values referred to by every other variable name. This opens up the prospect of successful register optimization. If you inadvertently slip in a dual reference, however, you may actually slow down your program.

Removing Array Bounds Checks

Whenever you access or modify an element in an array, VB validates the index values of the array to make sure that you aren't trying to overwrite its bounds. If you have 10 items in an array, you don't want to inadvertently refer to a nonexistent eleventh item, as for instance:

```
Dim aiMyArray (9) as Integer
```

This ordinarily gives you 10 items in the array, as Option Base 0 is the VB default. Because there are 10 items, attempting to access the item at index 10 is a common mistake:

```
Dim iIndex as integer
Do While iIndex <= 10
```

```
    iIndex = iIndex + 1
    aiMyArray(iIndex) = iIndex
Loop
```

Because the index is zero-based, this loop never assigns a value to the item at index 0, and the last pass through the loop attempts to assign a value to a nonexistent eleventh item in the array. By default, VB saves you from attempts like this to overstep the bounds of an array. (It doesn't help you with the overlooked item 0, however.) Naturally, this takes some processing time. You can eliminate this checking by using this optimization option.

To make sure nothing goes wrong, you will want to use the Ubound() and Lbound() functions to ensure that you aren't doing anything illegal. The real savings will occur if you process arrays in loops. Instead of having VB's automatic checking occur with every access, you can conduct your tests outside the loop. This way, you can still be assured that your program is safe and cut down on the array-processing overhead.

The preceding example, for instance, could be rewritten this way:

```
Dim aiMyArray (9) as Integer
Dim iIndex as Integer, iLimit as Integer
iIndex = LBound(aiMyArray) Ô assure start at Þrst item
iLimit = UBound(aiMyArray)
Do While iIndex <= iLimit
    iIndex = iIndex + 1
    aiMyArray(iIndex) = iIndex
Loop
```

Now that the necessary checks are in place, you can take advantage of this optimization without fear of writing beyond the bounds of the array.

Removing Integer Overflow Checks

This is conceptually similar to array bounds checking. Whereas the programmer determines the upper and lower bounds of arrays, the ranges of the basic data types are set in stone. Whenever you perform any calculations on integer types, VB automatically checks to make sure that the resulting values can still be stored in an integer variable. VB will raise an error code if you try to *overstuff* an integer variable with a value that it can't hold.

Just like all the other built-in checks, testing for overflow takes some processing time. If you don't want to spend time on these checks, you can turn off overflow checking.

Removing Floating-Point Error Checks

This is similar to integer overflow, but applies to the nonintegral data types (that is, singles and doubles). VB's automatic tests also check for division by zero. If you test your code and are confident that you aren't performing any disallowed arithmetic actions, you can save the overhead of these tests by deactivating them with this optimization.

Allowing Unrounded Floating-Point Operations

This is another floating-point optimization. It applies when a VB program compares the values of floating-point variables in the evaluation of conditional expressions. Because the variables being compared may not be of the same type, the compiler performs a rounding operation prior to the actual comparison. This enables it to compare like types to one another. For example:

```
Dim singleValue as Single, doubleValue as Double
singleValue = 1
doubleValue = 1
If singleValue = doubleValue Then
    Ô do something
End If
```

Before comparing the values, `singleValue` is rounded up to the same precision as `doubleValue`. Otherwise, the compiler might decide that 1.000000 doesn't equal 1.00000000000000, which would be confusing.

The rounding process takes some extra time, so this optimization enables you to turn it off. You can avoid any problems this may cause by making sure that you compare variables that are already the same type as one another.

Removing Safe Pentium FDIV Checks

This is yet another floating-point optimization. Some of Intel's early Pentium chips had a bug that affected certain floating-point division calculations. By default, VB's mathematical routines guard against the Pentium bug, but doing the math in VB code is slower than letting the processor chip do it for you. If you are confident that your programs won't run on a machine with the Pentium FDIV bug, you may want to activate this optimization.

Using Compile On Demand

Compile on demand doesn't really make your code execute any faster, but it may save you time just the same. The time savings comes from quicker loading of your application when you run it in Debug mode. To use it, pull down the Tools, Options menu and choose the General tab. Figure 20.21 shows this tab.

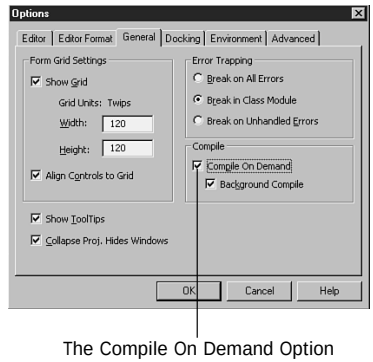


FIGURE 20.21
The Compile On Demand option on the General tab.

The two check boxes in the Compiler group determine how this feature operates. If you check the Compile On Demand box, VB no longer will perform a full compilation of your project before running it in Debug mode. Instead, VB will compile only as much as it needs to start your project. After the application is running, it will then compile your code on an as-needed basis.

If you have two command buttons on a form, for example, the code behind each button won't be compiled unless you click on the button while the application is running. Because you have asked to execute the code for the button's Click event, that portion of your code will be compiled. If there are no errors, the code executes at once. If there are compiler errors, however, a message box informs you of the problem and gives you a chance to fix your code.

The main advantage of compile on demand is that you don't have to get every bug out of your project to test one particular part of it. You may discover that some of your code for a list box won't compile, for instance, but now you don't have to remove that code (or comment it out) just to test the code for a set of command buttons. If compile on demand is activated, you can test each portion of your project independently.

The Background Compilation switch is available only if you also select Compile On Demand. With this switch activated, VB will try to compile additional portions of your application even if you haven't tried to use them yet. The compiler accomplishes this by waiting for idle time while you test the project. It won't notify you of compilation errors unless you have specifically tried to use a feature, but any successful compilations will be available instantly when you are ready to use them.

UNDERSTANDING CONDITIONAL COMPILATION

. Control an application by using conditional compilation.

NOTE

Simplifying the Discussion of Conditional Compilation VB can produce custom controls, DLLs, or EXEs, consisting of either P-Code or native code. Rather than unnecessarily complicating the explanation of the compilation process by referring to all options every time, the rest of this chapter assumes that the programmer is compiling an EXE to native code. None of the conditional compilation concepts change if you change your project options, but the rest of the explanations in this chapter should be more readable if the discussion addresses just one case.

Ordinarily, the VB compiler reads all the source code in a project and translates it into either P-Code or native code that can be used by a computer. It doesn't omit a single line of executable code (naturally, comments aren't compiled), so every instruction that was in the source code also winds up in the compiled EXE, DLL, or custom control.

With conditional compilation, this no longer is the case. Using conditional compilation, it is possible for the programmer to tell the compiler whether to skip or include a portion of code, or to compile one section rather than another. (Reasons for doing these things are discussed later in the section titled "Applications and Styles.") This means that there may be code in the source files that does not get compiled into the EXE for a project.

To understand how this works, it may help to think about the compilation process as consisting of two phases. During the first phase, the source code is scanned to determine which parts of the source are actually supposed to be translated into machine code. The code that fails the test is ignored; the code that passes is written to temporary storage for use in phase two. During the second phase, the code in the temporary storage is compiled into an executable program file.

Because the determination of which code is included and which code is omitted occurs before the compiler actually processes the code, this phase is often called *preprocessing*. Although it was not introduced into VB until version 4.0, other languages have had access to a preprocessor for a long time.

The C and C++ preprocessor, for instance, makes it possible for the contents of one file to be inserted into another (the #include directive), to expand a token into another series of characters as a macro (the #define directive), and even to change the rules of the language temporarily (the #pragma directive). Remember that all these actions occur *before* the code is presented to the compiler. The compiler never sees the original state of your source code; it only operates on the code that has passed the conditional tests put in place by the programmer.

The Visual Basic preprocessor is not as powerful as that found in C or C++—VB can't do the tricks mentioned in the preceding paragraph (not yet, anyway)—but it is still quite useful. The following section takes a look at VB's preprocessor directives.

Preprocessor Directives

A preprocessor gets its name precisely because it operates on source code before the compiler processes the code. The only function of the VB preprocessor is to conditionally evaluate code to determine which parts of it the compiler should process.

The preprocessor uses just the following four conditional flow control directives:

- #If
- #ElseIf
- #Else
- #End If

Aside from the "#" prefix, the preprocessor syntax is just like that of the If, ElseIf, Else, and End If flow control directives in the Visual Basic programming language. The only difference is that If/Then/Else/End If conditional tests used in a program determine the path of execution taken by the code, whereas the #If/#Then/#Else/#End If preprocessor conditional tests determine which code will be included when a project is compiled.

Because any code that the preprocessor screens out is completely absent from the final compiled product, its presence in the source code does not influence the executable program in any way. No size or performance penalty accrues.

The formal syntax for the preprocessor directives is represented like this in the VB Help system (blocks enclosed in brackets are optional):

```
#If expression Then
    statements
[ #ElseIf expression-n Then
    [ elseifstatements ] ]
[ #Else
    [ elsestatements ] ]
#End If
```

At minimum, a preprocessor block must consist of #If /Then/#End If . Optionally, any number of #ElseIf blocks can be included with additional expressions to be evaluated should the expression in the main #If be False . A single #Else block can be included, but is not required. The #Else block has no expressions to evaluate because it is the default block. It is reached only if all the #If and #ElseIf expressions evaluate as False .

The sections marked statements , elseifstatements , and elsestatements represent lines of VB code that are subject to conditional compilation. After the conditional expressions are evaluated, only those lines found in the active conditional block (that is, the lines in an #If or #ElseIf block that evaluates as True , or, if no expression is True , the lines in an #Else block) will be passed to the VB compiler. Besides containing lines of VB code, it is also possible to nest additional preprocessor blocks here.

If you know how VB evaluates a programming If block, you already know how to evaluate a preprocessor #If block. A conditional expression is evaluated on the #If line. Should the conditional expression be evaluated as True (that is, any non-zero value), the set of statements immediately following the #If line will be included in the compiled form of the program, until an #End If (or, optionally, an #ElseIf or #Else) is encountered. If the conditional expression is not True , the code in the #If branch of the conditional block is excluded from the compiled form of the program.

Should the preprocessor find an #ElseIf after an #If expression evaluates as False , the preprocessor evaluates the #ElseIf expression, applying to it the same guidelines as previously described for the #If block. This process continues until the preprocessor finds an expression that evaluates to True , whereupon the code wrapped in that branch will be sent to the compiler.

When no #If or #ElseIf condition evaluates to True , those branches are excluded from the compiled form of the program. The preprocessor then tries to find an #Else branch. If one is present, any code wrapped in it will be included by default in the compiled form of the program.

Again, if you understand VB flow control with an If block, you shouldn't have any difficulty with the logic used in the VB preprocessor. They follow the same rules, with just one exception: A conditional expression and a single action are allowed to appear on a single line in a VB program, such as this:

```
If TestCondition = True Then DoSomething()
```

When a single action is involved, the usual closing End If is unnecessary. (Of course, if more than one action should be taken as a result of a conditional test, this form can't be used—all the actions must appear on separate lines, and the terminating End If is required.)

With the preprocessor, however, the conditional test must be on a different line from any actions, even if there is just one action to perform. To work with the preprocessor, the preceding code would have to be written like this:

```
#If TestCondition = True Then
    DoSomething()
#End If
```

The reason for keeping preprocessor commands on separate lines from other VB actions is due to the way the preprocessor does its job. When it determines what portions of the source code to pass on to the compiler, it does more than just remove the code blocks that didn't satisfy its conditional tests: It also strips out the tests themselves, as well as every other line that begins with a preprocessor directive. After all, the preprocessor commands are not part of your program. (Think about it—they aren't in the Visual Basic programming language, so the VB compiler doesn't know how to compile them.) If a programming command were to appear on the same line as a preprocessor directive—as is the case with the single-line version of a conditional test shown previously—it would also be stripped out of the program.

It may help to think of the preprocessor and the VB compiler as two separate compilers that don't speak each other's language. So as not to confuse them, keep the commands for one separated from the commands for the other.

Types of Expressions

So far, this discussion has focused on how the process of evaluating conditional compiler expressions works. You still need to know more about the kinds of expressions that the preprocessor can handle. Just like any other conditional statement, the preprocessor requires the expressions it evaluates to have a truth value—that is, they must evaluate to True or False, where True is defined as any non-zero value and False is zero. These expressions may consist of the following three components, two of which are probably already familiar:

- Operators
- Literals
- Compiler constants

You probably understand operators already. The same arithmetic and logical operators used in any other conditional statement are available for use in preprocessor conditional statements, with one exception. You can use any arithmetic or logical operator except Is —because Is is a special operator used to compare VB object variables, and the preprocessor doesn't understand VB variables.

Literals are probably familiar by now, too. A literal can be a numeric value, such as 1 or 256, or a text string, such as ÒHello, worldÓ . When comparing values, the Option Compare statement has no effect upon expressions in #If and #ElseIf statements. Conditional compiler statements are always evaluated with Option Compare Text.

Because these tests use only operators and literals, both tests are valid:

```
#If 1 < 2 Then
    Ô do something
#ElseIf ÒMyStringÓ = ÒMyStringÓ Then
    Ô do something else
#End If
```

Not only are both of the tests valid, but it so happens that both are also True . However, only the do something code will be compiled into the executable because the conditional statement on which it depends, 1 < 2 , is evaluated first. The do something else conditional, ÒMyStringÓ = ÒMyStringÓ , is True , but because it is part of an #ElseIf test, it is skipped when any previous #If or #ElseIf at its level in the block evaluates to True .

However, this test is not valid:

```
Dim db1 as Database, db2 as Database
Set db1 = DBEngine(0)(0)
Set db2 = db1
iCounter = 100
#If db1 Is db2 Then
    Ô try this
#ElseIf iCounter > 0 Then
    Ô try this instead
#End If
```

There are two problems here. The first should be obvious from the rule for operators stated earlier: The #If statement uses the Is operator, which is explicitly disallowed.

What's wrong with the second conditional? Because it is True that iCounter is greater than zero, it may seem that nothing is wrong here. The problem is that the preprocessor can't use VB variables, so it has no idea what to do with iCounter. (In fact, that's also a problem with the #If statement's use of the db1 and db2 variables.)

Although these variables make perfectly good sense in the context of a VB program, remember that the preprocessor doesn't speak the same language as the compiler. The preprocessor may be responsible for deciding which code gets sent to the compiler, but it doesn't have to know anything about VB code to accomplish that task.

If you could only use literals in your tests, the preprocessor wouldn't make for a very interesting tool. You could only construct tautologies (statements that are always true) or contradictions (statements that are always false). You could move such statements from place to place in your code, explicitly selecting the lines you want to include by wrapping them with tautologies, and screening the lines you want to exclude with contradictions, but that's a lot of manual labor. It would almost be as easy to wade through the each project's code, manually commenting out any undesired lines on a build-by-build basis.

Fortunately, an easier way exists. Besides operators and literals, you will recall that a third component allowed in conditional compiler tests was mentioned earlier: compiler constants. Compiler constants are the key to doing tricks with the VB preprocessor.

Compiler Constants

Compiler constants have two sources. Some are predefined by VB, and the programmer creates others. The following sections discuss both types of compiler constants.

Predefined Compiler Constants

Visual Basic for Applications predefines two constants for use with the preprocessor: Win16 and Win32. These values are automatically available everywhere in a project (that is, they are global constants).

These constants exist for downward compatibility with projects that were originally created in VB 4.0, which developers could use to develop projects for both 16-bit and 32-bit Windows from the same source code.

In version 4.0 of VB, you could compile for either a 16- or a 32-bit platform. (In all previous versions, 16-bit was the only available platform; in subsequent versions, 32-bit is the only platform.) The values of the Win16 and Win32 constants depended on whether you were compiling on a 16-bit or 32-bit Windows platform. If you were compiling for a 16-bit platform (for example, Windows 3.1), the value of Win16 was defined as True, and the value of Win32 was defined as False. On 32-bit platforms (for example, Windows 9x and Windows NT), the values were reversed: Win16 was False, and Win32 was True.

In VB6, Win32 is always True and Win16 is always False. You no longer need to use compiler logic with Win16 and Win32 in your code. If you maintain code that was originally written in VB4, you may find these constants.

Leaving this compiler logic in your code does no harm in the VB6 environment, but neither does it do any harm to remove the Win16 logic: No future version of VB is ever going to support 16-bit code again.

Declaring Compiler Constants

Besides the predefined constants, you can also define preprocessor constants for yourself to automate your preprocessing requirements. You can declare a compiler constant in the following three ways:

- In code
- In the Project Properties dialog box
- From the command line

Each method produces slightly different results from the other two. First, it is important to understand the mechanics of declaration for each case. After that's clear, the discussion examines how each method behaves.

Declaring in Code

There is one more preprocessor directive to remember in addition to the #If, #Else, #ElseIf, and #End If talked about earlier: #Const. Not surprisingly, the #Const preprocessor directive is similar to the VB keyword Const. Const enables you to define a name to use in place of a constant value in your VB code, such as this:

```
Const MAX_LINES as Integer = 60
```

Then, whenever you need to use this value, you can type MAX_LINES rather than 60. This makes your code easier to understand because your code can use a meaningful name rather than a magic number. For instance,

```
Do While iCount <= 60
```

is harder to understand and more difficult to maintain than the following:

```
Do While iCount <= MAX_LINES
```

In the first case, it is not clear what makes 60 a special value; and if the value ever needs to be changed, every conditional that uses it must be changed manually. In the second case, the use of a meaningful constant name clarifies the meaning of the value; and if it ever needs to be changed, a single change to the constant definition propagates the change to all instances of its use.

#Const works in much the same way for the preprocessor. For instance:

```
#Const TESTING = 1
```

Just like a programming constant, a compiler constant must be defined only once. The syntax for compiler constants is slightly different, however, from programming constants. Although you can optionally specify a data type with a programming constant (if you don't, the constant defaults to a Variant, or whatever the programmer has specified as a default type via the family of Def statements—for example, DefInt, DefLng, and so on), it is not possible to specify a data type for a preprocessor constant. All preprocessor constants are treated as if they are Variants of type string.

You may notice something slightly odd about this chapter's treatment of compiler constants: Even though you generally use them as Booleans, the sample constants you will see in this chapter have all been assigned numeric values. In fact, the values True and False can be used for compiler constants too. The reason this chapter avoids doing so is that the value for True is explicitly defined as Ð1, which can lead to some puzzling results with the preprocessor:

```
#Const VexingVariable = True
#If VexingVariable = 1
    MsgBox ÒYou WonÕt See MeÓ
#End If
```

The message box won't appear because the preprocessor sees the test as `Ð1 = 1`, which it concludes is `False`. If you are accustomed to thinking of any non-zero value as `True`, this can be confusing. That is why this chapter assigns unambiguous literal values to compiler constants rather than Boolean values.

Once again, remember that the preprocessor and the compiler don't speak the same language. (In this case, perhaps it is more appropriate to say that they don't share the same name space.) Just as the preprocessor can't use a variable you have declared in your code, your code can't use compiler constants anywhere but in a preprocessor directive. This is illegal:

```
Dim ProgramVariable as Integer
#Const NothingButACompilerConstant = 1
If ProgramVariable = NothingButACompilerConstant Then
➥DoSomething()
```

When you try to compile code like this with Option Explicit, VB will tell you that you haven't declared `NothingButACompilerConstant`. By now you have certainly noticed the pattern. So long as you don't mix the stuff you do on lines beginning with pound signs with any of your other code, you should not have any trouble. Just keep track of those pound signs.

Declaring in the Project Properties Dialog Box

If you pull down the Project menu and choose Properties at the bottom of the menu list, the Project Properties dialog box appears. If you select the Make tab on the dialog box as shown in Figure 20.22, you will see a field for Conditional Compilation Arguments.

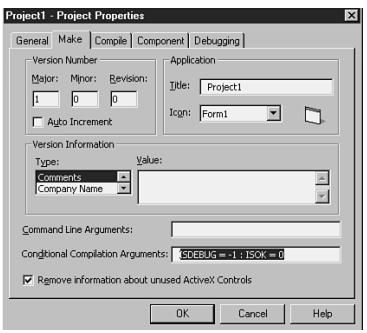


FIGURE 20.22
The Make tab of the Project Properties dialog box.

To define a preprocessor constant here, you don't use `#Const`, but the syntax is otherwise the same as for doing so in code: Type a name for the constant, an equals sign, and the value you wish to assign to the constant. As the illustration shows, you can define more than one constant in the Project Properties dialog box by separating them with colons. (When you realize that the colon also acts as a line separator in VB code, this makes pretty good sense.)

Declaring in the Command Line

You can also use the VB command line to set up preprocessor constants. If you want to compile a project from outside the development environment, you can start the compilation process from a command line with the `/make` switch, adding the `/d` switch to enter compiler constants. For example:

```
vb.exe /make ProjectName.vbp /d VERSION=2:CLIENT=ÓPointy-
➥haired bossÓ
```

You don't need to leave a space between the /d switch and the constant, but it makes the example easier to read. Notice how the example defines multiple constants by separating them with colons.

Scope and Persistence

The scope and persistence of a compiler constant depends on where it is declared.

If the #Const directive is used to define a compiler constant in code, that constant is Private to the module in which it is defined. This means that if you want to use the same constant in multiple modules, it must be defined in each module; you can't use #Const to create Public compiler constants. Naturally, compiler constants defined in code persist between sessions. That is, they won't go away unless you explicitly remove them from your source files.

If you need to use a Public compiler constant, you can define it either in the Project Properties dialog box or on the command line. Constants defined in either way are Public to all modules in a project. If you need to use the same constant throughout an entire project, this is obviously more convenient than manually entering it into every module and form.

Why are there two ways to define Public compiler constants? The scope is the same using either method, but the persistence differs. If you use the Project Properties dialog box to define a constant, it is saved with your project. If you close the project, any compiler constants you have defined as a property of the project will still be there the next time you open the project.

When you specify a compiler constant on the command line, it applies only to the instance of the project that you are running at that very moment. The primary reason for this is to enable you to use a different value for a Public compiler constant that you have already defined in the Project Properties dialog box. The value you use on the command line temporarily overrides the stored value, but doesn't erase it.

Consider that a Public compiler constant is defined under Project Properties such that USER=ÓWallyÓ, and that the following is entered on the command line:

```
vb.exe /make ProjectName.vbp /d USER=ÓAliceÓ
```

Any conditional compiler tests will substitute the value ÒAliceÓ for the USERconstant during the current debug session. The next time the project is run, however, USERwill still be defined as ÒWallyÓ (unless it is overridden on the command line again, of course). Values set on the command line don't persist between sessions.

What happens if you use the same name for a constant both in your code and in the Project Properties dialog box? The same thing that happens when a local variable has the same name as a global variable: The local variable overrides the global variable. The value of the compiler constant set in your code will override the Public constant set in the dialog box.

If BUDDYLIKESSOCKS = 0 is specified in the dialog box, for example, and #Const BUDDYLIKESSOCKS = 1 is specified in a code module, the value of BUDDYLIKESSOCKSwill be 1 in the code module and 0 everywhere else. This enables you to override Public compiler constants on a module-by-module basis.

Applications and Styles

This discussion to this point has focused on the mechanics of how conditional compilation enables you to selectively include and exclude parts of your code from the compiled version of a project. Now the focus changes to some applications of these concepts. What follows is hardly an exhaustive list of all the possibilities, but it should at least suggest some ways to use the preprocessor to your advantage.

Using Debug Mode

Conditional compilation makes it easier to remove debugging tests from your code prior to building a release version of a project. Rather than having to manually search and remove (or comment out) tests that may be scattered in multiple modules, you can place your debug code in conditional compiler blocks:

```
    Dim BuggyVariable as String
    Ô ... processing occurs
#If TESTING = 1 Then
    Ô see if THIS is where BuggyVariable blew up
    MsgBox ÒCurrent value of BuggyVariableÓ Ò &
➥BuggyVariable
#End If
```

Naturally, you don't want this to appear in a released product. The key is to define a constant that will be True only when you are debugging, and False otherwise. Visual C++ includes a predefined compiler constant that serves just such a purpose, but Visual Basic requires you to create your own "Debug mode" constant. Recall that there are three ways to define a compiler constant. You could define

```
#Const TESTING = 1
```

in every module with debug code, and then wrap your tests in the following:

```
#If TESTING Then
#End If
```

Of course, this requires you to manually alter each module so that TESTING is no longer defined as 1 when it is time to build a release. One of the reasons to use conditional compilation in the first place is to cut down on that kind of drudgery. Therefore, defining the constant via #Const probably isn't the best choice unless your debugging tests are confined to a single module.

Defining a Public compiler constant, either via the Project Properties dialog box or on the command line, defines a constant for the entire project. Of course, if you use the Project Properties dialog box, you must remember to change the value of TESTING (for example, TESTING = 0) before building your release. This approach has the advantage of leaving you with only one place to make the change. Still, it is possible to forget, especially if you don't visit the Project Properties dialog box regularly. This approach is less risky than having to track down #Const in multiple source files, but there is still a chance that you will release a project that includes your debug code.

What about the command line? Because you have to explicitly define your Public compiler constant with the /d switch before beginning each debug session, this is less convenient than setting the constant with the Project Properties dialog box. However, it also means that you're extremely unlikely ever to ship a version of a project with your debug tests accidentally enabled.

This raises another issue, however: What happens to a conditional compiler test when a compiler constant is undefined?

If you leave the conditional tests in place after eliminating the definition of a constant, you might expect to see some kind of preprocessor error message or warning. In fact, you won't get one.

Recall that preprocessor constants are treated as Variants. If a Variant hasn't been initialized, it has the value Empty. Now, assume that the RELEASEMODE constant in this example has not yet been defined:

```
#If Not RELEASEMODE Then
    MsgBox ÒBut I thought this code wouldnÕt be compiled!Ó
#End If
```

In this case, the message box will appear. A Variant containing Empty is treated as 0, and Not 0 evaluates to True. This means that the message box pops up even if you haven't assigned a value to RELEASEMODE

Because you can't rely on a compiler warning or preprocessor error message to inform you of your mistake (there is no Option Explicit for compiler constants), it is a good idea to take this default behavior into account when you code your tests to avoid surprises. Always test the compiler constant used to wrap your debug code for a non-zero value. (That is, never rely on a test that reduces to ÒMYCONSTANT = 0Ó.) If you use the command line switch to define your debug compiler constant, this will make it impossible to accidentally compile your debug tests into a release.

From a mechanical standpoint, there is nothing more to know about the process of conditional compilation. After you understand the rules for defining and testing constants, all that is left is to dream up new ways to apply them. The rest of this chapter suggests a few ideas to get you started.

Comparing Algorithms

If you are not sure which algorithm is best suited to a procedure, implement the ones you want to compare in a conditional block. For example:

```
#Const SortMethod = 1
#If SortMethod = 1
Ô Insertion Sort
#Else If SortMethod = 2
Ô Bubble Sort
#Else If SortMethod = 3
Ô QuickSort
Ô #Else If etc.
#End If
```

The quantity and state of the data it manipulates may affect the performance of an algorithm. Insertion sorts are fast with small lists, for example, but performance deteriorates as the size of the list increases. If a list is already nearly sorted, a bubble sort can be quite fast.

Conditional compilation to support different algorithms makes it easier for you to test different approaches to programming problems. Depending on the completeness of your implementation, you may also find it useful to be able to adapt your program to the demands of different data sets just by changing the value of a compiler constant and recompiling.

Feature Sets

The same body of code can support different combinations of features. You may want to use the same interface for a database program, for example, but employ it with different data engines. Using conditional compilation, you could produce separate versions of your program for DAO, RDO, and ODBC.

Unlike the database scenario, which requires selecting one approach from among competing alternatives, you can also take an additive approach. If you add fax support to your program, but you only want it to be included in the releases 2.0 and up, you can still produce a 1.0 EXE from your source, as follows:

```
#If ReleaseNumber >= 2 Then
Ô fax support
#End If
```

Just make sure that you specify `#Const ReleaseNumber = 1` when you need to generate a 1.0 copy of your program. It is no match for a full-blown version control system, but conditional compilation can serve as a limited VCS.

Client Specific

If you provide the same program to multiple clients who require slightly different features, you can implement these features in a common body of code by wrapping the client-specific features in conditional compiler blocks that compare a constant to the client name:

```
#If ClientName = ÒClintonÓ Then
Ô ingenious code that cripples the Republican party,
Ô except for that pesky Whitewater bug
#End If
#If ClientName = ÒDoleÓ
Ô ingenious code that cripples the Democratic party,
Ô except for that annoying 1996 bug
#End If
```

CHAPTER SUMMARY

KEY TERMS

- Compiler
- Compiler constant
- Compiler directive
- Executable file
- Interpreter
- Native code
- Object code
- Source code
- Pseudocode

This chapter discussed the following topics related to the compiling of a VB application:

‡ Pseudocode and native code

‡ The meanings of the various compiler optimization options, both the elementary and advanced options, and how to experiment with these options in sample applications

‡ How to use compile on demand when you want to quickly test specific changes to a large project—and how to compile fully if you want to make sure all compiler errors have been eliminated

‡ What conditional compilation is and why it is useful, including the use of preprocessor directives and preprocessor constants and their declaration

APPLY YOUR KNOWLEDGE

Exercises

20.1 Measuring the Benefits of Basic Optimizations

VB6 includes a sample Optimize project. As you have already seen, the project shows the relative merits of various approaches to coding so that you can write more efficient programs.

The time measurements used in the Optimize project all depend on the VB Timer() function. Timer() isn't really as accurate as the Optimize project would have us believe. The text boxes used to display elapsed time are formatted to display hundredths of a second, but Timer() only purports to return time values in whole seconds. Because the speed comparisons among the basic optimization choices were so close in certain cases, some advantages may merely have been apparent due to the inaccuracy of Timer() .

The goal of this exercise is to modify the optimization project to apply a finer measure of performance differences among the basic optimizations by substituting another function for Timer() . Hint: The Win32 API includes several functions that return elapsed time in milliseconds.

Estimated Time: 25 minutes

To create this exercise, follow these steps:

1. Open the Optimize project. Its default location is in the \VB\Samples\Pguide\Optimize directory.
2. Add a function declaration in a standard (.BAS) module. Use the Windows API Viewer if you like, or type in the following:

```
Declare Function GetTickCount Lib Òkernel32Ó
➥() As Long
```

3. Add a new function in the global module:

```
Public Function ElapsedTime(ByVal timeStart
➥As Single, ByVal timeEnd As Single) As
String
ElapsedTime = Format$((timeEndÑtimeStart) /
➥1000, Ò##.###Ó) & Ò secs.Ó
End Function
```

4. In each of the forms used in the Real Speed tests, change the calls to the Timer() function so that they call GetTickCount() instead. One technique to accomplish this might be to do an Edit/Search across the entire project for the string ÒTimerÓ to find all the calls to the Timer() function and replace them with GetTickCount .
5. In each of the forms used in the Real Speed tests, change the string assigned to the Label controls used to display elapsed time so that they receive their values from the new ElapsedTime() function.
6. Conduct your own tests of EXE files built to P-Code, fast native code, small native code, and unoptimized native code to see where the benefits are most apparent.

20.2 Measuring the Benefits of Advanced Optimizations

The tests discussed so far apply only to the basic optimizations. Next you will create some tests to measure the benefits of the advanced optimization features. The instructions assume that you have already completed the first exercise.

Estimated Time: 25 minutes

To create this exercise, follow these steps:

1. Again, you use the Optimize project. Its default location is in the \VB\Samples\Pguide\Optimize directory.

APPLY YOUR KNOWLEDGE

2. Create five native-code EXEs optimized for speed. Use the Advanced Optimizations features to create one executable for each of the five optimization features. Each EXE should have just one advanced feature activated. That is, one option will use Assume No Aliasing, but no other switches; one will use Remove Array Bounds Checks, but no others; and so on. Choose a meaningful name for each EXE to make it easier to remember which optimization is used in each of them.
3. Run the tests again. Compare your results to the previous tests conducted with the fast native-code EXE to see how much difference the advanced optimizations make.

20.3 More Accurate Measurements

Although it is more accurate than `Timer()`, the `GetTickCount()` function still isn't a perfect measure of optimizations. Like `Timer()`, it measures the total time elapsed between a starting and ending time. Because other processes run concurrently on a Windows computer besides the one that you want to time, the time between two measures of `GetTickCount()` includes time used by processes irrelevant to your measurement.

The Win32 API also has a `GetProcessTimes()` function, however, which measures the elapsed time within a single process. Its declaration, and that of a user-defined type that it requires, is as follows:

```
Type FILETIME
    dwLowDateTime As Long
    dwHighDateTime As Long
End Type

Declare Function GetProcessTimes Lib
➥Òkernel32Ó ( _
    ByVal hProcess As Long, _
    lpCreationTime As FILETIME, _
    lpExitTime As FILETIME, _
    lpKernelTime As FILETIME, _
    lpUserTime As FILETIME) _
    As Long
```

If you want to get an even more accurate view of the relative merits of VB6's compiler optimizations, replace the measurements of elapsed time derived from `GetTickCount()` with `GetProcessTimes()`.

20.4 Compiler Constants

In this exercise, you create a program that uses a compiler constant to conditionally compile code to generate different messages.

Estimated Time: 15 minutes

To create this exercise, follow these steps:

1. Create a new standard EXE project.
2. In the General Declarations section of a form, write code to create a compiler constant. Call the constant `ZIPPY` and assign it a value of `1`.
3. Create a command button on the same form. In the button's `Click` event code, enter the following:

```
#If ZIPPY = 1 Then
    MsgBox ÒZippy = 1.  What a surprise.Ó
#ElseIf ZIPPY = 2 Then
    MsgBox ÒZippy = 2.  Alert the media.Ó
#Else
    MsgBox ÒZippy = a value wholly unlike 1 or
➥2.  Yow!Ó
#End If
```

4. Compile and run the project.
5. Now open the Project Properties dialog box to give `ZIPPY` a value of `2`. Compile and run the project again. Notice there's no effect, because the coded assignment of step 2 overrides anything you type in the Project Properties window. You must first disable the line you created in step 2 to be able to control the value of `ZIPPY` from the Project Properties dialog box.

APPLY YOUR KNOWLEDGE

6. Finally, use the command line to set the value of ZIPPY to 3. Compile and run the project again. If you wish to attain a truly Zippy-like state of Zen (and honorary lifetime status as a true Pinhead), continue the tests, incrementing the value of ZIPPY by 1 each time, to see whether you ever produce a different result.

Review Questions

1. Which optimization options are available when compiling a VB application to native code?
2. What are the differences among interpreted code, machine code, and pseudocode?
3. The Compile On Demand feature performs what function?
4. A compiler constant verDEBUG is defined in code module MOD1.BAS and assigned a value of 1. On the command line, verDEBUG is assigned a value of 2. What value will be used when the constant is encountered in MOD1.BAS?
5. Code that was created in version 4.0 of VB could provide a single code base for two versions (16-bit and 32-bit) of an application. What conditional compilation directives were used to keep the version-specific elements of the code separated?

Exam Questions

1. Pseudocode is
   - A. A natural language description of the operations of a computer program.
   - B. A partial compilation of source code into machine code.
   - C. A portable code format capable of running on multiple platforms without modification.
   - D. A formal means of expressing an algorithm.
2. The total size of the files necessary to distribute an application will be _____ when the program is compiled to optimize for size than when compiled to optimize for speed.
   - A. Much smaller
   - B. Slightly smaller
   - C. Slightly larger
   - D. Much larger
3. Native code executes _____ than P-Code.
   - A. Much faster
   - B. A little faster
   - C. No more quickly
   - D. A little slower
4. A program that has been optimized to run on a Pentium Pro rather than on a 386 or 486:
   - A. Will not load into memory.
   - B. Loads into memory but produces a GPF on execution.
   - C. Runs more slowly.
   - D. Runs just as well.
5. The optional symbolic debug information that can be created when compiling to native code
   - A. Significantly increases the size of the executable file.
   - B. Slows down the execution of the executable file.

APPLY YOUR KNOWLEDGE

C. Makes it possible to reverse a program's execution during a trace.

D. Is useful only with an external debugger.

6. Compile On Demand makes it possible to

A. Distribute applications that build their EXE on-the-fly when the user is ready to install a program.

B. Create application components that are responsible for compiling themselves.

C. Test portions of an application without fully compiling the entire program first.

D. View the values of variables without explicitly setting a Watch window.

7. A compiler constant gDEBUGis defined and assigned a value of 1 in a global module. The following code is placed in the Load event of form frmStartUp :

```
#If gDEBUG Then
   MsgBox ÒDebuggingÓ
#End If
```

If Compile On Demand is active, what happens when frmStartUp loads?

A. The form's Load event code generates a compiler error.

B. A message box displays with the message ÒDebuggingÓ .

C. The program enters Break mode.

D. The form loads normally, but no message box displays.

8. A compiler constant verDEBUG has been defined and assigned a value of 1. Several code blocks are enclosed by #If verDEBUG / #End If pairs. The program is compiled by typing:

```
vb.exe /make ProjectName.vbp /d verDEBUG=0
```

Which of the following is true:

A. The value of verDEBUG set in code takes precedence.

B. The value of verDEBUG set on the command line takes precedence.

C. A VB runtime error occurs due to an invalid command line parameter.

D. The compiled program permits a user to specify the value of verDEBUG on the command line.

9. A compiler constant is defined #Const Bumble = 1 in the General Declarations section of a form. The same form has a command button with a compiler constant defined in its Click event code as #Const Bumble = 0 . The rest of the Click event code looks like this:

```
#If Bumble Then
    MsgBox ÒLookie what Bumble can do!Ó
#End If
```

What happens when the project is run and the command button is clicked?

A. The project will not run due to a compiler error.

B. The form-level constant takes precedence over the local constant, so the message box displays.

C. The form-level constant takes precedence over the local constant, so nothing happens.

D. The local constant takes precedence over the form-level constant, so nothing happens.

10. A compiler constant verDEBUG has been defined and assigned a value of 1. Select the combination of commands to begin and end each code block that will cause the compiler to ignore a code block when building a verDEBUG version of a program:

A.
```
#Ifdef verDEBUG
#Endif
```

APPLY YOUR KNOWLEDGE

B. #If verDEBUG = TRUE Then
#Endif

C. #If verDEBUG = TRUE
#End If

D. #If verDEBUG Then
#End If

11. Projects using conditional compilation require extra lines of code for the preprocessor directives and for the alternative versions of the project. Consequently, the size of the executable program is _____ than if separate versions of the project were maintained with no use of the preprocessor.

    A. Larger (by the size required by the extra lines of code only)

    B. Larger (by the size required by the preprocessor code only)

    C. Larger (by the size required by both the extra lines of code and the preprocessor code)

    D. No larger

Answers to Review Questions

1. Basic optimization choices include optimizing for fast code, optimizing for small code, and no optimization. You also may choose to favor the Pentium Pro. Advanced optimization choices are: assume no aliasing, remove array bounds checks, remove integer overflow checks, remove floating-point error checks, allow unrounded floating-point operations, and removing Pentium FDIV checks. See the individual sections for each of these under "Basic Optimizations" and "Advanced Optimizations."

2. When a program is interpreted, each line of source code is converted into machine instructions whenever it is encountered at runtime. When a program has been compiled to machine code, this conversion has already been done, so the program executes more quickly. Pseudocode stands midway between interpreted code and machine code. Your source code isn't compiled directly to machine code, but instead is turned into a series of tokens that represent particular operations. These tokens are passed to the runtime files, which contain the actual executable code. See "P-Code Versus Native Code."

3. Instead of fully compiling your application prior to testing it, Compile On Demand compiles code on an as-needed basis during testing. See "Using Compile On Demand."

4. The value of verDEBUG will be 1. The value assigned in the module overrides the assignment on the command line. See "Scope and Persistence."

5. The 32-bit-specific code could begin with #If WIN32 followed by an #Else , and then the 16-bit code. The block ends with #End If . See "Predefined Compiler Constants."

Answers to Exam Questions

1. **B**. Microsoft's implementation of pseudocode in VB partially compiles your program code, which still must be interpreted by a runtime DLL as it executes. For more information, see the section titled "P-Code."

2. **B**. The total size is only slightly smaller because the executable file is a relatively small portion of the files necessary to distribute an application.

APPLY YOUR KNOWLEDGE

The runtime files and custom control files consume a proportionately greater amount of space. For more information, see the section titled "Compiling to P-Code."

3. **A**. Native code is generally much faster than P-Code. For more information, see the section titled "P-Code."

4. **C**. The program will run, but not as quickly. For more information, see the section titled "Favoring Pentium Pro."

5. **D**. You need an external debugger to use the symbolic debug information. For more information, see the section titled "Creating Symbolic Debug Info."

6. **C**. Using Compile On Demand, an application need not be fully compiled to test it. For more information, see the section titled "Using Compile On Demand."

7. **D**. With Compile On Demand activated, the global module in which gDEBUG is defined will not have been compiled. For more information, see the sections titled "Using Debug Mode" and "Using Compile On Demand."

8. **A**. The value set in code overrides the value on the command line. For more information, see the sections titled "Declaring in the Command Line" and "Scope and Persistence."

9. **A**. A constant can only be defined to have a single value, so the second attempt to define the constant with a new value generates a duplicate definition error. For more information, see the section titled "Declaring in Code."

10. **B**. Somewhat of a trick question (because we want code that will compile and *ignore* a code block). The first choice uses the wrong syntax. (Ò#IfdefÓ isn't part of VB.) The second choice fails because TRUE is defined as Ð1. It is therefore the correct answer, because we want the comparison to fail. The third choice omits ÒThenÓ. Choice D is syntactically correct and will run the code block—which we don't want to happen. For more information, see the section titled "Using Debug Mode."

11. **D**. The extra lines aren't compiled, so they make no difference to the size of an EXE file. For more information, see the section titled "Preprocessor Directives."

OBJECTIVES

This chapter helps you prepare for the exam by covering the following objective and its subobjectives:

Use the Package and Deployment Wizard to create a setup program that installs a distributed/desktop application, registers the COM components, and allows for uninstall (70-175 and 70-176).

- Package and Deployment Wizard is a utility that comes with VB6 and that enables you to create Windows-standard install and uninstall routines for distribution to end users.

Register a component that implements DCOM (70-175).

- You can take just a couple of extra steps to make sure that your application's setup package implements DCOM (Distributed COM) on both the client and server side.

Configure DCOM on a client computer and on a server computer (70-175).

- Package and Deployment Wizard takes care of distributing the necessary files for DCOM on both client and server machines.

Plan and implement floppy disk–based deployment or compact disc–based deployment for a distributed/desktop application (70-175 and 70-176).

- After you have created a setup package, you can then *deploy* your application by moving its setup package to the site from where end users will actually install it. This deployment site might be floppy disks. You can use Package and Deployment Wizard to automate the deployment of your application.

CHAPTER 21

Using the Package and Deployment Wizard to Create a Setup Program

OBJECTIVES

Plan and implement Web-based deployment for a distributed/desktop application (70-175 and 70-176).

- You can use Package and Deployment Wizard to deploy your installation package via an Internet- or intranet-based download.

Plan and implement network-based deployment for a distributed/desktop application (70-175 and 70-176).

- You can use Package and Deployment Wizard to deploy your installation package via a network directory available to all potential users.

Deploy application updates for distributed applications (70-175 and 70-176).

- After you have deployed an application, you may not want to re-create the entire installation and deployment every time you have a minor change. By knowing some facts about the end products of Package and Deployment Wizard's activities, you can manually re-create the necessary components for minor changes in the deployment of an application.

OUTLINE

Using Package and Deployment Wizard to Create a Setup Program 966

Preparing to Run Package and Deployment Wizard 967
Starting Package and Deployment Wizard and Choosing the Type of Package 968
Choosing the Type of Setup Package 969
Creating a Standard Setup Package 970
Creating an Internet Setup Package 975
Creating a Dependency File 977

Standard Files Used in a Microsoft Setup 978

Setup File Information in SETUP.LST 979
Dependency Information in DEP Files 981
SETUP.EXE and Package and Deployment Wizard's Custom Setup 983

Customizing a Standard Setup 984

Customizing SETUP.LST and Your Application's DEP File 984
Customizing the Standard VB Setup Project 985
Implementing Application Removal 985

Registering a Component That Implements DCOM and Configuring DCOM 986

Deploying Your Application 987

Deploying to Floppy Disks 988
Deploying to a Network Directory or to CDs 990
Deploying to the Web 992
Deploying Updates to Your Application 994

Chapter Summary 995

STUDY STRATEGIES

- Create test projects for both standard EXEs and COM components. They need not have code for purposes of experimenting with Package and Deployment Wizard. Run Package and Deployment Wizard against these projects to create standard and Internet setup packages as well as a dependency file. See the sections in this chapter about their creation as well as Exercises 21.1 and 21.2.
- Understand the relationship between the various files and file types included in standard and Internet setup packages. This would include the SETUP.EXE file, the SETUP.LST file, and SETUP1.EXE (see "Standard Files Used in a Microsoft Setup"), as well as the file types with the following extensions: .CAB and .DDF (see "Creating a Standard Setup Package"), .INF (see "Creating an Internet Package Setup"), .DEP (see "Creating a Dependency File" and "Dependency Information in .DEP Files"), and .VBR (see "Registering a Component That Implements DCOM and Configuring DCOM"). See discussions of these file types throughout this chapter.
- Briefly view the source code for VB's customizable Setup project. See the section "Customizing the Standard VB Setup Project."
- Briefly view the contents of a SETUP1.LST file as created by Package and Deployment Wizard. See the discussion in "Setup File Information in SETUP.LST" and "Customizing SETUP.LST and your Application's DEP File."
- Briefly view the contents of an HTML file that Package and Deployment Wizard creates for an Internet package.
- Practice compiling applications and components that will implement DCOM, as discussed in "Registering a Component That Implements DCOM and Configuring DCOM." See also Exercise 21.3.
- Experiment with Package and Deployment Wizard to deploy a setup package that you have created. Try out Internet, floppy disk, and directory deployment, as discussed in the sections under "Deploying your Application." See also Exercises 21.4, 21.5, and 21.6.

INTRODUCTION

A *setup package* is a group of files that can be used to distribute an application to users or to install component applications on servers. Package and Deployment Wizard is a VB add-in utility that enables you to prepare such packages for your VB applications. Package and Deployment wizard enables you to create either a traditional Windows setup package (where the user runs SETUP.EXE) or an Internet setup package (your component application is installed from a Web page).

The packages created by Package and Deployment Wizard also typically take care of installing and registering any support files needed by your application, such as COM components and other elements.

These packages also typically install the necessary utilities and information on target computers so that users can use the Windows Add/Remove utility to cleanly and securely remove your application.

This chapter discusses how you can use Package and Deployment Wizard to create standard setup packages for your applications and then deploy those setup packages.

USING PACKAGE AND DEPLOYMENT WIZARD TO CREATE A SETUP PROGRAM

. Use the Package and Deployment Wizard to create a setup program that installs a distributed/desktop application, registers the COM components, and allows for uninstall.

The Package and Deployment Wizard in Visual Basic 6 is a complete remake of the Setup wizard of previous versions. As its name implies, the wizard has two functions:

‡ Packaging applications into—well, what else—packages. A package is a set of files necessary to implement an application, including supporting DLLs, VB runtime libraries, data-access drivers, and other files.

‡ Deploying packaged applications—that is, making the packages available for use from a given environment, such as a network or the Internet.

Package and Deployment Wizard runs as an add-in to VB6. When you install VB6, Package and Deployment Wizard should install along with it. You can verify that Package and Deployment Wizard is available in your environment by checking for it in the Add-Ins menu, as shown in Figure 21.1.

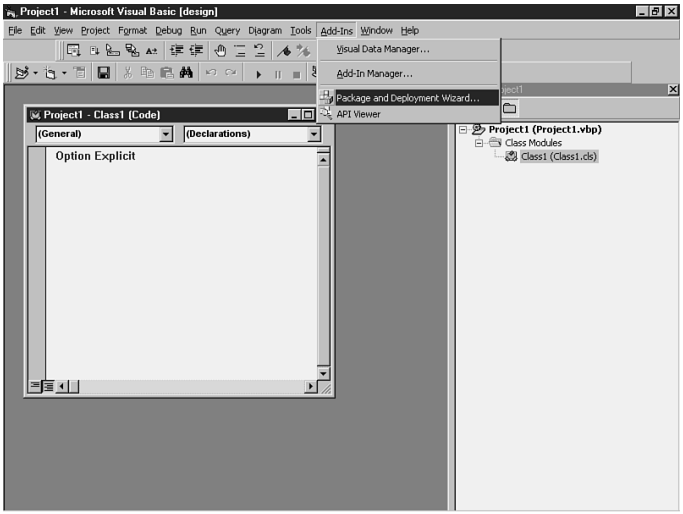


FIGURE 21.1
Package and Deployment Wizard as an option in the Add-Ins menu.

You run Package and Deployment Wizard after you have finished your application and are ready to create a distribution vehicle for the application. Such a distribution vehicle might be either distribution disks or a special directory on a network. The distribution vehicle would contain compressed versions of all files necessary to install and run the application.

If your distribution is for a COM component that you have created, you can also choose to create an Internet download setup for your component.

Preparing to Run Package and Deployment Wizard

Before you run Package and Deployment Wizard, you should perform the following steps:

‡ Use the Project, References option on the VB menu to remove any unneeded references to data-access libraries (such as ADO or DAO) or class libraries.

‡ Remove any unneeded references to ActiveX controls either by:

- Removing the individual references to unused controls with the Project, Components option on the VB menu.

or

- Making sure that the option Remove information about unused ActiveX controls is checked on the Make tab of the Project, Properties dialog box. If your application adds controls with the `Controls.Add` method, you must make sure that this option does not remove references to these controls.

‡ Save your application's project and compile it. (This is not strictly necessary because Package and Deployment Wizard will perform this step for you if you forget to do it.)

The following sections give you a tour of the screens that appear during a typical session with Package and Deployment Wizard.

Starting Package and Deployment Wizard and Choosing the Type of Package

You start Package and Deployment Wizard by choosing it from the Add-Ins menu, as shown earlier in Figure 21.1.

Package and Deployment Wizard's first screen (shown in Figure 21.2) enables you to choose what you wish to do.

‡ **Package**. This option is the main focus of this chapter. It enables you to create a setup package—that is, a compressed set of files and other necessary information for your application to run in a desktop or network environment. A setup package also includes all the auxiliary executable and information files needed to "unpack" the package and install it correctly on another system.

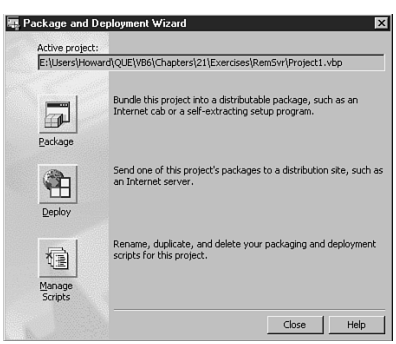


FIGURE 21.2
The first screen of the Package and Deployment Wizard.

‡ **Deploy**. This option helps you ready an already created package for distribution. See the discussion of deployment later in this chapter under the section titled "Deploying Your Application."

‡ **Manage Scripts**. This option enables you to save, rename, or delete Package and Deployment scripts. These scripts are just records of the options that you choose when using the Package and Deployment Wizard to create a package. Package and Deployment Wizard will manage a file with extension .PDM containing various scripts for your project in the folder where the project's VBP file resides. By saving those options in a Package and Deployment script, you can use them over again in the future to set up the same project or others like it.

NOTE

Prompts to Save and Compile At this point, Package and Deployment Wizard will sense whether your project has been saved and compiled and will prompt you if you have not saved or compiled. If you receive prompts asking you to save and compile, you may just answer "yes" to the prompts, wait a few instants, and then proceed.

To continue with packaging, you would choose the first option: Package.

Choosing the Type of Setup Package

After you have chosen Package on Package and Deployment Wizard's opening screen, you will see the Package Type screen, as shown in Figure 21.3.

The Package Type screen offers you two or three choices, depending on the type of project that you are packaging:

‡ **Standard Setup Package**. This option creates CAB files (cabinet files, the standard archive file type that Microsoft uses for distribution) and all the other files necessary to install your application with a standard Windows setup (running SETUP.EXE).

‡ **Internet Package**. This option creates CAB files and other files necessary to install your package as a download over the Internet or intranet (not available for standard EXE projects).

‡ **Dependency File**. This option creates a single DEP file that can be used to show your project's dependencies, or the other files that your project needs to run correctly on a computer. For a fuller discussion of the contents and purpose of dependency files, see the section titled "Dependency Information in DEP Files."

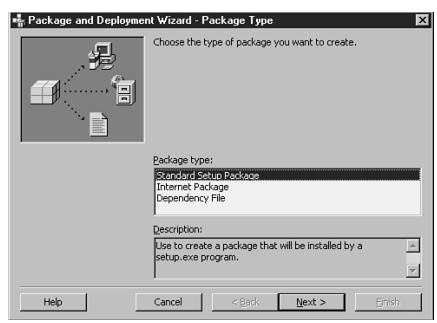


FIGURE 21.3
The Package Type screen of the Package and Deployment Wizard.

The following sections discuss how to use Package and Deployment Wizard to create these three types of setup package.

Creating a Standard Setup Package

As explained in "Choosing the Type of Setup Package", a standard setup package provides a setup to the user that enables the user to run a standard Windows SETUP.EXE to install your application.

You begin to create a standard setup package by running the Package and Deployment Wizard. You must choose the Package option on the first screen, and then choose Standard Setup from the second screen (as mentioned in the sections titled "Starting Package and Deployment Wizard and Choosing the Type of Package" and "Choosing the Type of Setup Package").

The following pages describe the additional steps that you must take to create your Standard setup package.

After you have selected Standard Setup as the type of package, you will see the Package Folder screen, as shown in Figure 21.4. You can use this screen to choose or create a folder where Package and Deployment Wizard will create your package. The default folder is the folder where the Project's VBP file resides. It is usually best to create a separate folder so as not to mingle development file with distribution package files.

After you have indicated a folder, you can click the Next button to proceed to the Included Files dialog box, shown in Figure 21.5. Package and Deployment Wizard displays all the files that it could determine were needed in the distribution package. You can use this screen to exclude files from the distribution package: Just uncheck the box next to the filename.

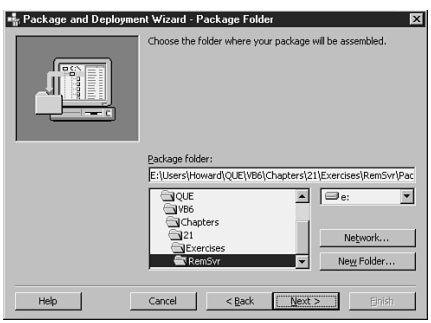


FIGURE 21.4▲
The Package and Deployment Wizard's Package Folder dialog box.

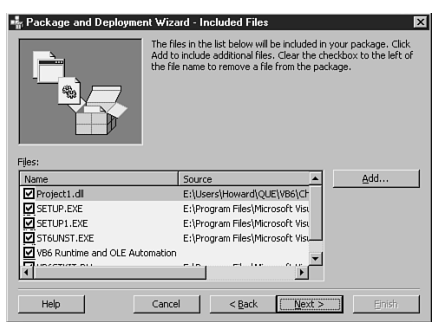


FIGURE 21.5.
The Package and Deployment Wizard's Included Files dialog box.

If you need to include a file that Package and Deployment Wizard didn't automatically detect, you can click the Add button on the Included Files screen and select the necessary file with the resulting standard File Open dialog box.

After you have selected a file to include in the package, you may see the Missing Dependency Information dialog box shown in Figure 21.6. This screen warns you that Package and Deployment Wizard could not find a dependency (DEP) file for the included file. (See the section titled "Dependency Information in DEP Files" for a further explanation of dependency files.)

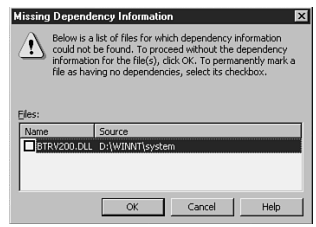


FIGURE 21.6▲
Package and Deployment Wizard's Missing Dependency Information dialog box.

If you check the filename on this screen, Package and Deployment Wizard will not bother you again for this file's dependencies. If you leave the filename unchecked, you should create a dependency file for this file and include the dependency file in the distribution package for this project. (See the section titled "Creating a Dependency File" for instructions on dependency file creation.)

After you have finished including file information, you can click the Next button to proceed to the Cab Options dialog box, shown in Figure 21.7.

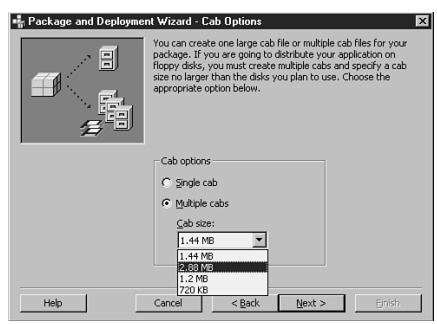


FIGURE 21.7▲
Package and Deployment Wizard's Cab Options dialog box.

The Cab Options screen enables you to control the number and size of CAB files that Package and Deployment Wizard will make for your application's distribution package.

Cabinet (CAB) files are now the standard format that Microsoft distribution packages use to compress information.

You can use the dialog box to decide whether to include all the package's files in a single CAB file or whether to break the information into multiple CAB files. If you decide to use multiple CAB files, you can determine the size of the individual files with the Cab Size drop-down list, as shown in Figure 21.7.

This screen is significant because it enables you to adjust the maximum size of your package's distribution files in accordance with the type of media that you will use to distribute the application.

Typically, you will choose the Single Cab option whenever you can make the package available to users from a medium that supports very large amounts of storage, such as a network installation or a CD.

NOTE

Cab Files Replace Compressed Individual Files In previous versions of VB and Visual Studio, standard distribution packages included a separate compressed copy of each file to be distributed (the last letter of the file's extension was an underscore—for example, EX_, DL_). This format has now been replaced with the CAB file format, which compresses multiple files into a single CAB file.

If you need to distribute your application to users on media with size restrictions, such as disks, you will choose the Multiple Cabs option and adjust the CAB size appropriately.

After Package and Deployment Wizard has finished and has created your CAB files, you can then deploy them (along with the files SETUP.EXE and SETUP.LST) to the distribution site. If you chose to create multiple CAB files, you can deploy them on disks of the appropriate size. See the following chapter for more information on application deployment.

After you click the Next button to move beyond the Cab Options dialog box, you will see the Installation Title dialog box, shown in Figure 21.8. You can modify the single field on this screen to adjust the title that users will see on the setup screen when they run the setup for your application.

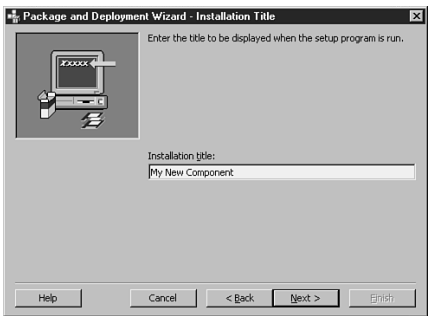


FIGURE 21.8▲
Package and Deployment Wizard's Installation Title dialog box.

The next screen after the Installation Title dialog box is the Start Menu Items dialog box, shown in Figure 21.9. This screen enables you to specify whether your application will appear under the Programs section of the Windows Start menu. You can also, of course, determine the exact wording that will appear on the menu item.

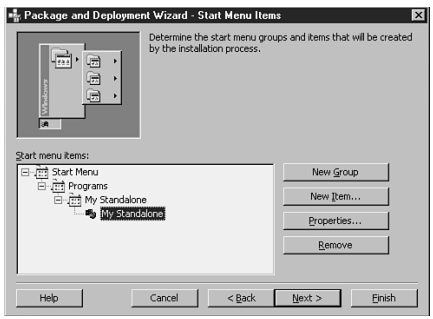


FIGURE 21.9.
Package and Deployment Wizard's Start Menu Items dialog box.

If your application is a standalone executable, Package and Deployment Wizard will automatically supply an entry on the Start menu items tree. You can remove the item altogether with the Remove button, add a new item or program group with the New Group or New Item buttons, or modify the properties of the current item with the Properties button, as shown in Figure 21.10.

Note that Start's drop-down list contains *macros* for typical paths on a machine with a Windows install. Because the exact paths and drive letters for these paths can vary from one machine to the next, these macros provide generic tokens that the setup routine will resolve into the correct paths on each user's system.

The next screen after the Start Menu Items dialog box asks you for the Install Locations of the compiled components of your application, typically just one file (see Figure 21.11).

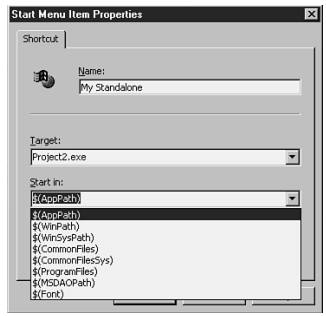
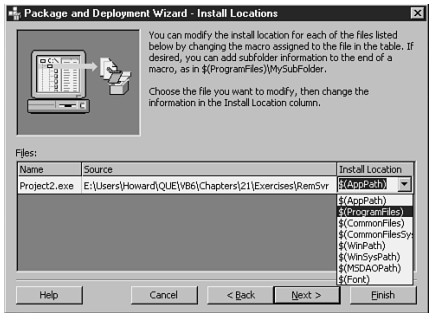


FIGURE 21.10▲
Specifying the properties for your application's Start menu icon.



, FIGURE 21.11
The Package and Deployment Wizard's Install Locations dialog box.

Once again, the Install Location drop-down list for the file offers a choice of path macros as in the Start menu Properties dialog box just mentioned.

The next screen after the Install Locations screen is the Shared Files dialog box, shown in Figure 21.12. As does the previous screen, this screen also lists the compiled components of your application.

The purpose of the Install Locations screen is to specify that your component can be shared with other applications on the system. If you mark the application as a Shared file, the system will maintain a reference count on the file. This means that when other applications that use the file are added to the system, the reference count will increase; it also means that when applications that use the file are removed, the reference count will decrease. The file will only be removed from the system if the reference count decreases to zero.

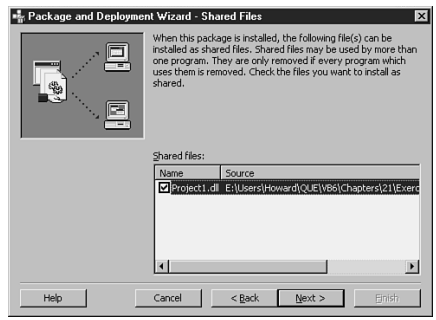


FIGURE 21.12▲
Package and Deployment Wizard's Shared Files dialog box.

The next screen is the Finished screen, shown in Figure 21.13. When you click the Finish button, the Package and Deployment Wizard builds your package and its support files.

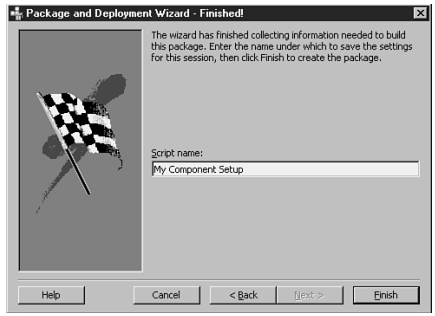


FIGURE 21.13
Package and Deployment Wizard's Finished screen.

After you exit the Finished screen, you will receive a notification screen, the Packaging Report screen (not shown here). This screen informs you of the location of the newly created package. It also informs you of the existence of a batch file (of the form *PROJECTNAME*.BAT) that you can use to re-create the project's CAB file or files if you need to change files that make up the package and redeploy it.

If you examine the directory where you had Package and Deployment Wizard create the package, you will note that there are one or more CAB files as well as SETUP.EXE and SETUP.LST. These are the files that must be distributed to users who need to install the application.

Underneath the directory where your package resides, you will find another directory, named Support. The Support folder contains the files necessary to rebuild the package, including the following:

- The files that are distributed in the CAB file (including SETUP1.EXE, SETUP.EXE, SETUP.LST, and the VB Application Removal utility, ST6UNST.EXE).
- A batch (BAT) file with the same name as the application. You can use this batch file to run the MakeCab utility to re-create the project's CAB files. This file might come in handy if you update one of the distribution files (say, your project's EXE file) and need to re-create the deployment package quickly. You could just copy the new version of the updated file into the Support directory, and then run the batch file and thus update the CAB files.

- A DDF file with the same name as the application. The DDF file is a text file read by the MakeCab utility to determine how to build the CAB files and also to determine which source files to use.

Creating an Internet Setup Package

As explained in "Choosing the Type of Setup Package," an Internet setup package provides a setup to the user that installs your package as a download over the Internet or intranet.

Internet setup packages are not available for standard EXE applications.

You create an Internet setup package by running the Package and Deployment Wizard, choosing the Package option on the first screen, and choosing Internet Setup from the second screen, as mentioned in "Starting Package and Deployment Wizard and Choosing the type of Package" and "Choosing the Type of Setup Package."

The following pages describe the additional steps that you must take to create your Internet setup package.

The Package and Deployment Wizard screens for an Internet setup package are comparable to the Package and Deployment Wizard screens for a standard setup package. The following is a brief description of the steps for creating an Internet setup package:

- The Package Folder dialog box is the same as the Package Folder dialog box for a standard setup (see preceding section).
- The Included Files dialog box is the same as the Included Files dialog box for a Standard setup (see preceding section), although typically you will see fewer files listed.
- The File Source dialog box, shown in Figure 21.14, is unique to the Internet setup. This dialog box enables you to specify where users will get the files for this setup when their browsers attempt to install the application. For the component files that you create, you may specify the current CAB file being created as the source, or you may specify an alternative URL as the source.

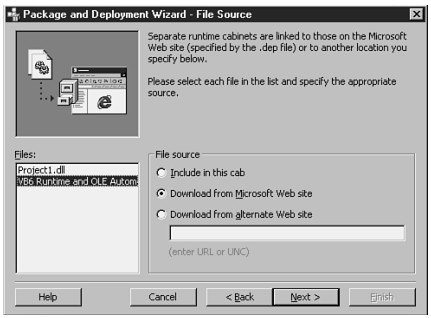


FIGURE 21.14▲
The Package and Deployment Wizard's File Source dialog box (Internet package setup only).

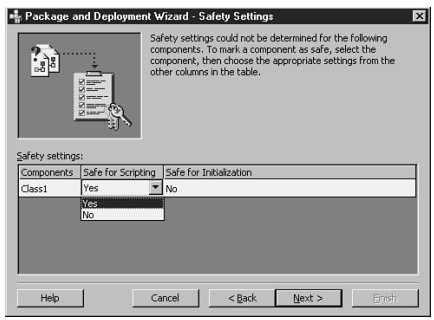
For the VB6 runtime files, you may also specify the Microsoft Web site as the source for the files (as illustrated in the figure). Specifying the Microsoft Web site option guarantees that users installing your application will always get the latest version of the VB6 runtime.

‡ The Safety Settings dialog box, shown in Figure 21.15, is also unique to an Internet Package setup. The meaning of the two options, Safe for Scripting and Safe for Initialization, is as follows:

- *Safe For Scripting* means that the component can't be used to corrupt the user's computer or get unauthorized information from the user's computer.

- *Safe for Initialization* means that the component cannot be used to do harm on a user's computer when it is initialized.

FIGURE 21.15.
The Package and Deployment Wizard's Safety Settings dialog box (Internet package setup only).



These screens provide a Package and Deployment Wizard (Internet package setup only).

When the Internet package setup completes, you will find a CAB file (containing the files to be distributed) and an HTML file in the designated package directory. This HTML file contains the information that you would need to embed in a Web page so that browsers will download your application. Listing 21.1 shows an example of the contents of such an HTML file.

LISTING 21.1

HTML CODE NECESSARY TO IMPLEMENT AN INTERNET DOWNLOAD FOR YOUR APPLICATION'S PACKAGE

```
<HTML>
<HEAD>
<TITLE>Project1.CAB</TITLE>
</HEAD>
<BODY>
<OBJECT ID=ÓClass1Ó
CLASSID=ÓCLSID:F4B1B409-7C2F-11D2-9C45-00A024C3B222Ó
CODEBASE=ÓProject1.CAB#version=1,0,0,0Ó>
</OBJECT>
</BODY>
</HTML>
```

When users navigate to a Web site containing this code, their browsers will begin a background download of your application.

If you look in the Support folder created under the package directory, you will see the following:

- The application's file or files (EXEs, DLLs, or OCX files, for example).
- An INF file. This file is only created for Internet download packages and becomes part of the CAB file. It is a text file that contains dependency information needed by the end user's browser. Based on the information in the INF file, the browser will download and install other files that your application needs to run.
- The batch (BAT) file needed to run the MakeCab utility and re-create the CAB file.
- The DDF file that the MakeCab utility needs to read to find instructions on how to create the CAB file.

Creating a Dependency File

As explained earlier, a dependency file shows your project's dependencies or the other files that your project needs to run correctly on a computer. For a fuller discussion of the contents and purpose of dependency files, see the section in this chapter titled "Dependency Information in DEP Files."

You create a dependency file by running the Package and Deployment Wizard, choosing the Package option on the first screen, and choosing Dependency File from the second screen.

The following pages describe the additional steps that you must take to create your dependency file.

The Package and Deployment Wizard screens for creating a dependency file are comparable to the Package and Deployment Wizard screens for a standard setup package. Following is a brief description of the steps for creating a dependency file:

- The Package Folder screen is the same as the standard setup package's Package Folder screen.
- The Included Files screen is the same as the standard setup package's Included Files screen.
- The Cab Information Screen dialog box is unique to a Package and Deployment Wizard dependency file setup (see Figure 21.16).
- The Install Locations screen is the same as the standard setup package's Install Locations screen.

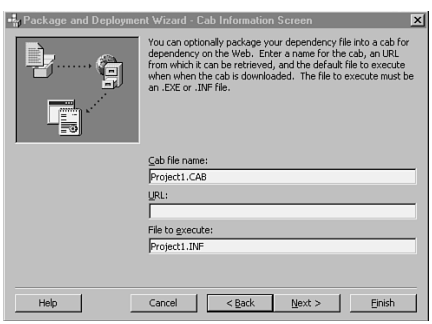


FIGURE 21.16
The Package and Deployment Wizard's Cab Information Screen dialog box (dependency file setup only).

After Package and Deployment Wizard has finished creating the dependency file, the DEP file will appear in the directory you specified (preferably, the same directory where you store the application's executable).

Standard Files Used in a Microsoft Setup

A standard Microsoft setup routine requires the user to invoke SETUP.EXE.

SETUP.EXE is *not* the customized setup routine that Package and Deployment Wizard created, however. The customized setup executable is normally named SETUP1.EXE, although this is not a required name. SETUP.EXE runs this customized setup routine.

SETUP.EXE uses a special text file named SETUP.LST to determine the name of the customized setup routine and the names of files needed by the setup routine to run on the user's system.

SETUP.LST also contains the names of other files that SETUP1.EXE needs to install on the user's system. Package and Deployment Wizard also writes to SETUP.LST.

The following sections explain the relationship between SETUP.EXE—the custom setup routine (usually named SETUP1.EXE), SETUP.LST, and other files.

Setup File Information in SETUP.LST

As just mentioned, Package and Deployment Wizard creates a special text file named SETUP.LST for each application that you set up. SETUP.LST looks a lot like an INI file (see the example in Listing 21.2). It has named sections with headers surrounded by brackets. Each section, in turn, contains entries of the form:

```
ItemName=value
```

The entries under the [Bootstrap Files] and [Setup1] sections are the names of files to be copied to the user's system. Each file entry contains a long comma-delimited list detailing the file's compressed and uncompressed names as well as installation instructions and version information.

SETUP.LST's [BootStrap] and [Bootstrap Files] sections contain information about actions that SETUP.EXE must perform on the user's system before the main setup routine can run.

If the main setup routine is a VB program (as it always is if you have created it with Package and Deployment Wizard), for instance, users may not have the necessary files installed on their workstations to run a VB application of the appropriate version. SETUP.EXE must copy and register the VB runtime libraries on the user's system before the main setup routine (written in VB) can run.

The Spawn entry under the [Bootstrap] section gives the name of the main Setup routine. By default, it is SETUP1.EXE. The [Bootstrap] section also gives other information about the setup environment in general, including the initial dialog box title and prompt as well as CAB file and uninstall information.

The [Bootstrap Files] section lists the files that SETUP.EXE must copy to and register on the end user's system before the main setup routine can run.

A couple of other groups may exist ([Icon Groups] and a section named with your application's title) if you chose to create an entry or a group for your application on the Windows Start Menu/Programs menu.

The [Setup] section contains general information needed during the setup process, such as screen captions and install directories.

The [Setup1] section lists other files that the main setup routine must copy to and register on the user's system.

LISTING 21.2

CONTENTS OF A TYPICAL SETUP.LST FILE

```
[Bootstrap]
SetupTitle=Install
SetupText=Copying Files, please stand by.
CabFile=Project2.CAB
Spawn=Setup1.exe
Uninstal=st6unst.exe
TmpDir=msftqws.Package and Deployment Wizard
Cabs=1

[Bootstrap Files]
File1=@VB6STKIT.DLL,$(WinSysPathSysFile),,,5/20/98 11:00:00
➥PM,102400,6.0.81.41
File2=@COMCAT.DLL,$(WinSysPathSysFile),$(DLLSelfRegister),,
➥11/18/97 12:00:00 AM,22288,4.71.1441.1
File3=@STDOLE2.TLB,$(WinSysPathSysFile),,,5/21/98 12:00:00
➥AM,17920,2.30.4260.1
File4=@ASYCFILT.DLL,$(WinSysPathSysFile),,,5/21/98 12:00:00
➥AM,147728,2.30.4260.1
File5=@OLEPRO32.DLL,$(WinSysPathSysFile),$(DLLSelfRegister),,
➥5/21/98 12:00:00 AM,164112,5.0.4260.1
File6=@OLEAUT32.DLL,$(WinSysPathSysFile),$(DLLSelfRegister),,
➥5/21/98 12:00:00 AM,598288,2.30.4260.1
File7=@MSVBVM60.DLL,$(WinSysPathSysFile),$(DLLSelfRegister),,
➥5/21/98 12:00:00 AM,1417216,6.0.81.41

[IconGroups]
Group0=My Standalone
PrivateGroup0=True
Parent0=$(Programs)

[My Standalone]
Icon1=Project2.exe
Title1=My Standalone
StartIn1=$(AppPath)

[Setup]
Title=My Standalone
```

```
DefaultDir=$(ProgramFiles)\Project1
AppExe=Project2.exe
AppToUninstall=Project2.exe

[Setup1 Files]
File1=@Project2.exe,$(AppPath),,,11/14/98 10:42:55
➥PM,16384,1.0.0.0

; The following lines may be deleted in order to obtain extra
; space for customizing this Þle on a full installation
➥diskette.
;
; XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
; XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
; XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
; XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
; XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

The developer may customize the setup process by changing the information in this file with a text editor.

Dependency Information in DEP Files

When Package and Deployment Wizard includes a file in your project's setup, that file might also require the presence of other, supporting files on the user's system to run properly.

Certain Windows common controls (listed in the Project, Components dialog box as Microsoft Windows Common Controls-3) that are implemented in the file COMCT332.OCX, also require other files to run properly.

Package and Deployment Wizard can include these additional files in the setup if each component that it needs to distribute has an accompanying *dependency* (.DEP extension) file.

A dependency file is a text file with the same name (but a different extension, of course) as the file whose dependencies it describes. The contents of a dependency file look like an INI file: section headers set off by square brackets ([ ])and individual key entries of the format *Keyname=value*.

The dependency file for the Windows common controls that are implemented in COMCT332.OCX is named COMCT332.DEP, for example. Listing 21.3 shows some of the opening and closing lines from that file.

LISTING 21.3

PART OF A TYPICAL DEPENDENCY FILE

```
[Version]
Version=6.0.0.8096

[msvbvm60.dll]
Register=$(DLLSelfRegister)
Dest=$(WinSysPathSysFile)
Version=6.0.80.91
CABFilename=MSVBVM60.cab
CABDefaultURL=http://activex.microsoft.com/controls/vb6
CABINFFile=MSVBVM60.inf
Uses1=

[STDOLE2.TLB]
Register=$(TLBRegister)
Dest=$(WinSysPathSysFile)
Version=2.30.4256.1
CABFilename=
CABDefaultURL=
CABINFFile=
CABRunFile=
Uses1=OleAut32.dll
Uses2=OlePro32.dll
Uses3=AsycFilt.dll
Uses4=

;...other information omitted for clarity

; Localized Dependencies ------------------------------------

; ** German (DE) ***
; (0007 = German)
;
[COMCT332.OCX <0007>]
Uses1=CmCt3DE.dll
Uses2=

;...other localized dependencies follow (French, Japanese, etc.)
```

If a component of your setup has an accompanying DEP file, Package and Deployment Wizard can include the other files that the component needs in the final setup.

If Package and Deployment Wizard can't find a component's DEP file, it will warn you. You can still continue to create the setup, but the distributed version of your application might not run on a user's system if the component requires other files (such as DLLs) to run, and if these files are missing from the user's system.

NOTE

Modifying VB6DEP.INI If you have additional files that you want to distribute to your users with every VB-based product that you install, you can add the information about these files to the VB6DEP.INI file on your development machine.

To enable bidirectional text display features for VB applications running under versions of Windows that support this feature (such as Arabic Windows), for instance, you can add an entry for VBAME.DLL to VB6DEP.INI.

Package and Deployment Wizard also reads the dependency file named VB6DEP.INI to be sure that it is distributing all the files needed by the VB6 runtime environment.

Microsoft recommends that you always generate a dependency file for an application that you are distributing, especially if the application is a COM component: If future applications use your component, you will need to supply dependency information for your component for those applications' setups to function correctly.

Your application's DEP file will normally be a compendium of all the DEP files that Package and Deployment Wizard found for your application's various components, as well as the contents of the VB6DEP.INI file.

NOTE

VB Dependency Filename in Other Versions of VB Previous versions of VB also provide dependency information for Package and Deployment Wizard, but the name of the dependency file for these earlier versions varies. The VB5 dependency file has a similar name to the VB6 filename, and the dependency filename for earlier versions of VB is SWDEPEND.INI.

SETUP.EXE and Package and Deployment Wizard's Custom Setup

Microsoft's SETUP.EXE utility is the file that the end user will run to install your application. SETUP.EXE itself contains no customized information about a specific application, however. Rather, its job is to prepare the environment for the main setup routine and start the main setup routine running.

SETUP.EXE'S main tasks are as follows:

1. Read the text file SETUP.LST (see previous information).
2. Copy all the files in SETUP.LST's BootStrap Files section to the user's system.
3. Run the file for the main setup routine indicated in SETUP.LST's Bootstrap section.

At this point, the main setup routine (whose default name is SETUP1.EXE) takes over and finishes the installation of your application, copying the other files listed in SETUP.LST and performing any customized tasks.

CUSTOMIZING A STANDARD SETUP

Package and Deployment Wizard's automated process will generate a fully usable, professional setup that you can distribute to users or to Internet developers. Sometimes, however, you will need to go beyond Package and Deployment Wizard's end products and alter the standard setup that it creates. You might need to customize Package and Deployment Wizard's product in situations such as the following:

- The setup process needs to perform special operations, such as providing a dialog box to determine user preferences for your application.
- You need to modify the distribution files slightly, providing a different name or a different version of an existing file or providing an extra supporting file.

Customizing SETUP.LST and Your Application's DEP File

You can modify the information about files that you need to distribute with your application's setup by editing SETUP.LST and your application's DEP file. By editing SETUP.LST and the DEP file, you can avoid having to rerun Package and Deployment Wizard for minor changes in your application's setup configuration.

Suppose, for example, that your application no longer uses a particular custom control, say DBGrid (you have removed all functionality of DBGrid from your application). Instead of rerunning Package and Deployment Wizard just to create a new setup that lacks DBGrid, you could edit the copy of SETUP.LST on your master distribution media. You would just open SETUP.LST in Notepad or another text editor and search for the line in SETUP.LST that refers to DBGrid.OCX. You could remove the line, and then save and exit the file. Then, you could go to the DEP file for your application and remove the reference there for DBGrid.

Customizing the Standard VB Setup Project

You can modify the setup routine for your application by locating and editing its VB source code. Recall that Package and Deployment Wizard provides the setup program (SETUP1.EXE) for your application. Setup1 is a VB application known as the "Setup Toolkit project," and its VB project exists under your VB directory in the path \Wizards\Package and Deployment Wizard\Setup1\SETUP1.VBP.

If you want to customize the Setup1 project (known as the "Setup Toolkit project"), you should save the customized version in a separate directory so that future sessions of Package and Deployment Wizard won't use it for all applications. The Setup Toolkit project contains many files, but the main focus of your customization efforts will probably be the `Load` event of the Setup1 project's main form, `frmSetup1` (see Figure 21.17).

NOTE

The Relationship Between SETUP.EXE and Customized Setup Routines For an explanation of the relationship between Microsoft's SETUP.EXE and your customized setup routine, see the section in this chapter titled "SETUP.EXE and Package and Deployment Wizard's Custom Setup."

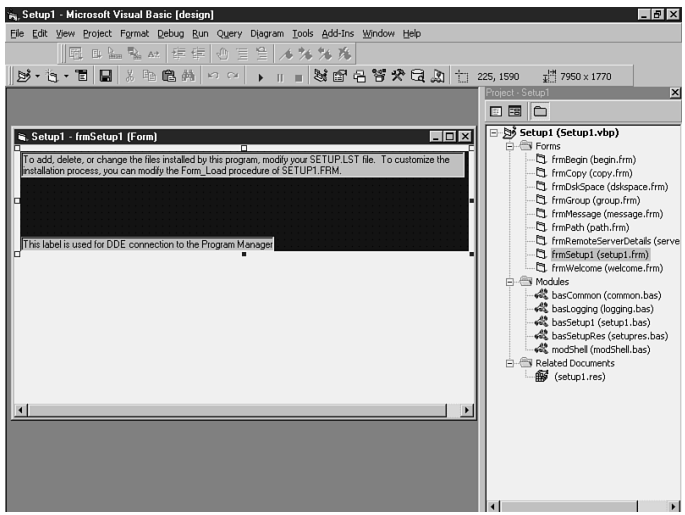


FIGURE 21.17
You can customize an application's setup routine by opening and modifying SETUP1.VBP. The designer for `frmSetup1` is shown here.

Implementing Application Removal

Setup1 creates a log file (ST6UNST.LOG) in the application directory containing information about the modifications that it makes to the system (such as files copied and Windows Registry entries). Setup1 also furnishes a VB6 application removal utility, ST6UNST.EXE.

The application appears on the Add/Remove facility of the Control Panel. If the user calls Add/Remove to remove the application, the system uses ST6UNST.EXE, which reads the log file and undoes the work of the setup.

Application removal can fail under the following circumstances:

- If the user or another application copies, moves, or removes application files and directories manually.
- The application setup log is removed from the application directory.
- The user installs the application more than once to more than one directory.

Registering a Component That Implements DCOM and Configuring DCOM

- Register a component that implements DCOM.
- Configure DCOM on a client computer and on a server computer.

If your application uses remote code components, you must make sure that DCOM is properly installed and configured on the client computer.

If your application implements remote code components, you must make sure that DCOM is properly installed and configured on the server.

To make sure that the correct DCOM components are installed on a client or server computer for a client application or a server component, you call up the Project, Properties dialog box from the VB menu and choose the Component tab. Then check the Component tab's Remote Server Files option, as shown in Figure 21.18.

When you compile either the client or the server project, the compiler will create a VBR (remote support) file in the same folder as the VBP file.

When you use Package and Deployment Wizard to create the component or the client application, Package and Deployment Wizard automatically packages the necessary information into the client and component installation packages to implement DCOM on either end.

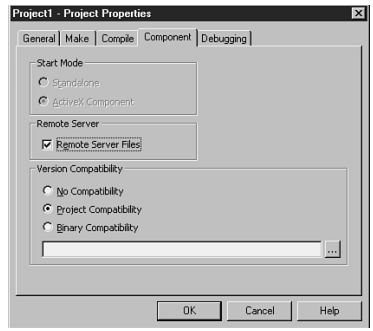


FIGURE 21.18
Setting the Remote Server Files option for a VB project.

DEPLOYING YOUR APPLICATION

- Plan and implement floppy disk–based deployment or compact disc–based deployment for a distributed/desktop application.
- Plan and implement Web-based deployment for a distributed/desktop application.
- Plan and implement network-based deployment for a distributed/desktop application.
- Deploy application updates for distributed applications.

After you have created a setup package, you can deploy your application from a location that's accessible to all potential users. *Deployment* is the term that Microsoft uses to mean moving your setup package to the location from which users can install your application.

You can choose one of several deployment methods:

- Disk-based deployment (Microsoft still calls this deployment to "floppy disks").
- Deployment to a network directory.
- Deployment to CD.
- Web publishing.

The following sections discuss each of these deployment options. Note that Microsoft considers network and CD deployment to be suboptions of the same deployment method, so these two options appear together in one section.

Deploying to Floppy Disks

You deploy to disks in a situation where

- Users are in many locations without any common central connection.
- Not all users have access to the Web.
- Not all users have a CD drive.

A typical application requiring disk deployment would be one whose users were salespeople with various types of laptop PCs scattered across a wide geographic area. You might not be able to guarantee that every salesperson has a CD drive or a fast modem.

As you can see from this example, a disk deployment is a "lowest common denominator" solution. Deploying to disks will cast the widest net around possible users of the application.

To deploy an application to disks, you must take the steps in Step by Step 21.1.

STEP BY STEP

21.1 Deploying an Application to Disks

1. Create an installation package as discussed in the first part of this chapter and make sure that you have saved the packaging script. Note that the installation package must have created multiple CAB files, and the size chosen for the CAB files must be less than or equal to the capacity of the disk media that you will be using.
2. From within the project, run Package and Deployment Wizard, choosing the Deployment icon (the middle icon) on the opening screen.
3. Choose Floppy Disks as the deployment method (see Figure 21.19).

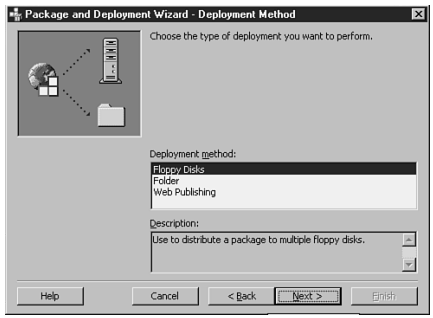


FIGURE 21.19
Choosing the Floppy Disks option.

4. Choose a disk drive (see Figure 21.20). You may also elect to have the wizard format each disk before copying information to the disk.

5. Rename your deployment script from the standard name supplied by the wizard (see Figure 21.21).

FIGURE 21.20▲
Choosing a disk drive.

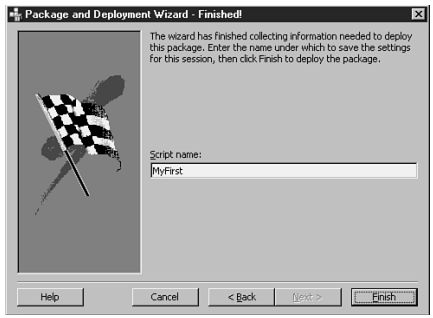


FIGURE 21.21
Assigning a name to the floppy disk deployment script.

6. After you have clicked the Finish button, you're not done! Setup wizard will begin copying information to disks. It prompts you before it begins with each disk (see Figure 21.22). This is also the first time that you know how many disks the deployment will require (the information appears in the prompt).

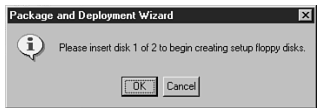


FIGURE 21.22.
The Package and Deployment Wizard prompt to insert a disk.

7. If you chose to format the disks (see step 4), then the wizard will present you with a Format dialog box, as shown in Figure 21.23.

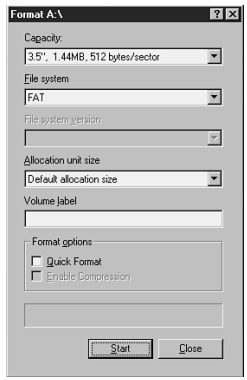


FIGURE 21.23.
Package and Deployment Wizard's Format dialog box.

8. Setup wizard will copy information to each disk (see Figure 21.24).

FIGURE 21.24▲
Package and Deployment Wizard copies information to a deployment disk.

9. You will repeat steps 6 through 8 until you have created all the disks that the deployment requires.

Deploying to a Network Directory or to CDs

Network and CD deployment are essentially the same deployment method from the point of view of Package and Deployment Wizard.

Using this option, you deploy to a single location instead of deploying to multiple disks as you do when deploying to disks.

To implement a network or CD deployment, you should follow the procedure in Step by Step 21.2.

STEP BY STEP

21.2 Deploying to Network or CDs

1. Create an installation package as discussed in the first part of this chapter and make sure that you have saved the packaging script. Note that the installation package may have created a single CAB file—because for practical purposes, your distribution medium (the network or a CD) has unlimited space available.
2. From within the project, run Package and Deployment Wizard, choosing the Deployment icon (the middle icon) on the opening screen.
3. Choose Folder Disks as the deployment method (see Figure 21.25).
4. On the Folder screen, choose a folder to create the deployment. Notice that a Package folder is created underneath the folder you choose (see Figure 21.26)

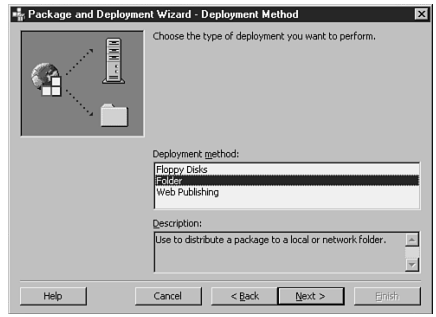
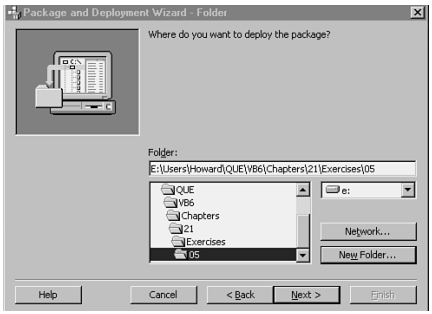


FIGURE 21.25▲
Choosing the Folder option. You also choose this option when you deploy to CDs.



, FIGURE 21.26
Choosing a deployment folder or CD drive.

5. On the Finished screen give a name to the deployment script (see Figure 21.27).

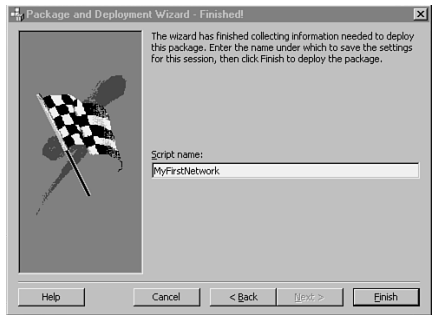


FIGURE 21.27
Naming the deployment script.

Deploying to the Web

At first glance, it might appear that Web deployment would be limited to applications that actually run as Internet applications (such as DHTML applications or ActiveX components embedded in a Web page).

It is possible, however, to distribute just about any VB application over the Web. You just need to package the application's CABs with an HTML page that points to the CAB files, and then place that page on the Web server. When users point their browsers to the page, they will download the application's setup files and the setup will run automatically on their machines.

Because you can deploy almost any application to the Web, Web distribution becomes an attractive choice. It offers several advantages over older forms of application deployment:

- The logistics of application distribution become simpler. You do not have to concern yourself with getting disks or CDs out to every user.
- Management of new software updates is more automatic. Users can get a new version of an application just by pointing their browsers to the setup's Web page.

To deploy a setup package via the Web, follow Step by Step 21.3.

STEP BY STEP

21.3 Deploying on the Web

1. Create an Internet or standard installation package as discussed in the first part of this chapter, and make sure that you have saved the packaging script.
2. From within the project, run Package and Deployment Wizard, choosing the Deployment icon (the middle icon) on the opening screen.
3. Choose Web Publishing as the deployment method (see Figure 21.28).

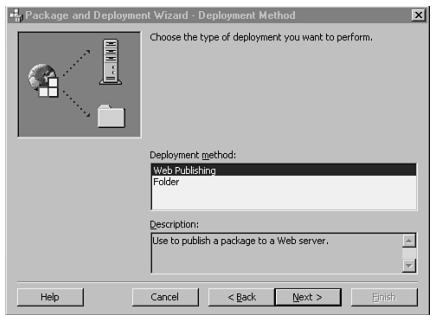


FIGURE 21.28
Choosing Web deployment.

4. The Items to Deploy screen enables you to indicate files to put on the Web deployment site (see Figure 21.29).
5. The Additional Items to Deploy screen enables you to specify more items to deploy (see Figure 21.30).

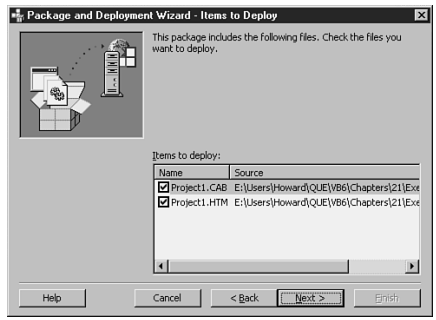


FIGURE 21.29▲
The Items to Deploy dialog box.

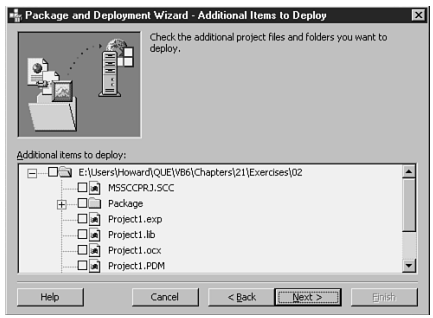


FIGURE 21.30
The Additional Items to Deploy dialog box.

6. On the Web Publishing Site screen (see Figure 21.31), specify the site to which you will deploy the installation.

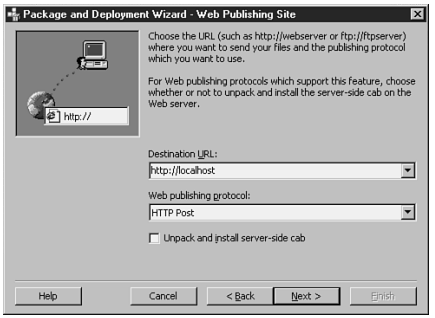


FIGURE 21.31.
Specifying a Web publishing site.

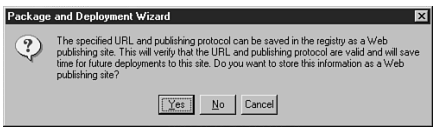


FIGURE 21.32▲
Option to save publishing site information to the Windows Registry.

7. When you click the Next button, the wizard may prompt you to save the publishing site information to the Windows Registry (see Figure 21.32).

8. On the Finished screen, assign a name to the deployment script, as in Figure 21.33.

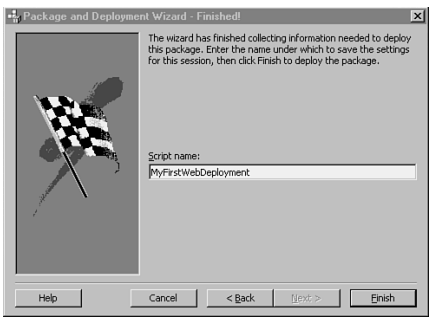


FIGURE 21.33.
Assigning a name to the Web deployment script.

Deploying Updates to Your Application

When you need to make changes to a distributed application, you will get the changes out to users by updating the setup package and redeploying the parts of the setup package that have changed.

There are two major techniques for deploying updates:

‡ **Automatic redeployment**. For this option to work best, you should save the scripts for both the packaging and the deployment. Just rerun the packaging and deployment scripts.

‡ **Manual redeployment**. You can choose one of two levels of automation for manual redeployment:

- Rerun Package and Deployment Wizard for packaging, and then for deployment.
- Manually re-create one or more of the setup package elements, and then manually deploy it (usually by copying one or more files).

CHAPTER SUMMARY

KEY TERMS

- CAB file
- Cabinet file
- DCOM
- DDF file
- DEP file
- Dependency file
- Deployment
- INF file
- Macro
- Remote support file
- VBR file

This chapter covered the following topics:

‡ Creating a standard setup package with Package and Deployment Wizard

‡ Creating an Internet setup package with Package and Deployment Wizard

‡ Creating a dependency file with Package and Deployment Wizard

‡ Standard files used in a Microsoft setup, such as SETUP.EXE, SETUP.LST, and SETUP1.EXE

‡ The significance of dependency files

‡ Customizing a standard setup by modifying SETUP.LST or the standard SETUP1.VBP project

‡ Implementing application removal

‡ Implementing DCOM

‡ Deploying a setup package to disks

‡ Deploying a setup package to a network directory or CDs

‡ Deploying a setup package via the Internet or an intranet

‡ Updating a deployment

APPLY YOUR KNOWLEDGE

Exercises

21.1 Creating a Standard Setup Package

In this exercise, you use Package and Deployment Wizard to create a standard setup package. You will also use the setup package that you create in this exercise in Exercises 21.4 and 21.5.

Estimated Time: 35 minutes

1. Create a new VB6 standard EXE project. Save and compile the project.
2. Invoke the Package and Deployment Wizard from the Add-Ins manager (see Figure 21.1) and choose the Package option from the first screen (see Figure 21.2).
3. Choose the Standard Setup package from the Package Type screen (see Figure 21.3).
4. On the Package Folder screen, create a new folder under the application's folder (see Figure 21.4).
5. View the Included Files screen (see Figure 21.5), but make no changes.
6. On the Cab Options screen (see Figure 21.7), indicate Multiple cabs and choose the cab size (1.44MB, for example) that corresponds to the capacity of disk drives of the machines where users will install your application.
7. View the Installation Title screen (see Figure 21.8) and make changes if you like.
8. View the Start Menu Items screen (see Figure 21.9), making changes if you like.
9. View the Install Locations screen (see Figure 21.11). Modify the Install Location if you like.
10. On the Shared Files screen, make sure that the compiled file's name is checked (see Figure 21.12).
11. On the Finished screen, give a name to the script for this package. Enter the name MyFirst.
12. Go past the Finished screen (see Figure 21.13) and view the Packaging Report screen. When done viewing this screen, close it.
13. The setup is now finished. In Windows Explorer, navigate to the folder that you designated for the package in step 4. View the contents of the project's LST file.
14. Examine the contents of the support directory created under the package's setup folder.
15. View the contents of the project's BAT file, which will call the MakeCab utility to rebuild the CAB file, based on the instructions in the DDF file.
16. View the DDF file, which contains instructions used by the MakeCab utility to create the CAB file.
17. Do not delete this setup package from your machine, as you will use it in Exercises 21.4 and 21.5.

21.2 Creating an Internet Setup Package

In this exercise, you create an Internet setup package, noting the differences between such a package and the standard setup package that you created in Exercise 21.1. You will also use the setup package that you create in this exercise in Exercise 21.6.

Estimated Time: 25 minutes

APPLY YOUR KNOWLEDGE

1. Create a new VB6 ActiveX control project. Save and compile the project.
2. Invoke the Package and Deployment Wizard from the Add-Ins manager (see Figure 21.1) and choose the Package option from the first screen (see Figure 21.2).
3. Choose the Internet Setup package from the Package Type screen (see Figure 21.3).
4. On the Package Folder screen, create a new folder under the application's folder (see Figure 21.4).
5. View the Included Files screen (see Figure 21.5), but make no changes. Note that fewer files are listed for this package than for the standard package of Exercise 21.1.
6. When you proceed beyond the Included Files screen, note that there is no Cab Options screen, Installation Title screen, or Start Menu Items screen as there are in a Standard Package setup.
7. Instead of the Standard Package's Install Locations screen, you will see the File Source screen (see Figure 21.14). In the Files list, select the compiled component's name and notice that you have two file-source options available to you (Include in This Cab and Download from Alternate Web site). Select the VB6Runtime and OLE Automation option in the File list, and you will see that Download from Microsoft Web site is also available (in fact, it's the default).
8. On the Safety Settings screen, manipulate the Safe for Scripting and Safe for Initialization options (see Figure 21.15).
9. Assign a name to your packaging script, such as MyInternetSetup.
10. Go past the Finished screen and view the Packaging Report screen. Close the screen after you have finished looking at it.
11. The Internet Package is now finished. In Windows Explorer, navigate to the folder that you designated for the package in step 4. Note the absence of the SETUP.LST and SETUP.EXE files.
12. Package and Deployment Wizard created an HTML file in the Internet Package's target directory. Use a text editor to view the file.
13. Examine the contents of the support directory created under the package's setup folder. Package and Deployment Wizard has created a BAT file and a DDF file, just as it did for a standard package.
14. Unlike the standard package's Support folder, the Internet package's Support folder contains an INF file. View the contents of the INF file with a text editor. This file will be bundled into the CAB file with some of the support files for your component. The INF file contains instructions that tell the browser where to get the support files not included in the CAB file. The INF file's information should reflect the information that you specified on the File Source screen in step 7.
15. Do not delete this setup package from your machine; you will use it in Exercise 21.6

APPLY YOUR KNOWLEDGE

21.3 Implementing DCOM

In this exercise, you follow the simple steps necessary to implement DCOM for a project.

Estimated Time: 20 minutes

1. Begin an ActiveX DLL project.
2. Choose the Project, Properties menu item and select the Component tab. On the Component tab, check the Remote Server Files option.
3. Compile the project.
4. Run Package and Deployment Wizard to create a standard Package, as described in Exercise 21.1. You have now created a package that can be deployed to a server and can implement DCOM.
5. If you create a client application for your component, you can enable it for DCOM in the same way, following steps 2–4.

21.4 Deploying an Application to Floppy Disks

In this exercise, you use Package and Deployment Wizard to deploy the setup package that you created in Exercise 21.1 to disks. To fully complete the exercise, you will need two 1.44MB disks.

Estimated Time: 20 minutes

1. Open the project you worked with in Exercise 21.1 and run Package and Deployment Wizard from the Add-Ins menu.
2. Click the middle icon (Deploy) on the first screen of Package and Deployment Wizard.
3. On the Package to Deploy screen, make sure that you select the name of the package that you created in Exercise 21.1 (MyFirst) and click the Next button.
4. On the Deployment Method screen, choose the Floppy Disks option and click the Next button (see Figure 21.19).
5. On the Floppy Drive screen, choose the disk drive to which you want to deploy. If you wish, you may check the option to format the disks during deployment. Click the Next button to proceed to the next screen (see Figure 21.20).
6. On the Finished screen, assign a name to the deployment script (see Figure 21.21).
7. Click the Finish button to begin deploying your application to disks.
8. The wizard will prompt you to insert a disk. Notice that the prompt tells you the total number of disks that the deployment will require (see Figure 21.22). Make sure that you have enough disks available.
9. If you chose the Format option in step 5, the wizard will also display a Format dialog box to prompt you for formatting options (see Figure 21.23). Click the Start button to format the disk. After the formatting process has finished, press the Close button on the Formatting dialog box.
10. The wizard will copy one of the CAB files to the disk (see Figure 21.24).
11. After the wizard finishes copying the CAB file, it will prompt you for the next disk. Repeat steps 8 and 9 until the wizard asks you for no more disks.

APPLY YOUR KNOWLEDGE

12. After the wizard finishes copying the last CAB file to disk, you will see a Deployment report. You can close this screen. Your deployment to floppy disks is complete.
13. Examine the contents of the disks where you created the deployment.

21.5 Deploying an Application to a CD-ROM or a Network Drive

In this exercise, you use Package and Deployment Wizard to deploy the setup package that you created in Exercise 21.1 to a CD-ROM or a network drive.

Estimated Time: 20 minutes

1. Open the project you worked with in Exercise 21.1 and run Package and Deployment Wizard from the Add-Ins menu.
2. Click the middle icon (Deploy) on the first screen of Package and Deployment Wizard.
3. On the Package to Deploy screen, make sure that you select the name of the package that you created in Exercise 21.1 (MyFirst) and click the Next button.
4. On the Deployment Method screen, choose the Folder option and click the Next button (see Figure 21.25).
5. The action you take in the Folder dialog box (see Figure 21.26) will depend on whether you are setting up for network or CD distribution:
   - **Network setup**. Browse to the network folder where you want to deploy your setup package. Note that you may create a new folder during the browse, if necessary.
   - **CD setup**. You may either choose to temporarily deploy to a network folder (for later transferral to a CD), or, if you are able to write to CDs from your machine, you may choose the CD drive itself and insert a writeable CD.
6. Click the Next button to proceed to the Finished! screen. Give your deployment script a name (such as **MyFirstNetwork**), as illustrated in Figure 21.27.
7. Click the Next button to view the Deployment report. Close the Report screen. Your network deployment is complete.
8. View the contents of the network folder where you deployed your installation package.

21.6 Deploying an Application on an Intranet or the Internet

In this exercise, you use Package and Deployment Wizard to deploy the setup package that you created in Exercise 21.2 to a Web site (either an intranet or the Internet).

Estimated Time: 20 minutes

1. Open the project you worked with in Exercise 21.1 and run Package and Deployment Wizard from the Add-Ins menu.
2. Click the middle icon (Deploy) on the first screen of Package and Deployment Wizard.
3. On the Package to Deploy screen, make sure that you select the name of the package that you created in Exercise 21.2 (MyInternetSetup) and click the Next button.

APPLY YOUR KNOWLEDGE

4. On the Deployment Method screen, choose the Web Publishing option, and click the Next button (see Figure 21.28).
5. On the Items to Deploy dialog box, you have the chance to add setup package items to the deployment or to remove them. Make no changes here. Click the Next button after viewing the screen (see Figure 21.29).
6. On the Additional Items to Deploy dialog box, you have yet another chance to deploy other files related to your project. Make no changes here. Click the Next button after viewing the screen (see Figure 21.30).
7. On the Web Publishing Site dialog box, enter the name of the Web server where you want to deploy your installation package. Click the Next button to continue (see Figure 21.31).
8. The wizard may prompt you to save information about the Web publishing site to the Windows Registry. You do not need to do so. (Perhaps in this case you will elect not to clutter your Registry with the results of an exercise.) You can respond to this dialog box, and then continue (see Figure 21.32).
9. On the Finish screen, you can give a name to your deployment script (see Figure 21.33).
10. Click the Finish button to finish the Web server deployment.

Review Questions

1. What is the purpose of a dependency file?
2. How can you customize the behavior of a standard setup routine?
3. How do setup routines created with Package and Deployment Wizard enable possible removal of the application?
4. What is the standard compression format for files created with the Package and Deployment Wizard?
5. What basic steps must you take to install and configure DCOM on a client and server?

Exam Questions

1. You can use the Manage Scripts icon in Package and Deployment Wizard to

   A. Manipulate the default behavior of a setup project.

   B. Manipulate the creation of Internet download scripts.

   C. Manipulate Internet download scripts directly.

   D. Manipulate Package and Deployment Wizard default behavior when you run Package and Deployment Wizard in the future.

2. The name of the file that contains the dependency information for Visual Basic is

   A. SWDEPEND.INI

   B. VB6DEP.INI

   C. SWSETUP.DEP

   D. VB6SETUP.DEP

3. You create a project that is installed by a network-based install and that includes your own custom in-process COM component. The component is

APPLY YOUR KNOWLEDGE

installed to reside on each user's workstation. Later, you need to make a change to the component because of a bug in the way it runs. The best way to get the component changed in user installs is to

A. Send the DLL as an attachment to emails to all the users, asking them to copy it to the appropriate place on their local hard drives.

B. Update the CAB file on the network, and ask all users to rerun the setup.

C. Put the DLL on each user's hard drive yourself, and edit the Windows Registry.

D. Unregister the old copy of the component with the RegSvr32 utility, place the new copy on each user's hard drive, and then reregister with RegSvr32.

4. If you want to distribute a standard EXE application to users on disks, you should use Package and Deployment Wizard to

A. Specify a single CAB file.

B. Specify floppy disks.

C. Specify multiple directories.

D. Specify multiple CAB files.

5. You want to display your own custom graphics ("billboards") to users while the setup routine runs for your application. The best way to do this is to

A. Modify the DDF file belonging to each CAB file.

B. Modify the setup package's INF file.

C. Modify the VB source code for SETUP1.EXE.

D. Modify the setup package's SETUP.LST file.

6. A user installs your application once and then installs it again in a different directory. When the user runs Control Panel's Application Removal utility against the application,

A. The first install is removed.

B. The second install is removed.

C. Both installs are removed.

D. The Application Removal utility fails.

7. You can create remote support (VBR) files for distribution with a DCOM project by

A. Using a text editor to manually edit a VBR file of the same name as the project.

B. Compiling the project with the correct options selected.

C. Letting Package and Deployment Wizard create the VBR file automatically.

D. Using Microsoft's Compress utility to create the VBR file.

8. To install an application on end-users' systems that can use components through DCOM, you should

A. Use Package and Deployment Wizard to specify the appropriate setup files.

B. Use RegSvr32 to register the component.

C. Edit the Windows Registry.

D. Mark the appropriate option in the project

APPLY YOUR KNOWLEDGE

before you compile it.

9. When you deploy to floppy disks, you must

   A. Specify a single CAB file in the packaging phase.

   B. Specify multiple CAB files in the packaging phase.

   C. Specify either single or multiple CAB files in the packaging phase.

   D. Specify a destination directory for the deployment.

Answers to Review Questions

1. A dependency file specifies the supporting files that a particular file needs to be successfully installed on a system. See "Dependency Information in DEP Files."

2. You customize the behavior of a standard setup routine by modifying the standard VB setup project, the SETUP.LST file, or dependency files. See "Customizing a Standard Setup."

3. A VB6 setup package furnishes a copy of ST6UNST.EXE to the host system. The VB setup package creates a log file, ST6UNST.LOG, in the application directory. When a user uses the Windows Add/Remove utility to remove the application, ST6UNST.EXE runs, using the information in the log file to remove files and registry entries. See "Implementing Application Removal."

4. The standard compression format for Package and Deployment Wizard files is the CAB (cabinet) format. See "Creating a Standard Setup Package."

5. To enable a client or a server to use DCOM, create a client application or component, check the Remote Server Files option on the Project, Properties Component tab, compile the project (this will create a VBR file), create a setup package for the project, and install it on the client or the server, whichever is appropriate. See "Registering a Component That Implements DCOM and Configuring DCOM."

Answers to Exam Questions

1. **D**. You can use the Manage Scripts icon in Package and Deployment Wizard to manipulate Package and Deployment Wizard's default behavior when you run it in the future. For more information, see the section titled "Starting Package and Deployment Wizard and Choosing the Type of Package."

2. **B**. The name of the VB6 dependency file is VB6DEP.INI. For more information, see the section titled "Dependency Information in DEP Files."

3. **B**. The best way to distribute a changed component to local installations with a network-based install is to update the CAB file containing the component on the network server and ask all users to rerun the setup. For more information, see the section titled "Creating an Internet Setup Package."

4. **D**. Specify multiple CAB files to distribute a standard EXE application to users on disks. You can specify the size of the CAB files, and then

APPLY YOUR KNOWLEDGE

copy the CAB files that the Package and Deployment Wizard creates to the disks. For more information, see the sections titled "Creating a Standard Setup Package" and "Deploying to Floppy Disks."

5. **C** To change the behavior of the custom setup routine as it runs, you can modify the VB source code for SETUP1.EXE (SETUP1.VBP). For more information, see the section titled "Customizing the Standard VB Setup Project."

6. **D**. The Application Removal utility will fail if the user attempts to remove an application that has been installed twice in separate directories. For more information, see the section titled "Implementing Application Removal."

7. **B**. You can create the necessary remote support (VBR) files for a project that uses DCOM by compiling the project with the Remote Server Files option checked in the Components tab of the Project, Properties dialog box. This will create the necessary files in the same folder as the project, and Package and Deployment Wizard will distribute these files as necessary. For more information, see the section titled "Registering a Component That Implements DCOM and Configuring DCOM."

8. **D**. To implement DCOM in an application, you should mark the Remote Server Files option on the Compile tab of the Project, Properties dialog box before you compile the project. For more information, see the sections titled "Registering a Component That Implements DCOM and Configuring DCOM" and "Deploying Updates to Your Application."

9. **B**. To deploy to disks, you need to specify multiple CABs when you package the setup. Each CAB will go on a different disk. If you specify only one CAB, the CAB may be too big to fit on the disk. You don't specify a folder during deployment to disks—this option is only for network/CD deployment. You do, however, specify the drive for the disks. For more information, see the sections titled "Deploying to Floppy Disks" and "Deploying to a Network Directory or to CDs."

PART

II

FINAL REVIEW

Fast Facts

Study and Exam Prep Tips

Practice Exams

Now that you have read through this book, worked through the exercises, and acquired as much hands-on experience using VB6 as you could, you are ready for the exam. This last chapter is designed as a "final cram in the parking lot" before you walk into the exam. You can't reread the whole book in an hour, but you will be able to read this chapter in that time.

This chapter is organized by objective category, giving you not just a summary, but a review of the most important points from this book. Remember that this is meant to be a review of concepts and a trigger for you to remember those tidbits of information you will need when taking the exam. If you know what is in here and the concepts that stand behind it, chances are the exam will be a snap.

Fast Facts

Developing the Conceptual and Logical Design

Given a conceptual design, apply the principles of modular design to derive the components and services of the logical design (70-175).

- A conceptual design is based on user scenarios.
- Part of the logical design process involves deriving business objects from the user scenarios of the conceptual design.

Deriving the Physical Design

Assess the potential impact of the logical design on performance, maintainability, extensibility, scalability, availability, and security (70-175 and 70-176).

- The user-interface component of an application normally resides on client workstations, because it is the part that actually provides the user connection to the rest of the system.
- A "thin client" (that is, a workstation client that implements as little functionality as possible) can improve a software solution's maintainability, because more processing will be implemented on servers. Such centralization of functionality means that there are less locations where software changes have to be distributed. By putting more processing burden on servers, however, performance can degrade dramatically as more demand is placed on the server through the addition of new users.
- A business-rules component is the best place to enforce constraints such as customer credit enforcement and shipping-charge rules. Both constraints are part of the way that the organization does business. These rules could change over time, or even change in different directions for different parts of the same organization.
- Putting a COM component for business rules or a data-access component on individual workstations would generally be an inferior solution, because it would give you the proverbial "maintenance nightmare" by requiring many individuals (at assuredly varying levels of system competence) to perfectly perform the same action at the same time when you needed to distribute changes.
- It is sometimes appropriate to use a "thick client" solution that implements user interface and business rules on the client workstation to best address concerns of scalability and performance. The key to both these considerations is the fact that such a design will offload a lot of processing to client workstations. If the workstations are powerful enough (and so capable of handling the extra work) and there is little maintenance anticipated to the system, a thin client can be viable. The network will be less likely to bottleneck because more users are quickly added to the system (so scalability is served), and performance will also degrade less because individual workstations will be more responsible for the performance for each user. This solution definitely does not address maintainability, because (1) it will be harder to make changes to business logic components that are scattered over many user workstations and (2) are intertwined with the user interface.
- An IIS application solution would favor maintainability (it is centralized and therefore easily changeable), availability (any user with a Web connection and any major Web browser), and performance from the user's point of view, although not for the server, because the server will be doing most of the work. It might not be the best solution for scalability, because it is more server-intensive than a DHTML application; it is probably not very easily extensible, either, because its logic is centered around one type of solution. Security could also be an issue, because users are connecting to your server and your corporate data through the Internet.

Design Visual Basic components to access data from a database in a multitier application (70-175 and 70-176).

- Data-access components in an n-tier model can be a component such as a COM server (ActiveX EXE or DLL) that sits between the business objects and the database engine and uses data-aware classes.
- It is usually *not* a good idea to implement data-access components in the stored procedures and triggers of the DBMS itself.
- The data-access interface of an application normally resides on the server so that it can provide consistent service and server resource management. It is also more maintainable if it resides in a single central location.
- A COM component with data-aware object classes is the best solution for implementing a data-access component whose data-access platform will change in the future. This provides the best maintainability, because it isolates the other tiers from needing to be aware of the type of data access that's needed.
- A COM component with data-aware classes would be the best choice for implementing a data access tier for centralized users. Though an IIS application could be a *part* of the solution, it is not really appropriate for a data access tier, but would work better for a user interface tier. The same could be said of a DHTML application and of a standard client-side executable.
- An independent data-access component is the best place to enforce rules of referential integrity. If the logical design calls for no separate data-access component, stored procedures and triggers in the actual database would be the best choice. When an independent data-access component is one of your options, however, you should normally favor that location.

Design the properties, methods, and events of components (70-175 and 70-176).

- It is best to keep the functions of data access and business rules separate in an n-tier application, to separate the functions of the data-services tier itself (data-integrity rules) with the data-access methods specific to the application itself.
- Data entry and display rules are best implemented as part of the user-interface component.

Establishing the Development Environment

Establish the environment for source-code version control (70-175 and 70-176).

- Pinning enables you to use an earlier version of a file (not the current version) in a VSS project. It is good for creating maintenance releases of a project while new development on the same project is going on at the same time.
- A developer can check a file out of Visual SourceSafe to get a modifiable copy.

- You can use the share, pin, and branch model to create a maintenance or "service pack" release on the production version of an application that already is under development for a new major version.
- Developers can check working code in at the end of each day to make sure that the project always has the latest working copies of source code.
- You can share some or all of a project's files to make it the basis for a second, new project that needs to share changes with the first project. If there is a need in the future for the projects to diverge their copies of these files, you could branch the second project at that time.
- Branching a project cuts off the link between changes in the projects.
- To keep track of all of a project's older version's source code as a group, you can label the older version.
- To move information from one SourceSafe database, you can archive it from the first database and restore it to the second database. You should never attempt to directly manipulate the contents of the VSS database.

Install and configure Visual Basic for developing desktop/distributed applications (70-175 and 70-176).

- Only the Enterprise Edition of VB6 actually includes SQL Server, although it is possible to access SQL Server data from the other editions.
- The lowest-end product that provides you with all the tools necessary to develop a solution using SQL Server is the Enterprise Edition. Although the other editions enable you to connect to SQL Server data, they don't actually come with a copy of SQL Server.

Configure a server computer to run Microsoft Transaction Server (MTS) (70-175)

- To run Transaction Server in Windows NT, you are required to have at least Service Pack 3 installed.
- Choose a custom install, and be sure to select the option under Transaction Server that includes the developer documentation. Developer documentation is not included by default.
- A typical installation of the NT Server 4 Option Pack will install the MTS runtime environment and everything you need to perform administrative tasks on your MTS machine, which includes the MTS Explorer and the core documentation. The only thing a Typical Installation lacks is the developer samples and documentation, which can only be installed through a custom installation.
- Windows 95 require DCOM support to be installed if you intend to call MTS objects on it from a remote machine. DCOM support is not built in to Windows 95, but can be downloaded for free from Microsoft.
- Anyone can administer an MTS machine immediately after it is installed. To limit access, you must first add users to the Administrator role for the system package.
- To limit a user from accessing the MTS Explorer to do administrative tasks, they should be added to the Reader role of the System package. Additionally, their user account, or any NT group of which they are a member, must not be mapped to the Administrator role.

Configure a client computer to use an MTS component (70-175).

- In general, coding a client application that calls MTS components need be no different from coding a client application that calls other types of COM objects. There are no special considerations. The only thing that needs to be done before coding begins, is that the client has to be configured to allow the developer to reference the MTS component.
- A client setup package would be run on a machine that is intended to be used for developing with the components in the MTS package, or that will be running software that uses the components in this package. This is true for both Windows client software and for Web servers that will call the component.
- If a client workstation is running software that makes calls to MTS components, it is important to run the client setup package on the workstation.
- Anytime you modify the contents of a package, such as by adding new components, you must re-export the package. This will cause the client setup package to be rebuilt to support the changes and additions to the package.

Creating User Services

Implement navigational design (70-175 and 70-176).

- The two methods used by Visual Basic to create menus for an application are the VB Menu Editor and the Win32 API. Both methods can be used from VB to generate menu systems. The built-in Menu Editor is a simple dialog box that enables the user to create a hierarchy of menu items and menu item order. The Win32 API is an external set of functions provided by the operating system, and allows for a wide variety of functions.
- The term top-level menu is used to refer to a menu item found directly under a window's title bar. This menu item is used to group other items into a submenu, which will appear under the top-level item when it is selected.
- When creating menus with the Menu Editor, the programmer uses the left and right arrows. The arrows allow the menu hierarchy to be customized as required. The up and down arrows of the editor allow the items to be ordered from top to bottom.
- The MouseUp event can be used to determine which mouse button has been clicked. By using a specific object's MouseUp event, the programmer can determine whether the right mouse button was used. If so, the form's PopupMenu method can be called.
- The menu provided when a user right-clicks on an object has a variety of names. One of the most common terms used is the "pop-up menu." Other terms are the "context-sensitive menu" and "right mouse menu." Certain objects and the operating system provide the menu or they can be created in Visual Basic.
- By setting the index value of one menu item at design time, new items can be dynamically loaded and controlled through code.
- The Unload statement can only be used to remove instances of menu items or other control array elements created at runtime. If the first element in the array—assuming it is the design time item—is passed to Unload , an application error will occur: CanÕt unload controls created at design time .
- To place an instance of a control on a form, double-click on the control's icon in the toolbox.

A second technique will be to single-click on the control and then use the mouse to draw its rectangular outline on the container's surface.

‡ Some objects that can contain controls (Container objects) are Forms, PictureBoxes, and Frames. Note that Image controls cannot contain other controls.

‡ The default property of a CommandButton is the Value property, a True/False property that you can set in code to fire the CommandButton's Click event.

‡ The default property of a Label is the Caption property, which represents the text that the user sees on the Label's surface.

‡ The default property of a TextBox is the Text property, which represents editable text that appears in the TextBox.

‡ When you rename a control, a brand-new event procedure appears with a new name based on the new control's name. If you wrote code in the event procedure before the name change, that code stays in the procedure with the old name. Therefore, any code that you wrote in the old event procedure before you changed the control's name is no longer associated with the renamed control's event.

‡ You can specify that a menu control will be a separator bar by specifying a dash as the Caption property. This is the only way to get a separator bar.

‡ The value of the Name property can't be changed at runtime. It can be read.

‡ To prevent the user from being able to give focus to a control under any circumstances, you can set the control's Enabled property to False. Setting TabStop will only affect the user's navigation with the Tab key. The user can still use the mouse as long as the Enabled property is True.

‡ When a TextBox's Enabled property is False, the text in the box will be grayed and the user won't be able to set focus to the TextBox. When the Locked property is True, the user can set focus to the TextBox (assuming that Enabled is True) and navigate through its contents, but cannot make changes.

‡ An access key for a TextBox control can be provided in an accompanying Label control, provided the Label immediately precedes the TextBox in the Tab order.

Create data input forms and dialog boxes (70-175 and 70-176).

‡ The four styles of the ListView are Icon, Small Icon, List, and Report. They are set by using the View property of the object.

‡ Only a single image format can be used in an ImageList. 16×16, 32×32, 48×48 icons, and bitmaps must be loaded into separate ImageLists. After an image of one format is loaded, images of different formats cannot be loaded.

‡ The ShowTips property of the toolbar dictates whether ToolTips are displayed.

‡ The ToolTipText property of the Button object on the toolbar identifies the text that will be displayed when the user rests the mouse pointer over a button.

‡ Each Visual Basic application has one Controls Collection per loaded form. Controls Collections are created and maintained for you automatically.

‡ To create text boxes dynamically from a control array, you must place at least one text box on your form at design time, and you must also set the Index property of that control to 0 to create a control array.

- The Load statement and Show statement will both load a Form object into memory. Load will load the form, but not show it. Show will automatically load the form and display the form onscreen. Any reference to a Form object's intrinsic members will cause an implied load in Visual Basic. The form will be loaded, but not shown.
- The Unload method will remove a form from memory. When the form is reloaded, all controls contained on the form will be re-initialized. Using the Hide method rather than Unload will hide the form from the display, but will still allow the programmatic reference to the controls as set by the user.
- When forms are created and then saved, two files can be generated. The first file is an FRM ASCII text file that contains information related to the form, the form's objects, properties of the objects, and any source code for those objects. The second file is a binary file that contains graphic information related to the form. If a picture control is used, an FRX file will contain the graphics information required by the control.
- The keyword used to reference the currently running form is ME.
- The keyword used to create a runtime version of an object created at design time is NEW. An object variable is declared or initialized as a NEW object. This tells VB to create another instance of the Form object at runtime. The following code demonstrates this syntax: Dim x as New Form1.
- The Forms Collection contains references to forms that are loaded into memory through design time and runtime actions. If a project contains multiple forms, but a particular form is not loaded into memory, the Forms Collection will not contain a reference to that form.
- The Visual Basic Forms Collection has only one property, Count, and it returns the total number of forms that are currently loaded in memory.
- A VB collection provides an item number that is an ordinal index value assigned to each individual object as it is added to the collection. You can use the item number to programmatically access individual forms and their associated properties. For instance, you could refer to the Caption property of the first form in the collection with this syntax: Forms(0).Caption.
- ObjectEvent is an event procedure for objects of type VBObjectExtender. When you add an ActiveX control to a form with Controls.Add, you can declare the control to be of type VBObjectExtender, using the WithEvents keyword. You can then program the ObjectEvent procedure to react to events raised by the control.
- ActiveX controls are implemented in files with the extension .OCX. The .VBX extension was used for 16-bit controls in older versions of VB.
- TLB files are type library files and don't implement ActiveX controls.
- The Sorted, SortKey, and SortOrder properties determine whether, and how, ListItems will be sorted in a ListView.
- Icons (*.ICO) and bitmaps (*.BMP) files can be loaded into the ImageList control.
- Images used by the ToolBar control can only come from an ImageList control that has been placed on the same form.
- Status information is displayed in one or more Panel objects on a StatusBar.
- You can only remove controls that you have created dynamically.
- The If TypeOf statement can be used to determine the class of an object. It is important to check the type before referencing any properties or methods of an object to avoid runtime errors.

- An ImageList contains ListImage objects. ListImage objects are referenced through the ListImages collection.
- The valid relationships of a ListItem to new items are tvwFirst, tvwLast, tvwNext, tvwPrevious, and tvwChild.
- Key values of the ListItems collection (and all collections) must be strings.
- The number of columns in a ListView control is controlled by the number of objects in the ColumnHeaders collection.
- The Style property controls the behavior of a button in the Toolbar control. Valid values include tbrDefault, tbrCheck, tbrButtonGroup, tbrSeparator, and tbrPlaceholder.
- If you use a For I = ... statement to access the elements of a Controls Collection, the index value of Controls Collection ranges from 0 to n–1, where n is the number of controls on the form (as returned by Controls.Count).
- The Load statement can be used to explicitly load a Form object into memory. If the Hide method of the form is used, an implied Load would happen first followed directly by hiding the form. Both would load a form into memory.
- The Hide statement can be used to remove a Form object from the screen view, but keep it loaded in memory. The Unload statement will remove it from the screen, but will also remove it from memory.
- To specify the Startup object of a VB project, the VB IDE provides the Project, Project Properties dialog boxes. The Startup object combo box can be found under the General tab.
- The keyword NEW is used to create a runtime Form object based on a template form. This instructs VB to create another new object based on the object name following the NEW keyword. You can also use the keyword NEW to create a new instance of any object variable type. When dimensioning an object variable, it can have any name you choose. The key is to Dim ÔvarÕ as NEW object. The NEW keyword will create a runtime object from the object name following NEW.
- The Form Collection's Count property, or the Count property of any collection, returns the total number of loaded forms in the collection. The collection's first item index is 0, however, which means that the highest index in the Form Collection would be Forms.Count – 1. The Form and Controls Collections always start at element 0; as items are removed, other items are reorganized to lower numbers.
- When you use the Controls.Add method, you must specify the ProgID for all controls. When specifying the ProgID for most intrinsic controls, the ProgID is a string composed of ÒVB.Ó plus the programmatic name of the control type (for example, ÒVB.CommandButtonÓ or ÒVB.LabelÓ). You can find the ProgID for ActiveX controls by looking in vendor documentation or by searching the Windows Registry.
- When you dynamically add a control, you should not access standard properties through the Object property, which provides access to custom properties only.
- When you dynamically add controls to the Controls Collection, you must use the VBObjectExtender data type for programming with any ActiveX control. When you program with intrinsic controls, you can use the object model provided for the control by the VB environment.
- A license key to use with the Controls.Add method is the second argument to the Licenses.Add method and must be licensed to you, in order for you to use it legally with software that you distribute to end users. You must provide the license key by invoking Licenses.Add on all user machines.

Write code that validates user input (70-175 and 70-176).

- GotFocus and LostFocus events are no longer necessary for field validation in VB6, because VB6 has introduced the Validate event for controls.
- You can prevent changes to TextBox by setting its Locked property to True when there might be more data in the TextBox than the user could see, and you wanted to allow the user to set focus to the TextBox to scroll through the data.
- A control's Validate event will fire when the user attempts to set focus to another control whose CausesValidation property is True .
- The form's KeyPreview property by default is False . Setting it to True enables the form's KeyUp, KeyDown, and KeyPress events.
- The KeyPress event detects a character; the KeyUp and KeyDown events detect physical keystrokes.
- A control's Validate event fires when the user sets focus to another control on the same form whose CausesValidation property is True . The user does not have to make a change in order for the event to fire. Firing of the Validate event depends on the setting of the CausesValidation property of the control that is receiving focus (not the control that loses focus).
- To process keystrokes at the form-wide level, you must set the form's KeyPreview property to True and program at least one of the KeyDown, KeyUp, or KeyPress events.
- The timing of the KeyPress event is before KeyUp, but after KeyDown.
- You can programmatically cancel a user's keystroke by setting the KeyPress event's parameter (known as KeyAscii) to 0. KeyAscii is of type integer.
- A line of code in the KeyUp or KeyDown event procedure that checks to see whether Ctrl was one of the Shift keys being pressed when F3 was pressed might read:

```
If (KeyCode = vbKeyF3) And (Shift And
➥vbCtrlMask) Then
```

  To find out whether a particular key press caused the KeyUp or KeyDown event to fire, you must check the value of the KeyCode parameter, comparing it with the appropriate VB internal constant (in this case, vbKeyF3). To check the state of the Shift keys (Ctrl, Alt, and Shift) in the KeyUp or KeyDown events, you must use the Shift parameter. The Shift parameter is a bit mask containing information about the state of all three Shift keys. To extract information about a single Shift key, you can AND the corresponding VB internal constant for the key with the Shift parameter, as the preceding example does with (Shift AND vbCtrlMask).

Write code that processes data entered on a form (70-175 and 70-176).

- Invoking the End statement will immediately terminate the application without running any further events. Events such as Form_Unload or Terminate events will not run.
- The "Stop button" in the VB IDE has the same effect as calling the End statement from code. Therefore, if you press this button to end a design time instance of your program, you will not fire the Unload events of forms or of other ending events such as QueryUnload and Terminate .

‡ You should put code in the Initialize event procedure to assign the beginning values of the form's Public variables, or of Private variables that represent the stored values of form custom properties (properties implemented with Property procedures). This will make the form's behavior consistent with other VB classes, because the Initialize event behaves like the Initialize event of any other VB class.

‡ The Show method will cause an *implicit load* of a form if the form was not already in memory, and so will fire the Load event. If the form was already in memory, the Show method merely makes it visible, but does not fire the Load event. Depending on the argument you pass to the Show method, the form will display modally as a dialog box (vbModal) or modelessly (vbModeless, the default). Note that referring to an intrinsic method or property of a form can cause an implicit load.

‡ You should put code in the Activate event procedure if a) you want it to run every time the user makes the form the active form in the application, or b) if you need to perform initialization tasks that require the form to already be loaded in memory (such as drawing graphics or using a data control's connection). Otherwise, it is okay to put initialization code in the form_Load event procedure.

‡ The Activate and DeActivate events fire whenever focus changes in the current application between the current form and another form. The GotFocus and LostFocus events only fire if there is no other object on the form capable of receiving focus.

‡ Although DeActivate and LostFocus fire when the form loses its status as the active form, a form is the active form only with respect to the application where it is running. So, when the user moves to another application, the active form of the current application does not change and these two events do not fire.

‡ A form's DeActivate event will *not* fire when the user navigates to another application with the mouse.

‡ You can cause a form's Terminate event to fire by unloading the form and then setting the form equal to Nothing in code. If you have any other Form object variables that refer to the form, they must be set to Nothing as well before Terminate will fire.

‡ The Terminate event fires when you set the form to Nothing after the Unload event begins. The Terminate event cannot fire before the form unloads, and does not fire until the form is destroyed by setting it to Nothing. The Unload event by itself does not destroy a form completely, because it leaves the form's properties in memory. Therefore, the Unload by itself cannot trigger the firing of the Terminate event.

‡ The QueryUnload and Unload events both have a Cancel parameter, which the programmer can set to True to halt the unloading action.

‡ The QueryUnload event has an UnloadMode parameter, whose purpose is to show why the unloading is taking place.

‡ The QueryUnload and Unload events fire at different times in an MDI application: when the MDI parent form unloads, events happen in the following order: 1) The MDI parent's QueryUnload; 2) all the loaded children's QueryUnload events; 3) all the loaded children's Unload; 4) the Unload event of the MDI parent.

‡ It is okay to call another form's Show method from a Load event procedure. The first form ends up as the active form when everything is finished.

‡ A form's QueryUnload event always precedes its Unload event. The QueryUnload event receives the Cancel and UnloadMode parameter; the Unload event receives only the Cancel parameter.

Note that, in the case of MDI child forms, a form's Unload event might not directly follow its QueryUnload event: When the main MDI form unloads, all child form QueryUnload events happen together, followed by all child form Unload events.

‡ A DeActivate event fires only when the currently active form or an object on the currently active form loses focus to another form in the current application. DeActivate will not occur when a form unloads. Neither will DeActivate occur when an application terminates.

‡ Although making any reference to a form's properties or methods will normally cause an implicit Load, there is an exception to this rule for calls to a form's *custom* members (that is, properties implemented as Public variables, methods implemented as Public procedures, or properties implemented with Property procedures). In such cases, no implicit Load occurs. The Initialize event will still run in such cases, however, provided the form is not already instantiated.

Add an ActiveX control to the ToolBox (70-175 and 70-176).

‡ To add an ActiveX control to the ToolBox, choose Project, Components from the VB menu. See "Add an ActiveX Control to the ToolBox."

‡ On the Project, Properties Make tab, the option Remove Information about unused ActiveX Controls should be unchecked to prevent a runtime error when adding an element to the Controls Collection, if that control type is available in the ToolBox but has no design time instance on the form. There is only a problem of this nature when the control appears in the ToolBox. See "Keeping a Reference in the Project to an ActiveX Control."

Create dynamic Web Pages by using Active Server Pages (ASP) and Web classes (70-175).

‡ An IIS (or Web Class) application runs on a Web server. The Web server invokes an instance of an IIS application to modify HTML pages that it sends to clients. Clients have no awareness of the IIS application.

‡ The tag pair <%...%> demarcates ASP code within a Web page file.

‡ The ProcessTag event fires each time that the IIS application encounters a pair of substitution tags in an HTML template.

‡ The UserEvent event of a WebClass fires when a dynamically named event fires. The UserEvent procedure's parameter gives the name of the event that just fired.

‡ The proper syntax for referring to a WebClass tag in an HTML template would be

```
<WC@MYTAG>ELIZABETH</WC@MYTAG>
```

assuming that WC@is the TagPreÞx for that Web Template item. You must use an HTMLtag pair that includes the full name of the tag. The tag pair surrounds the default initial value of the tag. The tag name includes the Web Template item's Tag PreÞx .

‡ No extra files are needed on the user's machine for an IIS application. Recall that IIS applications run server side to help the Web browser prepare a standard HTML page. DHTML applications, of course, are a different story, as they implement an ActiveX DLL that runs client side. Some special files may be required on the Web server only (such as MSWCRRUN.DLL and the application's compiled DLL).

- An HTML file that you use as an HTML template in an IIS application should be created in a directory separate from the project's development directory. When you run, save, or compile the project for the first time after adding the template to the project, VB makes a copy of the template in the project's directory (the directory where the project's .VBP file resides).
- Events defined by the programmer within the WebClass project get their own event procedures (and so UserEvent doesn't fire), and the WebItem's default event (which you can implement by calling URLFor with only one argument) is the Respond event.
- The ProcessTag event of the HTML template object may fire as an indirect result of the WriteTemplate method (when the Web server reads the tags in the template).
- You use the TagPrePx property to specify the substitution tag prefix for an HTML Template WebItem.
- The ProcessTag event's first parameter specifies the name of a single tag. ProcessTag fires separately for each substitution tag pair encountered in an HTML template, and so only handles one tag at a time.
- The UserEvent procedure runs only for events whose names were generated on the fly in HTML code. You can create such calls with syntax such as

```
Response.Write Ò<A HREF=ÓÓÓ & _
    URLFor(MyItem) & ÒÓÓ>MyItem</A><BR>Ó
```

  in the Start event of the WebClass.
- To enable the firing of a custom WebItem's Respond event, you can put code such as

```
Response.Write Ò<A HREF=ÓÓÓ & _
    URLFor(MyItem, MyVarName) &
➥ÒÓÓ>MyItem</A><BR>Ó
```

  where MyVarName is a variable name that will cause the firing of a like-named event. You then can write event-handling code to detect the custom event names in the UserEvent procedure of the WebItem.

Create a Web page by using the DHTML Page Designer to dynamically change attributes of elements, change content, change styles, and position elements (70-175 and 70-176).

- A DHTML application runs on the same machine as the end-user's browser (a Web client). The DHTML application is in the form of an ActiveX DLL that downloads with a Web page to the browser.
- You often must refer to the Style property of a DHTML page element to make visible formatting changes to a DHTML page element. For example, you must write the code

```
TextField2.Style.BorderStyle = ÒnoneÓ
```

- You can manipulate a background color of a DHTML page TextField element named TextField1 with the syntax TextField1.Style.Color.
- The property names for DHTML control objects are not always the same as the corresponding standard VB control property names. For example, the BackColor property in straight VB would correspond to the .style.BackGroundColor in DHTML.

- The event names for DHTML control objects are usually not the same as the corresponding standard VB event names. The DHTML event names are often the same as the VB event names, except that the DHTML event names are prefixed with the word *on*. The DHTML events—onclick, onmouseup, onkeypress, and onfocus, for example—correspond to the VB events Click, MouseUp, KeyPress, and Focus, respectively. An exception to this general rule is the DHTML onblur event, which corresponds to the VB LostFocus event. Several events that have the same name in DHTML and standard VB are Load, UnLoad, Terminate, and Initialize.
- You can refer to the InnerText property of a standard HTML object to change its displayed text in a DHTML application without affecting the visible format.
- Changing the OuterText property would perform the text replacement, but it would possibly change the visible formatting of the element, because it would replace the HTML tags surrounding the element.

Use data binding to display and manipulate data from a data source (70-175 and 70-176).

- Connection and Command objects can be placed on the surface of a Data Environment Designer.
- In a Data Environment setting, the Recordset object's name is automatically derived from its corresponding Command object's name by placing the letters *rs* before the Command object's name.
- You can place a DataGrid on a form that is automatically bound to the Recordset of a Data Environment's Command object by dragging the Command object with the right mouse button from the Data Environment to the form.
- A VB control's DataMember property refers to a Command object in a Data Environment. The Datasource property can refer to an ADO Data Control, a Data Environment, or one of VB's older data control types.

Instantiate and invoke a COM component (70-175 and 70-176).

- An object variable can be defined with early binding as follows:

  ```
  Dim objApp as Application
  ```

  If necessary, you can define and create the object as follows:

  ```
  Dim objApp as new Application
  ```

  The second method implements early binding, instantiating the variable as soon as you declare it.
- You can implement late binding with statements such as

  ```
  Dim x as Object
  Set x = New Application
  ```

  This method only instantiates the object on the second line. See "Creating an Instance of an Object with the Set Statement."
- The CreateObject statement can be used to create the instance of an object. This is done as follows:

  ```
  Dim objX as Object
  Set objX = CreateObject(ÒMyApp.SpreadSheetÓ)
  ```

- You can declare an object for programming with its events as follows:

  ```
  Private WithEvents xl as Excel.Application
  ```

- You can't use the NEWkeyword in a WithEvents declaration.
- WithEvents declarations must go into the General section of a module, and they cannot be placed in a standard code module.
- Type library references are usually found in type library files, executables, or dynamically linked libraries.
- If a type library reference is absent for an object that you instantiate, the application will not compile.
- The type library will provide the information needed for early binding such as the object types, properties, and methods supported.
- The CreateObject statement requires a ProgID that is made up of the application name and the object name.
- GetObject behaves differently according to the different syntax options you use. If you call it with the format

  ```
  Set X = GetObject(, ÒServer.ClassÓ)
  ```

  you will receive an error if there is no instance of the requested ActiveX Server already running on the machine. If you use the format

  ```
  Set X = GetObject(ÒÓ, ÒServer.ClassÓ)
  ```

  you will always create a reference to a new object.
- You can find an object's preloaded event procedures by using the drop-down boxes in the code window (just as you would for any control or form).

Create callback procedures to enable asynchronous processing between COM components and Visual Basic client applications (70-175 and 70-176).

- To implement callback functionality from a class to its calling code in a standard executable, provide a custom Class event with at least one By Reference parameter.
- A callback object is *manipulated* in the server (by calling a Notify method), but is *instantiated* in the client before being passed to the server.

Implement online user assistance in a distributed application (70-175) or a desktop application (70-176).

- A reference to a Help file for an application can be set through the Project Properties window. From the Project menu, choose <*project name*> Properties. Either type the Help filename in the Help File Name field, or browse for and select the file.
- If the name and/or location of a Help file is not known at design time, a reference to the file can be set at runtime by using the HelpFile property of the App object. After the filename has been set, pressing F1 in the application will display the online help.

- The current standard format for Microsoft Help files is HTML Help , and the extension for an HTML Help file is chm. The HTML Help format replaces the WinHelp format as the standard, and the extension for WinHelp files is HLP.
- Pop-up tips for controls in Visual Basic applications can be implemented by just putting the desired text in the ToolTipText property of controls. For ToolTips to work on ToolBars and TabStrips , you must also set the ShowTips property of these controls to True .
- To add context-sensitive help to an application, you need to know the mapping between topic IDs that will be used in the project and topics in the Help file. The mapping is usually created by the person who writes the Help file. Cooperation between the help author and developer is important to ensure that the correct topic IDs are associated with the proper objects in the application.
- The format for topic files in an HTML Help project is as a Web page (*.HTM).
- The format #deÞne TopicIDName TopicIDValue is appropriate to context ID mapping files for HTML Help topic files as well as to context ID mapping files for PopUp (WhatsThisHelp) topics. The [MAP] section of an HTML Help file contains a list of topic filenames and locations, and the [ALIAS] section contains a mapping between topic filenames and topic ID constant names.
- WhatsThisHelp must be set to True whether you are using ShowWhatsThis , WhatsThisMode , or a WhatsThisButton on your form.
- ToolTips don't come from a Help file. They come from the ToolTipText property of the control for which the tip is intended.
- Help will automatically be invoked with the F1 key when the App.HelpFile property is set and when the Help file is identified on the Project Properties dialog box.
- Visual Basic will check the active control for a HelpContextID first. If one is not found there, the container of the active control will be checked next.
- The purpose of the [ALIAS] section in an HTML Help project file is to map topic ID names to topic filenames.
- The WhatsThisHelpID identifies the help topic that will be used.
- You can implement error codes from a COM component either by setting a method's return value or by raising an error to the client. See "Handling Errors in the Server and the Client."
- You can check App.NonModalAllowed in an ActiveX DLL component's code to see whether the client supports nonmodal forms. See "Managing Forms in an In-Process Server Component."
- In VB5, a server could not have any forms and be marked for Unattended Execution. In VB6, however, this is possible. For more information, see the section titled "Managing Threads in ActiveX Controls and In-Process Components."

Implement error handling for the user interface in distributed/desktop applications (70-175 and 70-176).

- The Err object provides information about runtime errors. Its Description property gives a brief description of the error.

- The Raise method of the Err objects enables you to return error information from a class module. The arguments of the Raise method specify the error number, description, source, and help information.
- When a runtime error occurs, Visual Basic searches up through the calling chain for an error handler. If none is found, a fatal error occurs.
- To enable inline error handling, place the statement

  On Error Resume Next

  in your code. This will prevent VB's runtime error-handling system from taking over when an error occurs and will make you, the programmer, responsible for reacting to any error conditions that are generated.
- Err.Description is a brief description of the error. Err.Number is the Visual Basic number corresponding to the error.
- On Error Goto 0 disables error handling.
- Use of the vbObjectError differentiates Visual Basic errors from user-defined errors.
- The recommended way to generate runtime errors is with the Err.Raise method. The Error statement can also be used, but it is available in Visual Basic 6 only for the purpose of backward compatibility.
- The LastDLLError property of the Err object is only available in 32-bit environments.
- The Err.Source property is used to identify the location of an error. This property is a text property that the programmer of the application can set to indicate the procedure, object, or application where the error occurred.

Use an Active document to present information within a Web Browser (70-175).

- You can use the Hyperlink object in a UserDocument only when the container is Internet aware.
- NavigateTo and GoBack are methods of the Hyperlink object.
- The following HTML code will open an Active document called mydoc.vbd:

  <A HREF=Ómydoc.vbdÓ>Open User Document</A>

Creating and Managing COM Components

Create a COM component that implements business rules or logic. Components include DLLs, ActiveX controls, and Active documents (70-175 and 70-176).

- Automation components can be implemented as either executables or dynamic link libraries (DLLs). See "Overview and Definition of COM Components."
- One of the easiest methods to increasing speed of a COM components is to implement the component as a dynamic link library (DLL). In addition, you can use early binding on the client side to significantly improve function invocation performance.

See "Overview and Definition of COM Components" in Chapter 12, "Creating a COM Component that Implements Business Rules or Logic," and "Late and Early Binding of Object Variables" in Chapter 10, "Instantiating and Invoking a COM Component."

‡ The WithEvents keyword is used to define a variable that supports events. An example of defining a variable that supports events is shown as follows:

```
Private WithEvents m_obj as TextBox
```

See "Declaring with Events."

‡ "In-process" refers to ActiveX DLLs, and "out-of-process" refers to ActiveX EXEs. See "Comparing In-Process and Out-of-Process Server Components."

‡ After you declare an object variable, you can initialize it by setting the variable to a new instance of the class. For more information, see the section titled "Overview and Definition of COM Components."

‡ Each copy of a SingleUse ActiveX EXE program can provide one object. Creating a second object would cause another copy of the EXE to be loaded in memory. For more information, see the section titled "Using SingleUse Instancing for Separate Instances of Every Object."

‡ You must write code for all the methods and properties of an interface that you use in a class with the Implements keyword. Not doing so will lead to a compile-time error. For more information, see the section titled "Using Interfaces to Implement Polymorphism."

‡ The Implements keyword is used to specify that a specific module will provide an implementation of a specific interface. For more information, see the section titled "Creating the Interface Class."

‡ An event procedure for a Class 's custom event appears in a VB application's code window after you use WithEvents to declare an object variable of the Class . See "Handling a Class Event."

‡ An ActiveX component project's Name option as found on the Project tab of the Options dialog box will be the servername when a client instantiates servername.objecttype in the component. See "Steps to Create a COM Component."

‡ An ActiveX control would be the best way to implement rules for data entry, because it naturally provides a user interface. See "Choosing the Right COM Component Type."

‡ An ActiveX component residing on a network server would be the best implementation for business rules in general. See "Implementing Business Rules with COM Components."

‡ The IDispatch interface supports the Invoke , GetIDsOfNames , GetTypeInfo , and GetTypeInfocount methods.

‡ The IUnknown interface supports the AddRef , Release , and QueryInterface methods.

‡ You will cause vtable binding by correctly using the NEWkeyword in code. See "Under-the-Hood Information about COM Components."

Create ActiveX controls (70-175 and 70-176).

‡ Events are raised by controls by using the RaiseEvent statement. The RaiseEvent statement allows a control to fire an event that its container may respond to if something of interest occurs. If the user changes the Text property of an ActiveX control, for example, it may fire a Changed event to notify its container that the property has changed.

- The UserControl provides the developer with three events that help loading or writing properties. These events are the InitProperties, WriteProperties, and ReadProperties. The ReadProperties and WriteProperties events provide a PropertyBag object that is used to read or write property values so that they can become persistent.
- The developer of the ActiveX control can prevent the control from being visible at runtime by using the InvisibleAtRuntime property or by setting the Top or Left properties so that they are outside the visible portion of the screen. The developer should not use the Visible property.
- The PropertyBag object is used to read and write properties from a persistent location provided by the container. The PropertyBag object is only accessible when the ReadProperties or the WriteProperties events are fired.
- To test your control with Internet Explorer, make sure that the Debug tab of the Project, Properties menu dialog box indicates that the control should be loaded automatically, and that it should be loaded in the Web browser.
- The name of the collection that provides the controls being modified by a property page is SelectedControls.
- To enable an ActiveX control to be a data consumer (that is, give it a DataSource and DataField property), you must set the UserControl's DataBindingBehavior property, and you must use the Tools, Procedure Attributes dialog box to specify a property that is bound to data and is associated with the DataField property.
- The UserControl's GetDataMember event fires whenever a data consumer sets its DataSource property to an instance of the ActiveX control.
- Events can be raised by using the RaiseEvent statement.
- The ReadProperties event receives a property bag from the container used to read a property.
- An ActiveX control can read and write its properties from a persistent location. ActiveX controls can be used from any container that supports ActiveX controls, such as Visual Basic, PowerBuilder, or Microsoft Internet Explorer. In addition, ActiveX controls can be visible or invisible at runtime.
- The GetDataMember event is the place where you would initiate a data connection and recordset to provide to data consumers (bound controls). You do so by setting the second parameter to the initialized recordset. The first parameter is a string used to identify members of DataBindings collections. The event only fires when a bound control's DataSource is set to this control.
- To allow other programmers to program your ActiveX Datasource control's Recordset, you can expose it by implementing a property of type Recordset and initializing it in the GetDataMember event procedure. You must set the UserControl's DataSourceBehavior property to vbDataSource to implement a DataSource control, but this does not implement an exposed recordset by itself.
- You should make your ActiveX control the startup project when you want to test it with a container application. When you want to test it with an EXE project, make the EXE project the startup project.

‡ Before switching from an ActiveX control project to the EXE test project, you need to close the ActiveX control's designer. Otherwise, the control will be disabled in the ToolBox when you switch to the EXE project, and any instances of the control that you have already placed in the EXE will be disabled.

Create an Active document (70-175 and 70-176).

‡ Active documents, like automation components, can be implemented either as dynamic linked libraries or executables. Active documents can be viewed inside any container that supports Active documents such as Microsoft Internet Explorer or Microsoft Binder. Active documents can obtain data asynchronously from an URL or a file. The data in an Active document can be saved to a persistent location such as a file.

‡ Depending on the settings of the UserDocument's MinHeight and MinWidth properties, scrollbars that enable the user to navigate around the displayable area may appear.

‡ During a Write operation, the developer uses the PropertyChanged property of the Active document to notify the container that data has been changed in the Active document. At some point, the ActiveX container will fire the WriteProperties event of the Active document. The container supplies a PropertyBag object when the WriteProperties event is fired. This PropertyBag object can be used to write data to a persistent location by using the WriteProperty method.

‡ During a read operation (usually when the Active document is loaded or returned to be a browser application), the ReadProperties event is fired by the Active document container. As with the WriteProperties event, a PropertyBag object is supplied that allows the object to be read from the persistent location by using the ReadProperty method.

‡ You can use the TypeName function call to detect the type of container that currently is hosting the Active document.

‡ You need to know the type of container that is hosting an Active document, because different containers have different object models. In particular, the techniques for navigating between documents differ from one container type to another.

‡ Global variables defined in a module can be effectively used to pass data between documents.

‡ When an Asynchronous Data Request is completed for an Active document, the AsyncReadComplete event is fired to notify the client.

‡ The AsyncRead method starts an Asynchronous Data Request, and a CancelAsyncRead terminates the request.

‡ Active documents can be used with any container that supports Active documents, such as the OLE control or the Microsoft Binder application.

‡ The PropertyChanged method signals whether the Active document data has changed.

‡ A client application can use automation to create an instance of an Active document. The only variables that the client has access to are variables defined as Public variables.

‡ A VBD file is used by a container to open an Active document.

Design and create components that will be used with MTS (70-175).

- In general, coding a client application that calls MTS components need be no different from coding a client application that calls other types of COM objects. There are no special considerations. The only thing that needs to be done before coding begins, is that the client has to be configured to allow the developer to reference the MTS component.
- MTS components must be in the form of an ActiveX DLL.
- MTS uses a Context object to store information about current transactions.
- A transaction is atomic when all operations included in the transaction must execute successfully for the changes to be committed. If for any reason a part of the transaction fails, the whole transaction is rolled back.
- When a component that supports transactions is created, it may be enlisted in a transaction if the client has a transaction in progress. If there is no transaction present, however, objects from the component will be instantiated without a transaction.
- Setting the transaction support option to Requires a transaction will cause your component to always participate in a transaction.
- SetComplete, which is a method of the Context object of any MTS object, will cause a transaction to commit.
- The SetAbort method of the Context object will cause a transaction to be rolled back. Therefore, any changes that the component made on any ODBC data sources within the body of the transaction will automatically be rolled back.

Debug Visual Basic code that uses objects from a COM component (70-175 and 70-176).

- References to ActiveX components, even in the same project group, must be created by using the Project, References menu option.
- You can create as many instances of your ActiveX components as you want to when testing your components with project groups, but you can never include more than one copy of the same project in a project group.

Choose the appropriate threading model for a COM component (70-175 and 70-176).

- Apartment Model Threading is the default threading model for Visual Basic applications.
- If you want a COM server to run unattended and support multiple threads, you must set the Unattended Execution option. Any existing user-interface messages will be logged according to the application's logging options and the operating system.
- The Thread per Object option in a COM component project initiates a thread for each new SingleUse class that a client instantiates. Note that Thread per Object is not available for in-process servers (ActiveX DLLs), but only for out-of-process servers (ActiveX EXEs).
- You can't specify the number of threads for an in-process server.

Create a package by using the MTS Explorer (70-175).

- An MTS package can be named when it is created, or renamed at any time.
- By default, when you import a package from a package file, all of the components that were in the package when it was exported will be imported.
- The Package and Deployment Wizard from VB enables you to create a setup package that will register your component on the MTS machine and will install all necessary support files.
- The easiest way to duplicate a package across multiple MTS systems is to export it to a PAK file, and then import that PAK file on each of the target systems.

Add Components to an MTS package (70-175).

- Dragging and dropping an ActiveX DLL onto a package in the Explorer will cause it to be added to that package. If the component is not registered, it will automatically be registered on the machine that is running MTS.
- New components can be added to an existing package by either dragging and dropping the DLL file for the component onto the package in MTS Explorer, or by using the Component wizard. From within the Component wizard, you can either add components that are already registered or you can browse to DLLs using the Windows Explorer.
- The operating system uses the values from the Identity tab to determine how to apply permissions to any activity performed by an MTS object.
- When identity settings are configured, and a package is configured to use an NT user account, the password for that account is not verified. If the wrong password is entered, a runtime error will occur on any clients that call the component.

Use role-based security to limit use of an MTS package to specific users (70-175).

- For role-based security to take effect, authorization tracking must be enabled. This is done from the Security tab in the Package Properties window.
- Roles are created from the Roles folder, which is a child of the package in MTS Explorer.
- Standard user and group accounts from the local or domain account database can be added to a role.
- Roles are stored at the package level. Any component within the package can use a role from the package to apply security. If separate packages have identical needs in regards to a given role, the role must be created for each package.
- Role-based security can be assigned to component interfaces, which allows more granularity and flexibility.

Compile a project with class modules into a COM component (70-175 and 70-176).

- To implement a read-only property in a class, only define a `Property Get` procedure and omit the corresponding `Property Let` or `Set`. This is useful when the property is not supposed to be set by the object's client.

- Creating a property by using a Property procedure pair (Property Get /Let or Get/Set) allows for error checking, and ensures that the value passed is valid.
- Returning a value from a Property Get procedure is the same as returning a value from a function. Set the name of the procedure to the return value in the body of the procedure.
- When implementing custom properties with Get/Let/Set procedures, you store the value of the property in a Private variable of the Class event.
- To implement a custom method in a COM component, create a new Public subroutine or function in a class module.
- When you declare a class as SingleUse in a COM component, each client gets its own copy of an ActiveX EXE program. Each copy of the ActiveX EXE program uses up memory as the EXE is loaded.
- When you create a class whose Instancing property is set to MultiUse (as opposed to GlobalMultiUse), client applications must declare instances of that class with the syntax:

  ```
  Dim InstanceName As Servername.Classname
  ```

  When you create a class whose Instancing property is GlobalMultiuse, client applications can declare instances of the class with this syntax:

  ```
  InstanceName As ClassName
  ```

- Using a Public variable in the General Declarations section of a class module will define a new property for that class. However, Property procedures are the recommended way to implement properties.
- A COM component can have its Instancing property set to MultiUse if it is an EXE (out-of-process component) or DLL (in-process component). ActiveX DLLs cannot be SingleUse.
- An ActiveX EXE project may be defined as SingleUse, MultiUse, GlobalSingleUse, or GlobalMultiUse.
- The PublicNotCreatable instancing property setting means that clients can see the class, but that those clients must use other classes in the component to access it.
- A GlobalSingleUse or GlobalMultiUse class will be available to clients without the need for object syntax.
- The Instancing property of an in-process server class may be set to GlobalMultiUse, MultiUse, PublicNotCreatable, or Private. SingleUse and GlobalSingleUse can only be set for out-of-process servers.
- The default scope for a property is Public.
- A dependent class must be defined as PublicNotCreatable and is passed to the client via a function provided in the Application.
- You implement a collection's built-in features by writing wrapper procedures in the Collection class.
- When implementing a collection in an application, you would put the statement to declare the Collection object within the dependent Collection class.
- A class's Terminate event will always run when all object variables referring to the class go out of scope. The ENDstatement ends the component abruptly without any opportunity to run events.

- A collection's built-in Item method has a single variant parameter. This parameter can be used either as a traditional Integer-type index number or as a String-type unique key value to identify the specific item.
- To cause a custom Class event to fire, you use the RaiseEvent statement in the class module's code.
- An Interface may be referenced in other classes with the Implements keyword.
- The Friend keyword makes a member available throughout a component project, but not in clients.

Use Visual Component Manager to manage components (70-175 and 70-176).

- You can publish either a component's source code or the compiled component itself in Visual Component Manager.

Register and unregister a COM component (70-175 and 70-176).

- REGSVR32.EXE is the name of the utility used to register COM components on a machine's system Registry.
- To register an out-of-process COM server (an ActiveX EXE), you can run it standalone or run it with the /REGSERVER option.

Creating Data Services

Access and manipulate a data source by using ADO and the ADO Data Control (70-175 and 70-176).

- A Recordset always requires an open cursor.
- ADO Connection and Recordset objects are the only two ADO objects that support events.
- The ADO Data Control's RecordSource property contains the settings for creating a Recordset . The Recordset property actually exposes the Recordset . (The Recordset property is not available at design time.)

Access and manipulate data by using the Execute Direct model (70-175).

- The Execute Direct data manipulation model is appropriate when you need to perform one-time-only operations on the data or you need to execute queries typed by users.
- The Execute Direct data-access model would be most appropriate in situations where you want to run a query just once that will not be run again.
- You can implement the Execute Direct model with an argument to the Connection object's Execute method, an argument to the Recordset object's Open method, an argument to the Command object's CommandText property, or an argument to the Commandobject's Execute method.

‡ The Execute Direct model would be more appropriate when you don't plan to repeat the same query twice during the same session of the program (the Execute Direct model is more efficient for a single execution, but Prepare/Execute is more efficient for subsequent executions after the first one).

Access and manipulate data by using the Prepare/Execute model (70-175).

‡ The Prepare/Execute model would be more appropriate when you need to execute the same dynamic query several times during a single session of your application.

‡ The Prepare/Execute data manipulation model is appropriate when you need to execute the same dynamic query several times during a single session of your application.

‡ You can implement the Prepare/Execute model only with a Command object, because you prepare the data statement by setting the Command object's Prepared property to True .

Access and manipulate data by using the Stored Procedures model (70-175).

‡ The Stored Procedures model would be more appropriate when you need to execute the same query over many sessions and from many different workstations.

‡ The Stored Procedures data manipulation model is appropriate when you need to execute the same query over many sessions and from many different workstations.

‡ You can implement the Stored Procedures model with an argument to the Connection object's Execute method, an argument to the Recordset object's Open method, an argument to the Command object's CommandText property, or an argument to the Command object's Execute method.

‡ Stored procedures take up fewer workstation resources than inline SQL statements, provide persistent data-manipulation models, help to encapsulate business rules, and perform faster.

‡ The following SQL Server statement will correctly create a stored procedure with two parameters and a return value:

```
CREATE PROCEDURE Find_Result AS
      Int @MyParm1 Output,
      Int @MyParm2
@MyParm1 = Select LastName from Employee
➥Where
                EmployeeID = @MyParm2
If ISNULL @MyParm1
      Return 0
Else
      Return 1
GO
```

‡ After you have executed a SQL Server stored procedure that implements a return value, you can check the value that the stored procedure returned by checking element 0 of the Parameters collection.

Retrieve and manipulate data by using different cursor locations. Cursor locations include client-side and server-side (70-175).

- A client-side cursor can be better than a server-side cursor for smaller rowsets, can be more scalable than a server-side cursor as users are added to the system, and is the only option for persistent Recordset objects. A server-side cursor provides better visibility of other users' changes.
- A server-side cursor does not support the AbsolutePosition, Bookmark, and RecordCount properties of the Recordset.

Retrieve and manipulate data by using different cursor types. Cursor types include Forward-Only, Static, Dynamic, and Keyset (70-175).

- The Static cursor type doesn't make other users' updates visible, and it doesn't make other users' deletions or inserts visible. A Static cursor allows user updates on server data, however.
- A Keyset cursor gives visibility of other users' edits to existing records; allows movement in any direction through the Recordset; and enables the user to update, add, and delete records in the underlying database. However, a Keyset cursor does not give visibility of other users' additions or deletions of records. Only a Dynamic cursor does this.

Use the ADO Errors collection to handle database errors (70-175).

- The ADO Errors collection contains the last errors generated by an ADO action. It does not clear before every ADO action, but rather when a different error has occurred or when you call the Clear method of the Errors collection.

Manage database transactions to ensure data consistency and recoverability (70-175).

- A nested transaction is one that occurs completely within another transaction.
- Calling a rollback on a transaction will cancel all transactions nested within the current transaction.
- A transaction will be committed when there is an update without an explicitly defined transaction. A rollback will occur if the application terminates in the middle of an explicitly defined transaction.

Write SQL statements that retrieve and modify data (70-175).

- The SELECT keyword begins a SQL statement to retrieve records.
- A SQL statement to delete all records from the employees table would read as follows:

```
DELETE FROM employees
```

‡ A SQL statement to insert a new record might read as follows:

```
INSERT INTO employees
(LastName, FirstName)
VALUES (ÒRomeroÓ, ÒJose AntonioÓ)
```

Write SQL statements that use joins to combine data from multiple tables (70-175).

‡ A SQL statement to display matching records between a table named Customer and a table named Orders might read like this:

```
SELECT * FROM Orders
        INNER JOIN Customers
        ON Orders.CustID = Customer.CustID
```

‡ An *outer join* shows all records from one of the two tables, and only matching records from the other. A *right join* is an outer join that shows all records from the second named table, and only matching records from the first named table. A *left outer join* shows all records from the first named table, and only matching records from the second table.

‡ The SQL statement

```
SELECT * FROM customers LEFT JOIN orders
ON customers.custid = orders.custid
```

will display all customers, regardless of whether they have any orders.

‡ A SQL statement that uses the JOIN clause to match records from two tables, showing all records from one of the tables regardless of whether they have matches in the other table might be called an *equijoin*.

Use appropriate locking strategies to ensure data integrity. Locking strategies include Read-Only, Pessimistic, Optimistic, and Batch Optimistic (70-175).

‡ The most resource-efficient combination of cursor-type and cursor-locking strategy is the so-called *firehose cursor*, which is a forward-only, read-only cursor.

‡ Pessimistic locking strategies typically lock a record early on in the retrieve-edit-save cycle; optimistic locking strategies, on the other hand, wait till the last possible moment to lock the record (the moment the record's changes are saved).

‡ An optimistic locking strategy locks data when the Update method is called.

‡ VB ADO's default locking strategy is adLockReadOnly .

TESTING THE SOLUTION

Given a scenario, select the appropriate compiler options (70-175 and 70-176).

‡ Microsoft's implementation of pseudocode in VB partially compiles your program code, which still must be interpreted by runtime DLLs as it executes.

- Native code is generally faster than pseudocode.
- You need an external debugger, such as that supplied with Microsoft C++, to use the symbolic debug information.
- Instead of fully compiling your application before testing it, compile-on-demand compiles code on an as-needed basis during testing.
- With compile-on-demand, an application need not be fully compiled to test it.
- Basic compiler optimization choices include optimizing for fast code, optimizing for small code, and no optimization. You also may choose to favor the Pentium Pro.
- Advanced compiler optimization choices are assume no aliasing, remove array bounds checks, remove integer overflow checks, remove floating-point error checks, allow unrounded floating-point operations, and removing Pentium FDIV checks.

Control an application by using conditional compilation (70-175 and 70-176).

- A value for a compiler option set on the command line overrides the value set in code.

Set Watch expressions during program execution (70-175).

- The Watch on Demand feature isn't available in the Watch window; it appears like a ToolTip when the mouse pointer lingers over an expression.
- Deactivating a breakpoint in code doesn't resume program execution.
- Assertions are not compiled into an executable program; they are only available in the debug environment.
- In Break mode, a program is temporarily suspended during execution so that the programmer can inspect the program state.

Monitor the values of expressions and variables by using the Immediate window (70-175 and 70-176).

- A program must be in Break mode to use the Immediate window.
- Either `?` or `Print` is shorthand for `Debug.Print` when entered into the Immediate window.
- The `Debug` object's `Print` method displays values in the Immediate window.
- Procedures can be executed by typing them into the Immediate window.
- Arrays and user-defined types appear in Locals/Watch windows with a boxed plus sign to the left of their name. Their data elements can be selectively displayed or hidden by toggling the boxed symbol between "+" and "–".
- A boxed plus sign indicates that a variable contains sub-elements not currently displayed.

Implement project groups to support the development and debugging process (70-175).

- Any type of project can be a startup project, with the sole exception of ActiveX controls. ActiveX controls always need another project to host them before they can be executed.
- Visual Basic's error-trapping settings apply to all projects.
- In particular, the Break on All Errors setting is a Visual Basic environment setting that affects all projects.
- When running in the Visual Basic environment, you do not need to compile any of your ActiveX projects before you can run them. The calling project has to be the startup project, not the ActiveX DLL itself. When working with ActiveX controls, you do not have to do anything (except place the control on your form), but ActiveX DLLs and EXEs need to be referenced with the Project, References dialog box.
- Reference and compile information are all stored with each individual project and stay intact when the project(s) is opened outside of the project group.
- ActiveX controls are executed when they are displayed, both in Design and Execution modes. For more information, see the section titled "Using Project Groups to Debug ActiveX Controls."
- The design time testing of an ActiveX control can be done in a project group, among other options. In VB5, project groups were the only method you could use, but this is no longer true in VB6. See "Using Project Groups to Debug ActiveX Controls."
- You don't need to do anything to create a temporary Registry entry for an ActiveX DLL component that you are testing—VB automatically takes care of that for you.
- References to ActiveX controls in the same project group are created automatically for all projects in the group.
- References to ActiveX components, even in the same project group, must be created by using the Project, References menu option.
- You can create as many instances of your ActiveX components as you want to when testing your components with project groups, but you can never include more than one copy of the same project in a project group.

Given a scenario, define the scope of a Watch variable (70-175).

- A Watch may be set at three different levels: the procedure level, module level, or global level.
- The greater the scope of a Watch, the slower it may be calculated. A Watch set at the procedure level executes more quickly than a Watch set at the global level.
- Performance considerations may require you to narrow the scope of a Watch so that it can calculate more quickly. If you don't need to observe a variable in certain contexts, you may want to exclude it from certain contexts in the Watch.
- The Context group of controls on the Watches dialog box determines the scope of the watch.

- The Context group enables you to select from among the modules and procedures in the current project. Global scope is specified by selecting All Modules and All Procedures in the Module and Procedure combo boxes.
- Global variables are visible to watches of all scope levels.
- Module-level variables are visible to watches set at either the module level or procedure level.
- A Public form variable is essentially a property of the form, making it globally accessible throughout the program.

DEPLOYING AN APPLICATION

Use the Package and Deployment Wizard to create a setup program that installs a distributed/desktop application, registers the COM components, and allows for uninstall (70-175 and 70-176).

- A dependency file specifies the supporting files that a particular file needs to be successfully installed on a system.
- You customize the behavior of a standard setup routine by modifying the standard VB setup project, the SETUP.LST file, or dependency files.
- A VB6 setup package furnishes a copy of ST6UNST.EXE to the host system. The VB setup package creates a log file, ST6UNST.LOG, in the application directory. When a user uses the Windows Add/Remove utility to remove the application, ST6UNST.EXE runs, using the information in the log file to remove files and Registry entries.
- The standard compression format for Package and Deployment Wizard files is the CAB (cabinet) format.
- You can use the Manage Scripts icon in Package and Deployment Wizard to manipulate Package and Deployment Wizard's default behavior when you run it in the future.
- The name of the VB6 dependency file is VB6DEP.INI.
- To change the behavior of the custom setup routine as it runs, you can modify the VB source code for SETUP1.EXE (SETUP1.VBP).
- The application removal utility will fail if the user attempts to remove an application that has been installed twice in separate directories.

Register a component that implements DCOM (70-175).

- You can create the necessary remote support (VBR) files for a project that uses DCOM by compiling the project with the Remote Server Files option checked in the Components tab of the Project, Properties dialog box. This will create the necessary files in the same folder as the project, and Package and Deployment Wizard will distribute these files as necessary.

‡ To implement DCOM in an application, you should mark the Remote Server Files option on the Compile tab of the Project, Properties dialog box before you compile the project.

Configure DCOM on a client computer and on a server computer (70-175).

‡ To enable a client or a server to use DCOM, create a client application or component, check the Remote Server Files option on the Project, Properties Component tab, compile the project (this will create a VBR file), create a setup package for the project, and install it on the client or the server, whichever is appropriate.

Plan and implement floppy disk–based deployment or compact disc-based deployment for a distributed/desktop application (70-175 and 70-176).

‡ To deploy to disks, you need to specify multiple CABs when you package the setup. Each CAB will go on a different disk. If you specify only one CAB, the CAB may be too big to fit on the disk.

‡ Specify multiple CAB files to distribute a standard EXE application to users on disks. You can specify the size of the CAB files and then copy the CAB files that the Package and Deployment Wizard creates to the disks.

Plan and implement Web-based deployment for a distributed/desktop application (70-175 and 70-176).

‡ The best way to distribute a changed component to local installations with a browser-driven install is to update the CAB file containing the component on the network server and ask all users to rerun the setup.

Plan and implement network-based deployment for a distributed/desktop application (70-175 and 70-176).

‡ Specify a folder during deployment to network/CD. Specify the drive for the disk deployment.

Maintaining and Supporting an Application

Implement load balancing (70-175).

‡ Availability, extensibility, and performance considerations affect decisions about whether to make load balancing static or dynamic.

Fix errors and take measures to prevent future errors (70-175 and 70-176).

- The Break On All Errors option will stop execution in the IDE every time an error occurs, regardless of any error handling in place. This option is set on the General tab of the environment's Options dialog box.
- Because most ActiveX components consist of class modules, you usually want to use the Break In Class Modules option in the event that an error does occur during testing.

Deploy application updates for distributed/desktop applications (70-175 and 70-176).

- You can rerun the Packaging script in Package and Deployment wizard and then put the changed components in the network distribution location, on floppy disks, or on the Web page used for distribution.

Study and Exam Prep Tips

This chapter provides you with some general guidelines for preparing for the exam. It is organized into three sections. The first section addresses your pre-exam preparation activities and covers general study tips. This is followed by an extended look at the Microsoft Certification exams including a number of specific tips that apply to the Microsoft exam formats. Finally, changes in Microsoft's testing policies and how they might affect you are discussed.

To better understand the nature of preparation for the test, it is important to understand learning as a process. You probably are aware of how you best learn new material. You may find that outlining works best for you, or you may need to "see" things as a visual learner. Whatever your learning style, test preparation takes place over time. Obviously, you can't start studying for these exams the night before you take them; it is very important to understand that learning is a developmental process. Understanding it as a process helps you focus on what you know and what you have yet to learn.

Thinking about how you learn should help you recognize that learning takes place when we are able to match new information to old. You have some previous experience with computers and networking, and now you are preparing for this certification exam. Using this book, software, and supplementary materials will not just add incrementally to what you know; as you study you actually change the organization of your knowledge as you integrate this new information into your existing knowledge base. This will lead you to a more comprehensive understanding of the tasks and concepts outlined in the objectives and of computing in general. Again, this happens as a repetitive process rather than a singular event. Keep this model of learning in mind as you prepare for the exam, and you will make better decisions concerning what to study and how much more studying you need to do.

Study Tips

There are many ways to approach studying just as there are many different types of material to study. However, the tips that follow should work well for the type of material covered on the certification exams.

Study Strategies

Although individuals vary in the ways they learn information, some basic principles of learning apply to everyone. You should adopt some study strategies that take advantage of these principles. One of these principles is that learning can be broken into various depths. Recognition (of terms, for example) exemplifies a more surface level of learning in which you rely on a prompt of some sort to elicit recall. Comprehension or understanding (of the concepts behind the terms, for example) represents a deeper level of learning. The ability to analyze a concept and apply your understanding of it in a new way represents a further depth of learning.

Your learning strategy should enable you to know the material at a level or two deeper than mere recognition. This will help you do well on the exams. You will know the material so thoroughly that you can easily handle the recognition-level types of questions used in multiple-choice testing. You will also be able to apply your knowledge to solve new problems.

Macro and Micro Study Strategies

One strategy that can lead to this deeper learning includes preparing an outline that covers all the objectives and subobjectives for the particular exam you are working on. You should delve a bit further into the material and include a level or two of detail beyond the stated objectives and subobjectives for the exam. Then expand the outline by coming up with a statement of definition or a summary for each point in the outline.

An outline provides two approaches to studying. First, you can study the outline by focusing on the organization of the material. Work your way through the points and sub-points of your outline with the goal of learning how they relate to one another. For example, be sure you understand how each of the main objective areas is similar to and different from another. Then do the same thing with the subobjectives; be sure you know which subobjectives pertain to each objective area and how they relate to one another.

Next, you can work through the outline, focusing on learning the details. Memorize and understand terms and their definitions, facts, rules and strategies, advantages and disadvantages, and so on. In this pass through the outline, attempt to learn detail rather than the big picture (the organizational information that you worked on in the first pass through the outline).

Research has shown that attempting to assimilate both types of information at the same time seems to interfere with the overall learning process. Separate your studying into these two approaches and you will perform better on the exam.

Active Study Strategies

The process of writing down and defining objectives, subobjectives, terms, facts, and definitions promotes a more active learning strategy than merely reading the material. In human information-processing terms, writing forces you to engage in more active encoding of the information. Simply reading over it exemplifies more passive processing.

Next, determine whether you can apply the information you have learned by attempting to create examples and scenarios on your own. Think about how or where you could apply the concepts you are learning. Again, write down this information to process the facts and concepts in a more active fashion.

The hands-on nature of the Step by Step tutorials and the Exercises at the ends of the chapters provide further active learning opportunities that will reinforce concepts as well.

Common-Sense Strategies

Finally, you should also follow common-sense practices when studying. Study when you are alert, reduce or eliminate distractions, take breaks when you become fatigued, and so on.

Pre-Testing Yourself

Pre-testing enables you to assess how well you are learning. One of the most important aspects of learning is what has been called "meta-learning." Meta-learning has to do with realizing when you know something well or when you need to study some more. In other words, you recognize how well or how poorly you have learned the material you are studying.

For most people, this can be difficult to assess objectively on their own. Practice tests are useful in that they reveal more objectively what you have learned and what you have not learned. You should use this information to guide review and further studying. Developmental learning takes place as you cycle through studying, assessing how well you have learned, reviewing, and assessing again until you feel you are ready to take the exam.

You may have noticed the practice exams included in this book. Use it as part of the learning process. The TestPrep software on the CD-ROM also provides a variety of ways to test yourself before you take the actual exam. By using the Practice Exams, you can take an entire practice test. By using the Study Cards, you can take an entire practice exam, or you might choose to focus on a particular objective area, such as Planning, Troubleshooting, or Monitoring and Optimization. By using the Flash Cards, you can test your knowledge at a level beyond that of recognition; you must come up with the answers in your own words. The Flash Cards also enable you to test your knowledge of particular objective areas.

You should set a goal for your pre-testing. A reasonable goal would be to score consistently in the 90-percent range.

See Appendix C, "What's on the CD-ROM," for a more detailed explanation of the test engine.

Exam Prep Tips

Having mastered the subject matter, the final preparatory step is to understand how the exam will be presented. Make no mistake, a Microsoft Certified Professional (MCP) exam will challenge both your knowledge and test taking skills. This section starts with the basics of exam design, reviews a new type of exam format, and concludes with hints targeted to each of the exam formats.

The MCP Exam

Every MCP exam is released in one of two basic formats. What's being called exam format here is really little more than a combination of the overall exam structure and the presentation method for exam questions.

Each exam format uses the same types of questions. These types or styles of questions include multiple-rating (or scenario-based) questions, traditional multiple-choice questions, and simulation-based questions. It's important to understand the types of questions you will be asked and the actions required to properly answer them.

Understanding the exam formats is key to good preparation because the format determines the number of questions presented, the difficulty of those questions, and the amount of time allowed to complete the exam.

Exam Format

There are two basic formats for the MCP exams: the traditional fixed-form exam and the adaptive form. As its name implies, the fixed-form exam presents a fixed set of questions during the exam session. The adaptive form, however, uses only a subset of questions drawn from a larger pool during any given exam session.

Fixed-Form

A fixed-form computerized exam is based on a fixed set of exam questions. The individual questions are presented in random order during a test session. If you take the same exam more than once you won't necessarily see the exact same questions. This is because two or three final forms are typically assembled for every fixed-form exam Microsoft releases. These are usually labeled Forms A, B, and C.

The final forms of a fixed-form exam are identical in terms of content coverage, number of questions, and allotted time, but the questions are different. You may notice, however, that some of the same questions appear on, or rather are shared among, different final forms. When questions are shared among multiple final forms of an exam, the percentage of sharing is generally small. Many final forms share no questions, but some older exams may have a 10 percent to 15 percent duplication of exam questions on the final exam forms.

Fixed-form exams also have a fixed time limit in which you must complete the exam. The TestPrep software on the CD-ROM that accompanies this book carries fixed-form exams.

Finally, the score you achieve on a fixed-form exam, which is always reported for MCP exams on a scale of 0 to 1,000, is based on the number of questions you answer correctly. The exam's passing score is the same for all final forms of a given fixed-form exam.

The typical format for the fixed-form exam is as follows:

- ‡ 50–60 questions.
- ‡ 75–90 minute testing time.
- ‡ Question review is allowed, including the opportunity to change your answers.

Adaptive Form

An adaptive-form exam has the same appearance as a fixed-form exam, but its questions differ in quantity and process of selection. Although the statistics of adaptive testing are fairly complex, the process is concerned with determining your level of skill or ability with the exam subject matter. This ability assessment begins by presenting questions of varying levels of difficulty and ascertaining at what difficulty level you can reliably answer them. Finally, the ability assessment determines if that ability level is above or below the level required to pass that exam.

Examinees at different levels of ability will see quite different sets of questions. Examinees who demonstrate little expertise with the subject matter will continue to be presented with relatively easy questions. Examinees who demonstrate a high level of expertise will be presented progressively more difficult questions. Individuals of both levels of expertise may answer the same number of questions correctly, but because the higher-expertise examinee can correctly answer more difficult questions, he or she will receive a higher score and is more likely to pass the exam.

The typical design for the adaptive form exam is as follows:

- ‡ 20–25 questions.
- ‡ 90 minute testing time, although this is likely to be reduced to 45–60 minutes in the near future.
- ‡ Question review is not allowed, providing no opportunity to change your answers.

The Adaptive-Exam Process

Your first adaptive exam will be unlike any other testing experience you have had. In fact, many examinees have difficulty accepting the adaptive testing process because they feel that they were not provided the opportunity to adequately demonstrate their full expertise.

You can take consolation in the fact that adaptive exams are painstakingly put together after months of data gathering and analysis and are just as valid as a fixed-form exam. The rigor introduced through the adaptive testing methodology means that there is nothing arbitrary about what you'll see. It is also a more efficient means of testing, requiring less time to conduct and complete.

As you can see from Figure 1, there are a number of statistical measures that drive the adaptive examination process. The most immediately relevant to you is the ability estimate. Accompanying this test statistic are the standard error of measurement, the item characteristic curve, and the test information curve.

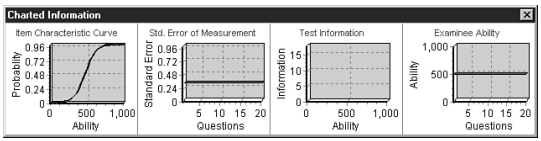


FIGURE 1▲
Microsoft's Adaptive Testing Demonstration Program.

The standard error, which is the key factor in determining when an adaptive exam will terminate, reflects the degree of error in the exam ability estimate. The item characteristic curve reflects the probability of a correct response relative to examinee ability. Finally, the test information statistic provides a measure of the information contained in the set of questions the examinee has answered, again relative to the ability level of the individual examinee.

When you begin an adaptive exam, the standard error has already been assigned a target value it must drop below for the exam to conclude. This target value reflects a particular level of statistical confidence in the process. The examinee ability is initially set to the mean possible exam score (500 for MCP exams).

As the adaptive exam progresses, questions of varying difficulty are presented. Based on your pattern of responses to these questions, the ability estimate is recalculated. Simultaneously, the standard error estimate is refined from its first estimated value of one toward the target value. When the standard error reaches its target value, the exam terminates. Thus, the more consistently you answer questions of the same degree of difficulty, the more quickly the standard error estimate drops, and the fewer questions you will end up seeing during the exam session. This situation is depicted in Figure 2.

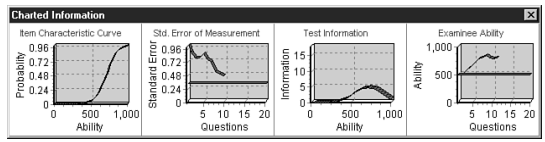


FIGURE 2▲
The changing statistics in an adaptive exam.

As you might suspect, one good piece of advice for taking an adaptive exam is to treat every exam question as if it is the most important. The adaptive scoring algorithm attempts to discover a pattern of responses that reflects some level of proficiency with the subject matter. Incorrect responses almost guarantee that additional questions must be answered (unless, of course, you get every question wrong). This is because the scoring algorithm must adjust to information that is not consistent with the emerging pattern.

New Question Types

A variety of question types can appear on MCP exams. Examples of multiple-choice questions and scenario-based questions appear throughout this book and the TestPrep software. Simulation-based questions are new to the MCP exam series.

Simulation Questions

Simulation-based questions reproduce the look and feel of key Microsoft product features for the purpose of testing. The simulation software used in MCP exams has been designed to look and act, as much as possible, just like the actual product. Consequently, answering simulation questions in a MCP exam entails completing one or more tasks just as if you were using the product itself.

The format of a typical Microsoft simulation question consists of a brief scenario or problem statement along with one or more tasks that must be completed to solve the problem. An example of a simulation question for MCP exams is shown in the following section.

A Typical Simulation Question

It sounds obvious, but your first step when you encounter a simulation is to carefully read the question (see Figure 3). Do not go straight to the simulation application! You must assess the problem being presented and identify the conditions that make up the problem scenario. Note the tasks that must be performed or outcomes that must be achieved to answer the question and review any instructions on how to proceed.

The next step is to launch the simulator by using the button provided. After clicking the Show Simulation button, you will see a feature of the product, as shown in the dialog box in Figure 4. The simulation application will partially cover the question text on many test center machines. Feel free to reposition the simulation or move between the question text screen and the simulation by using hotkeys, point-and-click navigation, or

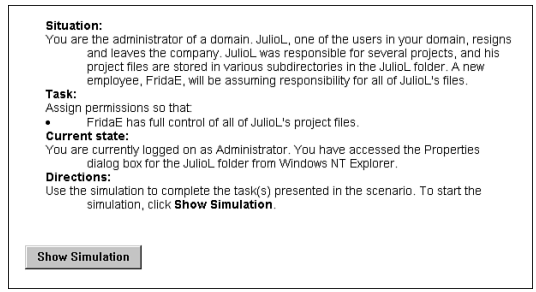


FIGURE 3▲
Typical MCP exam simulation question with directions.

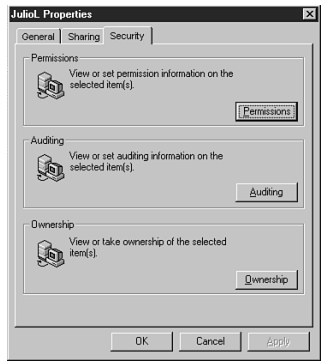


FIGURE 4▲
Launching the simulation application.

even clicking the simulation launch button again.

It is important to understand that your answer to the simulation question will not be recorded until you move on to the next exam question. This gives you the added capability to close and reopen the simulation application (using the launch button) on the same question without losing any partial answer you may have made.

The third step is to use the simulator as you would the actual product to solve the problem or perform the defined tasks. Again, the simulation software is designed to function, within reason, just as the product does. But don't expect the simulation to reproduce product behavior perfectly. Most importantly, do not allow yourself to

become flustered if the simulation does not look or act exactly like the product.

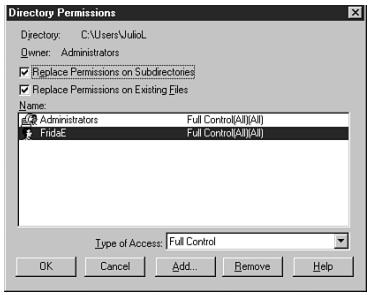


FIGURE 5
The solution to the simulation example.

Figure 5 shows the solution to the simulation example problem.

There are two final points that will help you tackle simulation questions. First, respond only to what is being asked in the question; do not solve problems that you are not asked to solve. Second, accept what is being asked of you. You may not entirely agree with conditions in the problem statement, the quality of the desired solution, or the sufficiency of defined tasks to adequately solve the problem. Always remember that you are being tested on your ability to solve the problem as it is presented.

The solution to the simulation problem shown in Figure 5 perfectly illustrates both of those points. As you'll recall from the question scenario (refer to Figure 3), you were asked to assign appropriate permissions to a new user, Frida E. You were not instructed to make any other changes in permissions. Thus, if you had modified or removed the Administrator's permissions, this item would have been scored wrong on a MCP exam.

Putting It All Together

Given all these different pieces of information, the task now is to assemble a set of tips that will help you successfully tackle the different types of MCP exams.

More Pre-Exam Preparation Tips

Generic exam-preparation advice is always useful. Tips include the following:

- Become familiar with the product. Hands-on experience is one of the keys to success on any MCP exam. Review the exercises and the Step by Steps in the book.
- Review the current exam-preparation guide on the Microsoft MCP Web site. The documentation Microsoft makes available over the Web identifies the skills every exam is intended to test.
- Memorize foundational technical detail, but remember that MCP exams are generally heavy on problem solving and application of knowledge rather than just questions that require only rote memorization.
- Take any of the available practice tests. We recommend the one included in this book and the ones you can create using the TestPrep software on the CD-ROM. Although these are fixed-form exams, they provide preparation that is just as valuable for taking an adaptive exam. Because of the nature of adaptive testing, these practice exams cannot be done in the adaptive form. However, fixed-form exams use the same types of questions as adaptive exams and are the most effective way to prepare for either type. As a supplement to the material bound with this book, try the free practice tests available on the Microsoft MCP Web site.
- Look on the Microsoft MCP Web site for samples and demonstration items. These tend to be

particularly valuable for one significant reason: They help you become familiar with any new testing technologies before you encounter them on a MCP exam.

During the Exam Session

The following generic exam-taking advice you've heard for years applies when taking a MCP exam:

- Take a deep breath and try to relax when you first sit down for your exam session. It is very important to control the pressure you may (naturally) feel when taking exams.
- You will be provided scratch paper. Take a moment to write down any factual information and technical detail that you committed to short-term memory.
- Carefully read all information and instruction screens. These displays have been put together to give you information relevant to the exam you are taking.
- Accept the Non-Disclosure Agreement and preliminary survey as part of the examination process. Complete them accurately and quickly move on.
- Read the exam questions carefully. Reread each question to identify all relevant detail.
- Tackle the questions in the order they are presented. Skipping around won't build your confidence; the clock is always counting down.
- Don't rush, but also don't linger on difficult questions. The questions vary in degree of difficulty. Don't let yourself be flustered by a particularly difficult or verbose question.

Fixed-Form Exams

Building from this basic preparation and test-taking advice, you also need to consider the challenges presented by the different exam designs. Because a fixed-form exam is composed of a fixed, finite set of questions, add these tips to your strategy for taking a fixed-form exam:

- Note the time allotted and the number of questions appearing on the exam you are taking. Make a rough calculation of how many minutes you can spend on each question and use this to pace yourself through the exam.
- Take advantage of the fact that you can return to and review skipped or previously answered questions. Record the questions you can't answer confidently, noting the relative difficulty of each question, on the scratch paper provided. Once you've made it to the end of the exam, return to the more difficult questions.
- If there is session time remaining once you have completed all questions (and if you aren't too fatigued!), review your answers. Pay particular attention to questions that seem to have a lot of detail or that require graphics.
- As for changing your answers, the general rule of thumb here is *don't*! If you read the question carefully and completely and you felt like you knew the right answer, you probably did. Don't second-guess yourself. If, as you check your answers, one clearly stands out as incorrectly marked, however, of course you should change it in that instance. If you are at all unsure, go with your first impression.

Adaptive Exams

If you are planning to take an adaptive exam, keep these additional tips in mind:

- Read and answer every question with great care. When reading a question, identify every relevant detail, requirement, or task that must be per-

formed and double-check your answer to be sure you have addressed every one of them.

- If you cannot answer a question, use the process of elimination to reduce the set of potential answers, then take your best guess. Stupid mistakes invariably mean additional questions will be presented.
- Forget about reviewing questions and changing your answers. Once you leave a question, whether you've answered it or not, you cannot return to it. Do not skip any questions either; once you do, it's counted as incorrect.

Simulation Questions

You may encounter simulation questions on either the fixed-form or adaptive-form exam. If you do, keep these tips in mind:

- Avoid changing any simulation settings that don't pertain directly to the problem solution. Solve the problem you are being asked to solve and nothing more.
- Assume default settings when related information has not been provided. If something has not been mentioned or defined, it is a non-critical detail that does not factor into the correct solution.
- Be sure your entries are syntactically correct, paying particular attention to your spelling. Enter relevant information just as the product would require it.
- Close all simulation application windows after completing the simulation tasks. The testing system software is designed to trap errors that could result when using the simulation application, but trust yourself over the testing software.
- If simulations are part of a fixed-form exam, you can return to skipped or previously answered questions and change your answer. However, if you choose to change your answer to a simulation question or even attempt to review the settings you've made in the simulation application, your previous response to that simulation question will be deleted. If simulations are part of an adaptive exam, you cannot return to previous questions.

Final Considerations

Finally, there are a number of changes in the MCP program that will impact how frequently you can repeat an exam and what you will see when you do.

- Microsoft has instituted a new exam retake policy. This new rule is "two and two, then one and two." That is, you can attempt any exam twice with no restrictions on the time between attempts. But after the second attempt, you must wait two weeks before you can attempt that exam again. After that, you will be required to wait two weeks between subsequent attempts. Plan to pass the exam in two attempts or plan to increase your time horizon for receiving a MCP credential.
- New questions are being seeded into the MCP exams. After performance data is gathered on new questions, the examiners will replace older questions on all exam forms. This means that the questions appearing on exams will be regularly changing.
- Many of the current MCP exams will be republished in adaptive form in the coming months. Prepare yourself for this significant change in testing as it is entirely likely that this will become the

preferred MCP exam format.

These changes mean that the brute-force strategies for passing MCP exams may soon completely lose their viability. So if you don't pass an exam on the first or second attempt, it is entirely possible that the exam's form will change significantly the next time you take it. It could be updated to adaptive form from fixed form

Practice Exams

This portion of the Final Review section consists of two exams of 61 questions each. These practice exams are representative of what you should expect on the actual exams. The answers are at the end of each exam. It is strongly suggested that when you take this exam, you treat it just as you would the actual exam at the test center. Time yourself, read carefully, and answer all the questions as best you can.

Some of the questions are vague and require deduction on your part to come up with the best answer from the possibilities given. Many of them are verbose, requiring you to read a lot before you come to an actual question. These are skills you should acquire before attempting the actual exam. Run through the test, and if you score less than 750 (missing more than 15), try rereading the chapters containing information where you were weak (use the index to find keywords to point you to the appropriate locations).

Exam 1: Developing Distributed Applications (70-175)

1. One of the strengths of ActiveX Data Objects (ADO) is its support for asynchronous operations. Some of these operations issue events before they begin and/or after they complete. To test this, you decide to create and populate a recordset object from a connection object method. Which ADO event will indicate that the recordset has been populated?

   A. CommitTransComplete

   B. ConnectComplete

   C. ExecuteComplete

   D. FetchComplete

2. You have declared an object variable to be of type ADODB.Recordset , but when you pull down the object combo box in the code window, you don't find your variable listed. Why is it not listed?

   A. ADODB.Recordset objects don't support events.

   B. You didn't declare the variable Private .

   C. You didn't declare the variable Public .

   D. You didn't use the WithEvents keyword.

3. Review the following code. The line numbers are for your reference only.

```
1. Private Sub Command1_Click()
2.     Dim oSomeObject As New Object
3.     oSomeObject.SomeMethod
4. End Sub
```

   When will oSomeObject actually be created?

   A. oSomeObject won't get created.

   B. When it is declared on line 2.

   C. When it is first referenced on line 3.

   D. When the event procedure has finished.

4. Assuming that the component has been properly registered, which of the following code examples will always create a new instance of MyClass ? Select all that apply.

   A. Set MyObject = New MyClass

   B. Set MyObject = CreateObject(ÒMyClassÓ)

   C. Set MyObject = GetObject(,ÓMyClassÓ)

   D. Set MyObject = GetObject(ÒMyClassÓ)

   E. Set MyObject = GetObject(ÒÓ,ÓMyClassÓ)

5. You will be designing an application for a state government agency. The agency is involved with issuing occupational licenses to qualified applicants. In your analysis, you determine that a license will contain such information as applicant ID, expiration date, and fee amount. Which object attribute would you use to support these items?

   A. Method

   B. Event

   C. Property

   D. Trigger

6. In building your class module, you've been asked to make a property write-only. How can you specify a write-only property for a class?

   A. Define a Property Set procedure without a Property Let procedure.

   B. Define a Property Let procedure without a Property Set procedure.

   C. Define a Property Let procedure without a Property Get procedure.

D. You cannot make a property write-only in a Visual Basic class module.

7. Which of the following are valid approaches to registering an out-of-process component (.exe file) on a client's system?

   A. Run REGSVR32.EXE.

   B. Run the executable.

   C. Run the executable with a /Regserver command-line argument.

   D. Specify the client's shared folder when compiling the executable.

   E. Use the Package and Deployment Wizard.

8. You and your team have spent hundreds of hours developing an ActiveX control to be used by the medical insurance industry. How can Visual Basic help you protect your investment?

   A. Require a License Key.

   B. Enable the control's safety settings.

   C. Digitally sign the control.

   D. Mark the control's source file read-only just prior to compiling the OCX.

9. In developing an ActiveX DLL, one of the requirements for your class is to expose a public event. Client applications will use this event to display a progress bar during a lengthy, asynchronous task. How would you write the event declaration to allow the user to cancel this process?

   A. Public Event PercentDone (ByVal Percent as Single)

   B. Public Event PercentDone (ByVal Percent as Single) as Boolean

   C. Public Event PercentDone (ByVal Percent as Single, Cancel as Boolean)

   D. Public Event PercentDone (ByVal Percent as Single, ByVal Cancel as Boolean)

10. You've been asked to design an application that will support a user interface as well as expose its classes to potential client applications. Which project template should you select?

    A. ActiveX DLL

    B. ActiveX EXE

    C. ActiveX control

    D. Standard EXE

11. Before delivering your compiled ActiveX document application to the client, you decide to explore the CD-ROM to ensure that it contains the correct files. What two file extensions should you look for?

    A. HTM

    B. VBD

    C. DOB

    D. OCX

    E. DLL

12. Several months ago, you developed a business component that your company uses to validate information collected from a trucking scale. The Visual Basic component contains a single class and was distributed to more than 20 end users who are running a custom Visual FoxPro application.

    Recently, you've been asked to add functionality to your component to support a new line of instruments. What information will you need to provide Visual Basic in order to maintain binary compatibility with the previous component?

    A. The class ID (CLSID) of the original component

B. The interface ID (IID) of the original component

C. The location of the original component

D. The location of the original project

13. During a recent meeting, it was revealed that your DLL component has been crashing several applications. You have been asked to make the component more stable. One idea presented would be to recompile the DLL in such a way that each application would get its own instance of the class, rather than one instance serving all applications, as it is now. How would you enable this in Visual Basic?

A. Convert the project to an ActiveX DLL project.

B. Change the Instancing property to SingleUse .

C. Change the Instancing property to MultiUse .

D. ActiveX DLL components cannot behave in this manner.

14. You've been asked to author an ActiveX control for use by your company. Your ActiveX control contains a constituent label control that exposes its caption property. How could you enable the control's caption property to reflect its name when it is placed on a form at design time?

A.
```
Private Sub UserControl_InitProperties()
    LblCaption.Caption = Me.Name
End Sub
```

B.
```
Private Sub UserControl_InitProperties()
    LblCaption.Caption = UserControl.Name
End Sub
```

C.
```
Private Sub UserControl_InitProperties()
    LblCaption.Caption = Extender.Name
End Sub
```

D.
```
Private Sub UserControl_Initialize ()
    LblCaption.Caption = Me.Name
End Sub
```

15. Your UserDocument contains the following code:

```
Private Sub UserDocument_Initialize()
    MsgBox TypeName(UserDocument.Parent)
End Sub
```

After compiling the ActiveX document, you launch Internet Explorer 3.0 and drag the compiled document into the main window. What will be displayed in the message box?

A. IWEBBROWSER

B. IWEBBROWSERAPP

C. String

D. Nothing will be displayed because an error will result.

16. Your company's COM component will include three classes. The main class will encapsulate insurance policy processing. This class will instantiate and manipulate two private classes to support financial processing. The specifications call for client applications to be able to access some properties and methods from the financial classes but they will never be allowed to instantiate them directly. What strategy should you chose?

A. Make the properties and methods Public .

B. Make the properties and methods Friend .

C. Change the Instancing property to GlobalMultiUse .

D. Change the Instancing property to PublicNotCreateable .

17. Why would you want to change the DLL Base Address of your component? Select the best answer.

A. To avoid collisions

B. To decrease load time

C. To avoid rebasing

D. All of the above

18. Which two properties of a text box control are used to bind to data at runtime?

    A. DataField and DataMember.

    B. DataSource and DataMember.

    C. DataSource and DataField.

    D. You can only bind to data at design time.

19. In reviewing your associate's Visual Basic project, you see that she has chosen to make updates through a cursor opened with a LockType of adLockOptimistic. Without further review, what assumption can you make about the update routine?

    A. This routine should provide adequate consistency.

    B. This routine should provide adequate concurrency.

    C. This routine could allow for disconnected Recordset updates.

    D. This will be a forward-only cursor.

20. Which of these SQL statements will return all Invoice information for the calendar year of 1998?

    A. SELECT * FROM Invoices WHERE InvDate IN
    ➥(Ô1/01/1998Õ,Õ12/31/1998Õ)

    B. SELECT * FROM Invoices WHERE InvDate
    ➥BETWEEN Ô12/31/1997Õ AND Ô1/1/1999Õ

    C. SELECT * FROM Invoices WHERE InvDate
    ➥BETWEEN Ô1/1/1998Õ AND Ô12/31/1998Õ

    D. SELECT * FROM Invoices WHERE InvDate LIKE
    ➥Ô1998Õ

21. Your SQL Server DBA has been complaining that your application is consuming precious resources for every active client in the company. After analyzing the Visual Basic code, you locate the ADO Recordset object that is consuming server resources. Which recordset property should you check first?

    A. CursorLocation

    B. CursorType

    C. ConnectionTimeout

    D. SourceOfData

22. You have been hired as an architect to design a client's distributed database application. You won't be involved with the physical design. During discussions with the client, a business requirement is revealed: Each workstation must be able to browse a list of users currently running the software.

    How should this design be implemented?

    A. Using the Registry

    B. Using a component

    C. Using a table

    D. Using a Recordset

23. A hydrologic plant-monitoring application is being designed by your shop. This will be a distributed application, using Microsoft Transaction Server and Microsoft Message Queue Server. Users will execute queries to retrieve real-time hydrologic data. The users will construct these queries by selecting from various combo boxes on a central form.

    Which cursor type should you use to populate the combo boxes?

    A. ForwardOnly

    B. Dynamic

    C. KeySet

    D. Static

24. Here is a section of source code that you have been asked to optimize:

```
Public Function SoftwareVersion() as String
        SoftwareVersion = App.Major & Ò.Ó &
➥App.Minor & Ò.Ó & App.Revision
End Function
```

How could you make this section of code perform faster?

A. Make it a Sub instead of a Function.

B. Use With and End With statements.

C. Change the function type to Variant.

D. Use the + operators instead of the & operators.

25. You create a new form and immediately place on it a text box, naming it txtLastName. Next, you place another text box, naming it txtFirstName. After running the application a few times, you decide you would like to change the tab order so that the cursor initially appears in txtFirstName.

What is the best way to do this?

A. Add txtFirstName.SetFocus to the Form_Load() event procedure.

B. Set txtFirstName.TabIndex = 0.

C. Set txtFirstName.TabIndex = 1.

D. Set txtLastName.TabStop = False.

26. In order to make inventory changes offline, you've decided to implement a disconnected Recordset. You declare, create, and populate the Recordset correctly. How do you go about disconnecting it?

A. Set rsInventory = Nothing

B. Set rsInventory.Connection = Nothing

C. Set rsInventory.ActiveConnection = Nothing

D. rsInventory.Disconnect

27. Review your associate's SQL statement:

```
SELECT Customers.CompanyName, Orders.OrderDate
FROM Customers LEFT OUTER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
```

When executed via an ADO connection object, what will be the outcome of this statement?

A. All customers will be returned.

B. All orders will be returned.

C. Only customers with orders will be returned.

D. Nothing will be returned because the statement is invalid.

28. You've been asked to deploy the client tier of your distributed application via floppy disks. The compiled executable is small enough to fit on a single disk. What method should you use to deploy your application?

A. Use the Package and Deployment Wizard to create a single CAB file.

B. Use the Package and Deployment Wizard to create multiple CAB files.

C. Copy and distribute the .EXE to the floppy disk.

D. Manually compress and distribute all files required to install the application.

29. Your distributed application has been installed and running for weeks. Recently, improvements have been made to the client tier and it is time to redeploy. You run the Package and Deployment Wizard, regenerating a single CAB file and then email it to your users. After running the setup, your users complain that they do not see any of the improvements. What is the most likely reason?

A. You forgot to include the Visual Basic runtime library.

B. You should have generated multiple CAB files.

C. You forgot to maintain binary compatibility.

D. You forgot to increment the project's version.

30. A distributed application is in its final stage of implementation by your department. Because all computers involved with this application are running 32-bit operating systems, the developers have chosen DCOM to communicate with the Microsoft Transaction Server components. Your responsibility is to deploy the client tier. How will you configure DCOM on each user workstation?

A. Use the Package and Deployment Wizard.

B. Use DCOMCNFG.

C. Use REGEDIT.

D. Have a developer select the Remote Server Files on the Components tab of Project Properties.

31. You've been asked to add programmatic security to a client application. The DLL was created in Visual Basic and deployed to Microsoft Transaction Server. An associate has started writing the code:

```
Dim objContext As ObjectContext
Set objContext = GetObjectContext()
If Not objContext Is Nothing Then
    Ô Missing code goes here
End if
```

Specifications require that only users in the Marketing role be allowed to call a specific method. Select the appropriate section of code to insert in the preceding code:

A.
```
If objContext.IsCallerInRole(ÒMarketingÓ)
➥Then
    If objContext.IsSecurityEnabled Then
    Ô Call method
    End If
End If
```

B.
```
If objContext.IsSecurityEnabled Then
    If objContext.IsCallerInRole
➥(ÒMarketingÓ) Then
    Ô Call method
    End If
End If
```

C.
```
If objContext.IsCallerInRole Then
    If objContext.IsSecurityEnabled
➥(ÒMarketingÓ) Then
    Ô Call method
    End If
End If
```

D.
```
If objContext.IsSecurityEnabled
➥(ÒMarketingÓ) Then
    If objContext.IsCallerInRole Then
    Ô Call method
    End If
End If
```

32. At the initial meeting about your corporation's enterprise-wide application project, you've decided to explore the merits of Windows DNA (Distributed interNet Applications) architecture. Which element of Windows DNA provides for automatic management of object instances, processes, and threads?

A. MSMQ

B. DCOM

C. Windows NT

D. MTS

33. The client has requested a flowchart that shows the flow of data from the database to the client application through the middle-tier. Which of the following demonstrates proper component-based architecture?

A. Database > Business Service Object > Data Service Object > Client

B. Database > Data Service Object > Client

C. Database > Business Service Object > Client

D. Database > Data Service Object > Business Service Object > Client

34. During initial brainstorming sessions, your development team debates whether to design the corporate purchasing system as a two-tier (client/server) or a three-tier (Windows DNA) application. You decide to list the strengths and weaknesses for both architectures on the whiteboard. Which strengths could be listed for three-tier architecture? Select all that apply.

A. Less overall programming

B. Reusability

C. Scalable

D. Load balancing

E. Security

F. Low developer learning curve

35. Your next project will be to build a Visual Basic component to be deployed to Microsoft Transaction Server. This component will use ActiveX Data Objects to communicate with Microsoft SQL Server 7.0 Efficiency and scalability are essential requirements. What guidelines should you follow? Select all that apply.

A. Compile an out-of-process component.

B. Compile an in-process component.

C. Compile a type library.

D. Make the component self-registering.

E. Make the component single-threaded.

F. Manage state.

36. The MTS administrator is about to implement role-based security. Initially, there will be one role (custodian) and six users performing custodian duties. How should the administrator implement this?

A. Create an NT group and add it to an MTS role.

B. Create an MTS role and add it to an NT group.

C. Create the NT users and add them to an MTS role.

D. Create an MTS role and add it to the NT users.

37. Microsoft Transaction Server has been installed and running for only a few weeks. Security has never been enabled or configured, although several Visual Basic DLLs have been deployed successfully. Now that your distributed application has gone into production, how can you ensure that you are the only person in the company to have administrative access to MTS?

A. Remove all users IDs from the administrator role of the system package.

B. Map your user ID to the administrator role of the system package.

C. Remove everyone from the administrator role of the system package.

D. Remove guests from the administrator role of the system package.

38. Your Visual Basic DLL has been thoroughly tested and is ready to be deployed on Microsoft Transaction Server. You create a new package with default Identity settings and then you successfully install the DLL into this package. When a client application calls the component, what identity will the component assume?

A. No identity

B. The identity of the user currently running the client application

C. Your identity, since you installed the component

D. The identity of the user currently logged on to the Windows NT server account

39. An order-processing component is being built by your team. You have been assigned the task of creating the class that will post changes to the inventory tables in the database. It is important that these database updates happen automatically and inside their own transaction. Keep in mind that other classes already involved in their own transactions may be instancing your class. How should you enforce these requirements?

A. Compile your class into its own component.

B. Set your MTSTransactionMode property to RequiresTransaction.

C. Set the Instancing property of your class module.

D. Set your MTSTransactionMode property to RequiresNewTransaction.

40. Corporate headquarters has created a Visual Basic out-of-process component to assist each district office with calculating quarterly earnings. This will merge nicely with a custom application that you are currently building. In order to achieve the fastest binding possible, you would like to use vtable binding. How can you ensure vtable binding to the component?

A. Declare your variables as explicit class types.

B. Declare your variables as Object.

C. vtable binding is not supported by out-of-process components.

D. You'll need access to the component's source code to achieve this.

41. In debugging a section of your manager's Visual Basic application, you see these statements:

```
#If Win32 Then
Declare Sub MessageBeep Lib ÒUser32.dllÓ
➥(ByVal N As Long)
#Else
Declare Sub MessageBeep Lib ÒUser.dllÓ
➥(ByVal N As Integer)
#End If
```

What is the behavior of this structure?

A. At compile time, Visual Basic will determine which DLL to call.

B. At runtime, the Win32 constant will determine which DLL to call.

C. At runtime, the Win32 variable will determine which DLL to call.

D. At runtime, the Win32 command-line argument will determine which DLL to call.

42. In designing your company's order entry system, you've decided to use the text box control for most of the data input. One of the validation rules is to require all purchasing codes to be entered in uppercase. Which two attributes of the text box control would you use to enforce this rule?

A. KeyPress property

B. KeyDown event

C. KeyPress event

D. UpperCase function

E. UCase function

F. KeyAscii argument

43. Transactional processing will become an important feature of your database application. Your team has chosen Visual Basic, ActiveX Data Objects, and SQL Server to implement its design.

To ensure a proper atomic transaction, in which order should you call the respective ADO objects and methods?

A. Begin Transaction, Open Connection, Execute Command, Commit Transaction

B. Begin Transaction, Open Connection, Commit Transaction, Execute Command

C. Open Connection, Begin Transaction, Commit Transaction, Execute Command

D. Open Connection, Begin Transaction, Execute Command, Commit Transaction

44. You've been asked to debug a small DHTML application. You open the Visual Basic project and launch the DHTML designer. The DHTML page contains an input button and an input text control. The original developer had intended for the current date and time to appear in the text box when the input button is clicked. She claims to have written the appropriate code, yet the current date and time do not appear. What is the most likely cause for this bug?

A. She used an HTML input button instead of a DHTML input button.

B. She removed or changed the input button's Name property.

C. She removed or changed the input button's ID property.

D. She forgot to use WithEvents when declaring the input button object.

45. A colleague has created a simple DHTML application that allows potential car buyers to calculate their monthly car loan payments. Users will enter preliminary information about their budget, the loan amount, and interest rate. Then, by manipulating two input buttons, the user can increase or decrease the number of months allowed to pay off the loan. The calculated monthly payment then shows at the bottom of the page. The DHTML page was deployed and has been working fine for weeks.

You have recently been asked to modify the DHTML page and add a feature that will make the monthly payment appear in red if it is higher than the budget allowance. Here is the section of code you started:

```
Private Sub ComputeMonthlyPayment()
    txtPay.Value = Pmt(txtRate.Value / 12,
➥txtMonths.Value, (txtLoan.Value * Ð1))
    If txtPay.Value > txtBudget.Value Then
        Ô  Missing code goes here
    End If
End Sub
```

Select the appropriate line to insert in the preceding code:

A. txtPay.Color = ÒRedÓ

B. txtPay.Color = Red

C. txtPay.Style.Color = ÒRedÓ

D. txtPay.Style.Color = Red

46. Your IIS application has run into some difficulties. Your HTML forms have been submitting data via the GET method and this has caused problems triggering the corresponding events. After some research, you realize that you should be using the POST method rather than the GET method. Which Active Server Page object and corresponding collection should you use to retrieve the values of those form elements passed using the POST method?

A. Request object, QueryString collection

B. Request object, Form collection

C. Response object, Post collection

D. Response object, Form collection

47. The data entry form that you are designing will have some very strict field-level validation to ensure that accounting and other business codes are correct as you tab from field to field. Which text box event should you use to enforce this validation?

    A. KeyPress event

    B. Change event

    C. LostFocus event

    D. Validate event

48. You decide to update your Visual Basic 5 application to take advantage of the new validation features found in Visual Basic 6. Editing your main data entry form, you place code in all of the appropriate validation events. You also change the Cancel button's Cancel property to True, but you leave the CausesValidation property to its default value.

    Which behavior is exhibited when you click the Cancel button while the form is editing data?

    A. The Cancel button's Click event fires only when the control with focus has invalid data.

    B. The Cancel button's Click event fires only when the control with focus has valid data.

    C. The Cancel button's Click event always fires.

    D. The Cancel button's Click event never fires. Only pressing the Esc key will fire the event.

49. In order to enable a form's context-sensitive help, which two properties must be set?

    A. App.HelpFile

    B. Me.HelpFile

    C. Me.Help

    D. Me.HelpContextID

    E. Me.WhatsThisHelp

    F. Me.WhatsThisMode

50. You have purchased a middle-tier component to facilitate communication between client applications and a digital phone system. The author of the component has published several hundred different errors that could be raised by his component. These errors can be trapped and handled by your Visual Basic application using the On Error statement. What properties of Visual Basic's Err object might you want to query in order to determine the error that was raised?

    A. Number, Text, Content

    B. ID, Text, Source

    C. Number, Description, Source

    D. ID, Description, Source

51. Please review the following event procedure:

```
Private Sub cmdShell_Click()
On Error Resume Next
    Shell txtProgram1
If Err.Number <> 0 Then
MsgBox txtProgram1 & Ò is an invalid
➥programÓ, vbCritical, ÒProgram 1 ErrorÓ
End If
Shell txtProgram2
If Err.Number <> 0 Then
MsgBox txtProgram2 & Ò is an invalid
➥programÓ, vbCritical, ÒProgram 2 ErrorÓ
End If
End Sub
```

    Assuming that txtProgram1 contains an invalid program name and txtProgram2 contains a valid program name, what message box(es) will be displayed to the user?

    A. Only the first message box will be displayed.

    B. Only the second message box will be displayed.

    C. No message boxes will be displayed.

    D. Both message boxes will be displayed.

52. What does the following Event procedure accomplish?

```
Private Sub cmdList_Click()
Dim frmTest As Form
Dim ctrTest As Control
For Each frmTest In Forms
For Each ctrTest In frmTest.Controls
MsgBox frmTest.Name & Ò : Ò & ctrTest.Name
Next
Next
End Sub
```

A. All controls on all forms in the project are enumerated and displayed.

B. All controls on loaded forms in the project are enumerated and displayed.

C. Only visible controls on all forms in the project are enumerated and displayed.

D. Only visible controls on loaded forms in the project are enumerated and displayed.

53. Your consulting firm has recently purchased the source code for a middle-tier component. This component has been deployed and running at several of your client sites. In analyzing the Visual Basic source code, you have found that the original developers have checked the box marked Unattended Execution. Based on that setting alone, which of the following statements are true?

A. This component could be implemented as an ActiveX DLL.

B. This component could be implemented as an ActiveX EXE.

C. This component could be apartment-threaded.

D. This component could be multi-threaded.

E. This component could be single-threaded.

F. This component will not contain forms.

G. All of the above.

54. In debating the merits of using one type of ADO cursor over another, which of the following arguments can be made?

A. `Keyset` cursors cannot see data values that other users update.

B. `Dynamic` cursors cannot see data values that other users delete.

C. `Dynamic` cursors incur less overhead than `Keyset` cursors.

D. `Keyset` cursors cannot see data values that other users insert.

55. During the initial discussions with the client, your goal is to define the architecture that will be used to implement its enterprise-wide software. Everyone at the meeting has decided on a three-tier, distributed-application architecture. You would like to know which communication mechanism the client applications will use to communicate with the NT servers. The two choices available to your team are Remote Automation (RA) and Distributed Component Object Model (DCOM). Which of the following questions would best assist you in determining your answer?

A. Do you plan to use Microsoft Transaction Server?

B. Do you plan to use Visual Basic to build the components?

C. What operating system are the clients running?

D. Which network protocols are being used?

56. The users of your distributed application run a process each afternoon that takes several minutes to complete. To provide feedback, you added a "percent complete" message to the form's status bar. Occasionally, this percentage will climb over 100 and confuse your users. Several different classes in your component affect the percentage.

Which of the following provides the best way to find the code causing the overflow?

A. Debug.Assert

B. Debug.Print

C. Immediate window

D. Watch window

E. Call stack

57. A client has asked you to build a browser-based application to be used on its corporate intranet. The corporate standard browser is Internet Explorer. There is a chance that your application will be made available to the public next year. Which Visual Basic project type should you begin?

A. IIS application

B. DHTML application

C. ActiveX document

D. ActiveX control

58. A corporate timesheet program has been implemented as a hybrid intranet application, meaning that it combines active content with native HTML. All components are created using Visual Basic 6 and all users are running Internet Explorer 4. The main page contains an ActiveX control that displays the current number of users logged into the application. To learn more about the control, you view the source and find this OBJECT tag:

```
<OBJECT
CLASSID=Óclsid:dcf0768D-bc7c-101c-b67a-
➥0000c0c3ed5fÓ
CODEBASE=Óhttp://201.202.203.204/controls/
➥userscontrol.cab#version=Ð1,Ð1,Ð1,Ð1Ó
ID=MyControl>
<PARAM NAME=ÓDownloadÓ VALUE=ÓNeverÓ>
</OBJECT>
```

In looking at this script tag, how often will the component be downloaded to the client's browser?

A. Only when the client's computer has an older version

B. Only when the client's computer has a newer version

C. Never

D. Always

59. A colleague has created a stored procedure in SQL Server. The stored procedure does not return row data but does expect several parameters, some of which are used to return information to the client. You've been asked to write the Visual Basic code to interface with that stored procedure. Which ActiveX Data Object must you use?

A. Connection

B. Recordset

C. Command

D. Query

60. One of the features of the Component Object Model is the ability of an object to support multiple interfaces. Furthermore, object classes can be polymorphic, meaning that many classes can provide the same property or method, and a client doesn't have to know what class an object belongs to before calling the property or method. What two features in Visual Basic are used to support polymorphism?

A. Implements statement

B. WithEvents statement

C. CreateObject function

D. CreateInstance function

E. Abstract classes

61. In debugging your manager's Visual Basic component, you have created a watch expression to monitor the value of her ADO RecordCount property. For much of the program's execution you see <Object variable or With block variable not set> under the value column in the watch window. When a value does appear, it shows as a -1. What is the most likely cause of this negative value?

    A. The Watch expressions' procedure context is wrong.

    B. The Watch expressions' module context is wrong.

    C. It's a result of the cursor type selected.

    D. The Recordset doesn't contain any records.

Answers to Exam Questions

1. C. The ExecuteComplete event will be issued when the connection object's execute method has completed. FetchComplete is a Recordset event and will not be issued in this case.

2. D. An object that raises events is called an *event source*. To handle the events raised by an event source, you can declare a variable of the object's class using the WithEvents keyword.

3. A. When using the New keyword during declaration, it can't be used to declare variables of any intrinsic data type, such as Object.

4. A, B, E You can create new instances by using the New keyword in the Set statement or by calling CreateObject. GetObject has many uses and accepts two parameters: pathname and class. If pathname is a zero-length string ("""), GetObject returns a new object instance of the specified type. If the pathname argument is omitted, GetObject returns a currently active object of the specified type. If no object of the specified type exists, an error occurs.

5. C. A property is a named attribute of an object. Properties define object characteristics, such as size and name, or the state of an object, such as enabled or disabled.

6. C. There are many possible combinations of Property procedures. All of them are valid, but some are relatively uncommon, like write-only properties (only a Property Let, no Property Get).

7. B, C, E ActiveX executables can be registered by compiling the component, running the component, running the component with a /Regserver command-line argument, or by creating a setup program. Regsvr32.exe is a utility to register DLLs and OCX.

8. A. This will enable licensing for ActiveX control projects. A Visual Basic license file (*.VBL) will be created when you build the project. The *.VBL must be registered on the user's machine for the components to be used in the IDE.

9. C. When the client application receives a PercentDone event, the Percent argument will contain the percentage of the task that's complete and the Cancel argument will allow the client to set it to True to cancel the task. Because the Cancel argument was not declared with the ByVal keyword, Visual Basic uses its default behavior, which is to pass by reference.

10. B. ActiveX executables are an out-of-process component that can double as a standalone desktop application. A desktop application that provides objects should test App.StartMode, and show its main form only if it was started standalone.

11. B, E. An ActiveX document can be built as an out-of-process component (an .EXE file) or an in-process component (a .DLL file). In either case, when you compile the project, in addition to creating the .EXE or .DLL file, Visual Basic creates a Visual Basic Document file, which has the extension vbd.

12. C. To maintain compatibility between two versions of a component, Visual Basic needs you to provide the path to a previously compiled version of the component in the Version Compatibility box on the Component tab of the Project Properties dialog box.

13. D. The idea cannot be implemented because ActiveX DLL classes cannot be instanced as SingleUse. ActiveX EXE classes, however, can be either SingleUse or MultiUse.

14. C. You can achieve this behavior by evaluating the Name property of the Extender object in your control's InitProperties event procedure. These extender properties are provided by the container on which your control is placed.

15. D. You can't use the Initialize event to test the container because when the Initialize event fires, the document is not yet sited. The appropriate event would be UserDocument_Show().

16. D. The Instancing property of the two private classes should be changed to PublicNotCreateable, so that other applications can use objects of this class. Only the component can create the objects. Other applications cannot create objects from these classes.

17. D. By setting a unique DLL Base Address, you can avoid, or at least limit, the chance that your component loads into the same memory location as another component. If such collisions can be avoided, the operating system won't have to rebase, or relocate the component elsewhere in memory, and component load time will be decreased.

18. C. Visual Basic allows you to bind a data consumer to a data source at runtime. This can be done by setting the text box's DataSource property to an ADO Recordset object and the DataField property to a specific in that Recordset.

19. B. With an optimistic locking strategy, concurrency and performance are good because locks are not held on the rows that make up the cursor. Typically, pessimistic locking strategies allow for better consistency. Only a batch optimistic cursor can allow for disconnected updates.

20. C. Using the Between keyword specifies an inclusive range to search.

21. A. By changing the CursorLocation to the client computer, you won't be consuming server resources for every active client. This change, however, may have other impacts.

22. C. Maintaining application information such as this would be best handled by a table. Because this is a database application, one could be easily built and maintained. The Registry would not be accessible by all users and a component should not hold state such as this.

23. A. Because the application is populating only the combo boxes, it does not require scrolling or dynamic capabilities. The forward-only cursor is the best way for retrieving data quickly with the least amount of overhead.

24. B. Using the With statement allows you to perform a series of statements on a specified object, such as the App object, without re-qualifying the name of the object. This type of referencing is faster.

25. B. By default, the first control placed on a form has a TabIndex of 0, the second has a TabIndex of 1, and so on. When you change a control's TabIndex, Visual Basic will automatically renumber the positions of the other controls to accommodate insertions and deletions.

26. C. Setting the Recordset's ActiveConnection property to Nothing disconnects it from the active connection. The connection can then be closed and changes can be made to the client-side Recordset.

27. A. A left-outer join will return all rows from the first-named table, which is the table that appears leftmost in the join clause. Unmatched rows in the rightmost table do not appear.

28. B. If you are deploying your application on floppy disks, you should choose multiple CAB files, specifying a size no larger than the disks you plan to use. Don't forget that you will need to distribute the Visual Basic runtime library.

29. D. Before creating a package, you should ensure that your project's version number has been set on the Make tab of the Project Properties dialog box, especially if distributing a new version of an existing application. Without incrementing the version number, the end user's computer may determine that critical files do not need to be updated.

30. A. The Package and Deployment Wizard will determine whether your project includes remote automation or DCOM features and will display screens that allow you to specify the appropriate options in these cases.

31. B. It's a good idea to call IsSecurityEnabled before calling IsCallerInRole. If security isn't enabled, IsCallerInRole will not return an accurate result. IsCallerInRole requires a single parameter that contains the name of the role in which to determine if the caller is acting.

32. D. Microsoft Transaction Server is a component-based transaction processing server for developing, deploying and managing remote-server applications. MTS provides a runtime infrastructure that will automatically track all instances of objects, no matter how many clients are using the objects. And, depending on the threading model used by the component, MTS will create threads as necessary to ensure that object code executes efficiently.

33. D. Component-based applications provide resources and services through COM-based objects. Data service objects manage and satisfy requests for data generated by business service objects, and are often implemented as stored procedures or COM components. Business service objects are the logical layer between client and data services, and a collection of business rules and functions that generate and operate upon information. They accomplish this through business rules, which can change frequently, and are thus encapsulated into components that are physically separate from the application logic itself.

34. B, C, D, E Distributed applications are component based, and because business rules get encapsulated into components, they can be reused by many different applications. When deployed within an efficient infrastructure, such as the Windows Distributed interNet Architecture (DNA), distributed applications become scalable and can be load balanced. Security is improved because the middle-tier components can be centrally secured using a common, easily-administered infrastructure. Building three-tier solutions can take more overall time than two-tier solutions, since you are having to identify your middle-tier services, design your class modules, and encapsulate all business rules. Because component-based development may be new to some developers, training and experience will need to be obtained.

35. B, F. To use a component with Microsoft Transaction Server, it must be a COM DLL, which is also referred to as an in-process component. When Visual Basic compiles a DLL, it automatically includes a type library and makes it self-registering. For scalability, the component should be apartment-model threaded rather than single-threaded, and the developer should manage state carefully.

36. A. The most flexible approach to mapping users to roles is to create an NT group for each role in the application. After you have assigned the NT groups to all the roles in your MTS application, you can use NT's User Manager to add or remove users in each group.

37. B. When you install MTS, the system package does not have any users mapped to the Administrator role, so security is disabled. With security disabled, any user can use the MTS Explorer to modify package configuration on that computer. If you map users to the system package administrator role, MTS will check when a user attempts to modify packages in the MTS Explorer.

38. D. When objects access resources, such as files and databases, the objects are authenticated before being allowed access. By default, an object's identity is that of the user who is currently logged on to the computer running Microsoft Transaction Server. You should assign a specific Windows NT user account to the package, so that all objects running in that package share that identity.

39. D. By setting the MTSTransactionMode property of a class module, you can specify the transactional behavior of that class. The RequiresNewTransaction setting indicates that the object must execute within their own transactions. When a new object is created, MTS automatically creates a new transaction for the object, regardless of whether its client has a transaction.

40. A. You can ensure that early binding is used by declaring variables of specific class types. It's the component that determines whether DispID or vtable binding is used. Components you author with Visual Basic will always support vtable binding.

41. A. The #If directive allows for conditional compilation. Conditional compilation is typically used to compile the same program for different platforms, as with the previous Win32 example.

42. C, F. You use the KeyPress event whenever you want to process the standard ASCII characters entered in a text box. If you want to force a character to be uppercase, you can use this event and modify the KeyAscii argument.

43. D. With ADO, transactions are handled by the connection object, so it must be created first. Next, you begin the new transaction and execute the command(s). When you are satisfied with the results, you should commit the transaction, which ends the current transaction and saves the changes.

44. C. An ID is an attribute of an HTML tag that provides a unique identifier for each element on a page. Without an ID, there is no way to distinguish one control from another. Some elements are automatically assigned an ID when you add them to a page. For those that are not, you must add the ID yourself by entering a value for the ID property, because only elements with IDs can be programmed.

45. C. DHTML allows us to change the style of any HTML element in a document. You can change colors, typefaces, spacing, indentation, position, and even the visibility of text. Style attributes can be set from the style sub-object for each element.

46. B. The Form collection of the Request object retrieves the value of form elements passed in an HTTP request when the request is passed using the POST method.

47. D. Take advantage of the new validate event in VB 6. It is superior to the LostFocus event in that you can designate some controls on the form to not fire the validate event, such as a Cancel button.

48. B. By default, a command button's CausesValidation property is set to True. Valid data would therefore have to be entered into the current control before the Cancel button's Click event would fire.

49. A, D. After you set the application's HelpFile property, when a user presses the F1 key, Visual Basic calls Help and searches for the topic identified by the current HelpContextID. If the control with focus has a HelpContextID of 0 (the default), Visual Basic will search that control's container, and then it's container, until it reaches the form, looking for a valid HelpContextID to lookup.

50. C. In determining which error has occurred, you will want to query the error number (Err.Number) and maybe the source of the error (Err.Source). If those two properties don't adequately describe the error that occurred, you may want to query the message text (Err.Description).

51. D. With inline error handling such as this, you test for an error immediately after each statement or function call. It then becomes very important to explicit clear the error object, by calling Err.Clear, after handling the error, otherwise the next check of Err.Number will report the same error, even though no new error has occurred.

52. B. The Forms collection represents each loaded form in an application. These can include MDI form, MDI child, and non-MDI forms, but they must be loaded. The controls collection represents all loaded controls on a component, such as a form.

53. G. This setting on the General tab of the Project Properties screen indicates that the component will have no user interaction, user documents, user controls, or forms. It can be set for either ActiveX DLL (in-process) or ActiveX EXE (out-of-process) project types. If the project is an ActiveX DLL, then you can select apartment or single-threaded models. If the project is an ActiveX EXE, you can select a multithreaded model, where each object created from such a class can be on a separate thread of execution.

54. D. Dynamic cursors incur more overhead than Keyset cursors, but are able to detect all changes made to the data values (inserts, updates and deletes). Keyset cursors can see data updates made by other processes, but cannot see inserted data values made by other processes.

55. C. Remote Automation is the choice for 16-bit platforms or for 32-bit platforms that do not implement DCOM. In general, Remote Automation is slower and less stabile than DCOM. Because the remote servers will be NT servers and NT servers support DCOM, the only question is about the client computers.

56. D. By adding Watch expressions to the Watch window, you can direct Visual Basic to put the application into break mode whenever an expression's value changes or becomes true. The application will stop and enter break mode on the line that caused the expression to change or become true.

57. A. Because the chance exists that the public might be accessing your application in the near future, you should opt to make your application browser-independent, which means building an application that runs on a Web server, such as Internet Information Server (IIS). The three other project types will force your users to use a browser that supports Active content or the Dynamic HTML object model.

58. D. If the version number specified in the codebase attribute is Ð1,Ð1,Ð1,Ð1, then Internet Explorer will always try to download the latest version of the desired component. This can be a costly effort involving many network transactions.

59. C. The Command object must be used, because it has support for parameters. A Connection object is not required, because a command can be created independently of a previously defined connection, by passing a valid connection string. Because the stored procedure is not returning row data, a Recordset object is not required.

60. A, E. Abstract classes are used to define an interface. They are not meant for creating objects. Their purpose is to provide the template for an interface that you'll add to other classes via the Implements statement.

61. C. The RecordCount property will reflect how many records are contained in the recordset, if ADO can make that determination. Certain cursor types, such as ForwardOnly, will return a RecordCount of Ð1.

EXAM 2: DEVELOPING DESKTOP APPLICATIONS (70-176)

1. You've been asked to design the user interface to an employee information program that will run on a kiosk computer in your building's lobby. By default, only employees' first and last names, companies, and floor numbers are to be displayed. One of the interface requirements calls for a menu to allow the user to select additional employee information to be displayed. The three menu items are to be "Title," "Telephone Number," and "Email Address." Which Menu Control property should you use to enable the user to toggle their selections?

   A. Selected

   B. Checked

   C. Enabled

   D. Visible

2. The users of your application would like to be able to right-click on a list box and be presented with some specific menu options. Using the Menu Editor, you create a menu named mnuPopup1 and add these items. Next, you write the following event procedure code:

```
Private Sub Form_MouseUp(Button As Integer,
➥Shift As Integer, X As Single, Y As Single)
    If Button = vbRightButton Then
    Form1.PopupMenu mnuPopup1
    End If
End Sub
```

   When you test the program, it does not behave as expected. Why?

   A. Your forgot to set mnuPopup1's visible property to False .

   B. The code should have been placed in the list box's event.

   C. The code should have been placed in the form's MouseDown event.

   D. The code should read Form1.MenuPopup mnuPopup1.

3. Kristen is troubleshooting a Visual Basic program that dynamically adds and removes customer menu items at runtime. In looking at the code, she sees Load and Unload statements. Why are application errors occurring when the program tries to unload some of the customer menu items?

   A. Menu items must have their visible properties set to False before they can be unloaded.

   B. The menu items were created at design time.

   C. The menu items are not part of a control array.

   D. The menu items must have had their Checked property set to True .

4. Which of the following are possible ways in which to add controls to a form?

   A. Double-click on the toolbox icon.

   B. Drag the toolbox icon to the form.

   C. Click the toolbox icon and then draw it on the form.

   D. Right-click the toolbox icon and select Add to Form.

   E. Use the Project—Components screen.

5. How do you set a label's caption to read "This & That"?

   A. Label1.UseMnemonic = True

   B. Label1.UseMnemonic = False

   C. Label1.Caption = ÒThis & ThatÓ

   D. Label1.Caption = ÒThis /& ThatÓ

   E. You cannot display an ampersand in a label caption.

6. Late one evening, you create a new form with a single grid control. You add code to a few of the grid's event procedures and then, realizing that you've forgotten to name the control properly, set the name property to grdEmployee in the Property window. You then save and close your Visual Basic 6 project. The next day, when you review your form module, where do you find your grid's event procedure code?

   A. It has been moved to the newly named control's Click event.

   B. It has been moved to the general declarations section.

   C. It has been moved to the standard module that contains sub main() .

   D. It has been removed from the project.

7. You are performing a peer review of a section of code. Your associate's code should trap the F5 key being pressed while typing comments into a text box. Here is your associate's code:

```
Private Sub txtComments_KeyPress (KeyAscii
➥as Integer)
    If KeyAscii = vbKeyF5 Then
                    Ô Do Something Here
    End If
End Sub
```

   Which of the following statements is true of this code?

   A. This code will run and will satisfy the requisite.

   B. This code will run but does not satisfy the requisite.

   C. This code will not run due to a runtime error.

   D. This code will not compile due to a compiler error.

8. Your client's user interface specifications require items of information to be listed with an accompanying icon. The icons will be provided by the client. Your client also needs the ability to search for a particular item. Which Visual Basic 6 control has this functionality built-in?

   A. ImageCombo

   B. Treeview

   C. Listview

   D. ImageList

9. You are building an application that will display the international "no" symbol (a red circle with a diagonal bar inside it) dynamically over various images, such as those for smoking, parking, and dogs. Which code below will perform the overlay using Visual Basic picturebox and imagelist controls?

   A.
```
Set picNo.Picture = ilsPics.ListImages
➥(ÒSmokingÓ).PictureSet picNo.Overlay =
➥ilsPics.ListImages(ÒNoÓ) .Picture
```

   B.
```
Set picNo.Picture = ilsPics.Overlay
➥(ÒSmokingÓ,ÓNoÓ)
```

   C.
```
picNo.Overlay(ilsPics.ListImages
➥(ÒSmokingÓ) .Picture, ilsPics.ListImages
➥(ÒNoÓ).Picture)
```

   D. This cannot be done with a single picturebox control.

10. Your form contains a single toolbar with three button groups. The first group contains file options: New, Open, Save, and Close. The next group contains clipboard options: Cut, Copy, and Paste. The third group contains two drop-down style buttons: Undo and Redo. The last two groups are separated by a placeholder-style button. How many ButtonClick event procedures will you have available?

A. 1

B. 3

C. 4

D. 9

11. Code maintenance and readability are important in your shop. During an annual review, a section of your code was sent around the office via email for all developers to scrutinize:

```
Private Sub tlbMain_ButtonClick(ByVal Button
➥As MSComctlLib.Button)
    Select Case Button.Index
    Case 1
        Call NewLedgerFile
    Case 2
        Call OpenLedgerFile
    Case 3
        Call CloseLedgerFile
    End Select
End Sub
```

What improvement do you think they suggested?

A. You should not have used the Call statement.

B. You should have used separate Click events for each of these buttons.

C. You should have examined the Button.Key property instead.

D. You should have examined the Button.Caption property instead.

12. Your form's StatusBar must display graphical icons in a few of its panels. Which of the following methods is not a valid approach to perform this action?

A. Place the icons in an ImageList control and assign the images to the panels at runtime.

B. Place the icons in an ImageList control and bind it to the StatusBar at design time.

C. Load the icons into each panel at design time.

D. Use the LoadPicture function to load the icons at runtime

13. A modification request asks you to update the form used to enter customer narrative The form's StatusBar needs to reflect the current state of the Caps Lock key. The form contains three text boxes with their MultiLine properties set to True . Where should you start writing your procedure code?

A. You don't need to write any code.

B. Trap for the vbKeyCapital key in the form's KeyUp event.

C. Trap for the vbKeyCapital key in each control's KeyUp event.

D. Make a Windows API call.

14. You are constructing a data-driven questionnaire that will display from 1–12 check boxes, depending on the question being asked. You add one check box to the form at design time and decide to add any others dynamically at runtime using the Load statement. When you test it, no error is generated, yet nothing happens. What is the most likely cause?

A. You forgot to make the original control part of an array.

B. You forgot to set the original control's index to 0.

C. You forgot to set the original control's index to 1.

D. You forgot to set the visible property.

15. Jerry's form was designed with 10 OptionButton controls, named optAttributes(0) through optAttributes(9) . At runtime, Jerry calls the Load statement and dynamically adds a new OptionButton control. Immediately after the new control is loaded, which of the following statements will always be true? Select all that apply.

A. optAttributes(10).Value is equal to optAttributes(0).Value .

B. optAttributes(10).Value is equal to optAttributes(9).Value .

C. optAttributes(10).TabIndex will always be equal to optAttributes(9).TabIndex + 1 .

D. optAttributes(10).Top will always be equal to optAttributes(9).Top .

E. optAttributes(10).Visible will always be False .

16. You want to enumerate all controls on all Visual Basic forms in your project. Which approach will you need to take in order to do this?

A. Enumerate the Forms collection.

B. Enumerate the Controls collection.

C. Enumerate both the Forms and Controls collections.

D. Load each form and enumerate each form's Controls collection.

17. During the testing phase of your application, your manager would like to have a "panic" button placed on all forms, which will immediately halt execution of the application, regardless of the application's state or what the user is currently doing. You add a command button and implement the following routine to all forms:

```
Private Sub cmdPanic_Click()
    Dim frmCurrent as Form
    For Each frmCurrent in Forms
        Unload frmCurrent
    Next frmCurrent
End Sub
```

Which of the following statements is true of your solution?

A. This code meets your manager's needs and is an ideal solution.

B. This code meets your manager's needs and is an adequate solution.

C. This code fails to meet your manager's needs.

D. This code will cause a runtime error.

18. To disallow users from typing numbers into one of eight text box controls on your form, you should do which of the following?

A. Set the form's KeyPreview property to True .

B. Use the SendKeys statement.

C. Set the value of the KeyAscii argument to 0.

D. Add code to the change event.

19. An insurance form that you are constructing for a client must contain 10 check box controls and 10 corresponding text box controls. As the user checks a box, the corresponding textbox will be enabled for the user to enter text. When the form loads, no boxes will be checked and all of the text boxes will be disabled. Here is the code you have written:

```
Private Sub chkOption_Click(Index As Integer)
    txtOption(Index).Enabled =
➥chkOption(Index)
End Sub
```

Which of the following statements is true of your solution?

A. This code meets your client's needs and is an ideal solution.

B. This solution meets your client's needs and is an adequate solution.

C. This code fails to meet your client's needs.

D. This code will cause a runtime error.

20. You plan to include an HTML document with your Visual Basic application. The document will list other products that your company produces.

Rather than require your users to launch a browser to view the document, you've added a form to the application that contains a Browser control. You want the Browser control to center and size itself appropriately on the form when the form is first displayed, as well as whenever the user changes the form's height or width. How would you do this?

A. Place resizing code in the form's Load event.

B. Place resizing code in the form's Resize event.

C. Place resizing code in the form's Load and Resize events.

D. Place resizing code in the form's Load and Click events.

21. Deron places a timer control on an application's startup form. He then sets its interval property to 5,000, which equates to roughly five seconds. Next, he places a command button on the form and writes the following code:

```
Option Explicit
    Private Sub Timer1_Timer()
    Unload Me
End Sub
```

In addition to the form's Initialize, Load, and Unload events, which events will fire when the application is run? Select all that apply.

A. Form_Paint()

B. Timer1_Initialize()

C. Form_GotFocus()

D. Form_Resize()

E. Form_Terminate()

22. Eric is writing an application for a local grocery store to check out videos. His SDI application will display modeless forms. The customer form displays summary values that are calculated dynamically as videos are checked out. How can Eric ensure that those summary values are refreshed when the user returns to the customer form after having just checked out some videos?

A. Place calculations in the customer Form_Load() event procedure.

B. Place calculations in the customer Form_Paint() event procedure.

C. Place calculations in the customer Form_Activate() event procedure.

D. Use a timer control.

23. With which of the following Form events do you have the ability to prevent a form from unloading? Select all that apply.

A. LostFocus

B. QueryUnload

C. Deactivate

D. Unload

E. Terminate

24. Which of the following scenarios would cause a form's Deactivate event to fire? Select all that apply.

A. When closing the form and returning focus to the previous form

B. When moving the focus to another displayed form

C. When loading and displaying another form

D. When moving the focus to another running application

E. When bringing up Task Manager

F. When closing the application with Task Manager

25. Dave is reviewing the contents of a Visual Basic setup CAB file to determine if the help files were included. Assuming Dave has used a format native to Visual Basic, what file extensions could he be looking for? Select all that apply.

    A. *.HTM

    B. *.HPJ

    C. *.CHM

    D. *.HLP

26. To get her project back on schedule, Kerri decided to have two separate consultants assist in building the application's help system. Each consultant created an HTML help file. Now, Kerri must figure out how to associate two different help files with her application. How must Kerri do this?

    A. Reference the help files at design time.

    B. Reference the help files at runtime.

    C. Assign each form's HelpFile property to the corresponding help file.

    D. It can't be done. The two help files must be merged.

27. In building an Windows Explorer style application, you want to enable a Launch button whenever a user clicks on a file with a recognized extension. You will maintain a list of these recognized extensions in a dynamic array. Which Visual Basic function will you use to activate the file's associated application?

    A. CreateObject

    B. GetObject

    C. Shell

    D. Open

28. Immediately after installing Visual Basic 6, you start a new Standard EXE project. When you browse the Project References screen, what preselected references do you find? Select all that apply.

    A. Visual Basic Type Library

    B. OLE Automation

    C. Visual Basic for Applications

    D. Microsoft ActiveX Data Objects 2.0 Library

    E. Visual Basic objects and procedures

29. While updating a section of code, you find that you need to declare an object variable that can refer to an existing Drawing object. Which expression would you use?

    A. Set X as Drawing

    B. Set X as New Drawing

    C. Dim X as Drawing

    D. Dim X as New Drawing

30. Monty has forwarded a snippet of code to you:

```
Private Sub cmdBrowse_Click()
    Dim oBrowser As Object
    Set oBrowser = New InternetExplorer
    oBrowser.Visable = True
End Sub
```

    You agree that there is a misspelling in the Property name, but Monty also insists that the compiler does not seem to be catching this problem. Why must you agree with him?

    A. Because Visual Basic is using late binding

    B. Because Visual Basic is using early binding

    C. Because the InternetExplorer class doesn't have a type library

    D. Because Visual Basic is using vtable binding

31. Under which circumstances are you *not* allowed to use the WithEvents keyword? Select all that apply.

    A. In the general declaration section of a form module

    B. In the general declaration section of a class module

    C. In the general declaration section of a standard module

    D. With the As New keyword

    E. With an object that doesn't support events

32. Which feature would you use to indicate to an ActiveX control that a property should be persisted to the PropertyBag object?

    A. PropertyChanged method

    B. CanPropertyChange method

    C. WriteProperties event

    D. WriteProperty method

33. Mike is almost finished designing his ActiveX control and would like to begin testing it. He adds a standard EXE project to his project group and opens Form1. Mike finds his control in the toolbox, but is unable to add it to the form. How can Mike resolve this problem?

    A. Mike needs to register his control.

    B. Mike needs to make his standard EXE the startup project.

    C. Mike needs to start a second instance of Visual Basic to properly test the control.

    D. Mike needs to close the Control Designer.

34. Developers using your ActiveX control would like a pop-up color-picker dialog box to assist them when they are setting the CaptionColor property. In order for Visual Basic to do this automatically, you declare your CaptionColor property as OLE_COLORIn testing your control, you drop an instance onto a form and name the control MyControl . What would be displayed when the following code was executed in the test project?

    ```
    MsgBox TypeName(MyControl.CaptionColor)
    ```

    A. OLE_COLOR

    B. Long

    C. Object

    D. Nothing will be displayed because an error will result.

35. Your ActiveX document project contains two separate user documents. You would like to pass data from UserDoc1 to UserDoc2 during navigation. How would you implement this?

    A. By declaring a Public variable in UserDoc1

    B. By declaring a Public variable in UserDoc2

    C. By declaring a Public variable in a standard module

    D. By using the Property Bag object

36. You are currently viewing UserDocumentA inside Internet Explorer. Which feature of an ActiveX Document allows you to display UserDocumentB ?

    A. Hyperlink.GoForward

    B. Hyperlink.NavigateTo

    C. OpenURL

    D. UserDocument_Show

37. Select two appropriate ways to stop your application and enter Break mode, while developing in the Visual Basic IDE.

    A. Debug.Assert True

    B. Debug.Assert False

C. Stop

D. Halt

E. Break

38. Your company is very good about installing the latest version of ActiveX controls on all of its developers' computers. This is performed automatically by Microsoft Systems Management Server. How can you ensure that your Visual Basic projects will always make use of the latest version of these ActiveX controls?

A. Create a new project template that uses these ActiveX controls.

B. Select Upgrade ActiveX Controls in the Project Properties dialog box.

C. Select Upgrade ActiveX Controls in the Tools Options dialog box.

D. Enable binary compatibility on all of your projects.

39. In examining your apprentice programmer's application, you find that he uses many public variables. What concerns you is how he is declaring them. He has declared some with the Public keyword and some with the Dim keyword. Which declaration is not actually visible by the entire application?

A. A variable declared in a form's declarations section as Public

B. A variable declared in a form's declarations section as Dim

C. A variable declared in a standard module's declarations section as Public

D. A variable declared in a standard module's declarations section as Dim

40. Before delivering your compiled ActiveX Document application to the client, you decide to view the CAB file to ensure that it contains the correct files. What two file extensions should you look for?

A. HTM

B. VBD

C. DOB

D. OCX

E. EXE

41. Your UserDocument contains the following code:

```
Private Sub UserDocument_Show()
    StatusBar1.SimpleText =
➥TypeName(UserDocument.Parent)
End Sub
```

When you start this component using Internet Explorer 3.0, what will the preceding code do?

A. Display String in the status bar

B. Display IwebBrowserApp in the status bar

C. Cause an error

D. Nothing

42. You've been hired to author an ActiveX control to be used by your local police department. Your control will contain a constituent label control that exposes its caption through property procedures. You would like for the caption of this label to initially contain the name of the control itself. Following is the code that will assign the label's caption property the name of the control:

```
lblCaption.Caption = Extender.Name
```

Where should you place this line of code to achieve the desired behavior?

A. UserControl_InitProperties event

B. UserControl_Initialize event

C. UserControl_ReadProperties event

D. UserControl_WriteProperties event

43. In designing a component to assist in home financing, you create three class modules. Each class can be instanced by a client application. One of the classes contains a method that should not be able to be called by a client application, but must be able to be called by any instance of any of the three classes. How should you implement this?

A. Make the method Public .

B. Make the method Private .

C. Make the method Protected .

D. Make the method Friend .

E. Only properties can behave in this manner.

44. An associate has created an out-of-process component to assist you in calculations. The component was built in Visual Basic, but no external type library was provided to you. How can you ensure vtable binding to the component?

A. Declare your variables as explicit class types.

B. Declare your variables as Object .

C. vtable binding is not possible without an external type library.

D. You'll need access to the component's source code to achieve this.

45. While reviewing a peer's code, you see the following event procedure:

```
Private Sub cmdSort_Click()
    Dim strTemp As String

    Ô Enter several characters
    strTemp = InputBox(ÒEnter several
➥charactersÓ)

    Ô Sort the string
    strTemp = SortString(strTemp)
    Ô Display the results
    MsgBox strTemp
End Sub
```

Having never heard of the SortString function, you ask her where she obtained it. She claims that it is a method of a class that her project is referencing. How is this possible?

A. The object must have been declared with the New statement.

B. The class must have its instancing property set to GlobalMultiUse .

C. The SortString method is the default method of the class.

D. This cannot be true.

46. Because all users in the Accounting division are running Internet Explorer, you have been contracted by them to write a DHTML application. Using Visual Basic, you begin designing a customer form by placing the following tag atop a new DHTML page:

```
H1 ID=CustStatus>Customer Status</H1>
```

What statement would you use to dynamically change the text to read Customer is Delinquent , with Delinquent in boldface?

A. CustStatus.Text = ÒCustomer is
➥<B>Delinquent</B>Ó

B. CustStatus.innerText = ÒCustomer is <B>
➥Delinquent </B>Ó

C. CustStatus.innerHTML = ÒCustomer is <B>
➥Delinquent </B>Ó

D. This cannot be done dynamically.

47. The next version of your DHTML application will include some minor improvements. One enhancement the users would like is for the customer form to show the status in red if the customer is delinquent. Here is the current tag:

```
<H1 ID=CustStatus> Customer is
➥<B>Delinquent</B></H1>
```

What DHTML statement would cause this to happen?

A. CustStatus.Color = Red

B. CustStatus.Color = ÒRedÓ

C. CustStatus.Style.Color = Red

D. CustStatus.Style.Color = ÒRedÓ

48. Having just compiled an ActiveX control on your computer, you would like to test it from Microsoft Word. Should you use the RegSvr32.exe utility to register the OCX file?

A. No; Visual Basic registered the component for you when you compiled it.

B. No; RegSvr32.exe is for registering DLLs only.

C. Yes; the component has not yet been registered on that computer.

D. Yes; the component has been registered only for Visual Basic's use.

49. Conditional compilation allows your company to maintain one set of source code to be used to generate an engineering application in English, French, and German. In addition to displaying text native to each country, each executable performs calculations using logic local to each country. Here is some pseudocode:

```
#If conGermanVersion then
    Ô Compute using German logic
#ElseIf conFrenchVersion then
    Ô Compute using French logic
#Else
    Ô Compute using English logic
#End If
```

One way to set the conditional compilation constant is in code, such as in the following example:

```
#Const conGermanVersion = 1    Ô Will
➥evaluate true in #If block.
```

What are two other ways to set a compilation constant?

A. Use a conditional compilation control.

B. Set an argument in Tools, Options.

C. Set an argument in Project, Properties.

D. Specify the constant in the Package and Deployment Wizard.

E. Pass a command-line parameter to Visual Basic.

F. Pass a command-line parameter to the application.

50. The data entry form that you are designing will have some very strict field-level validation to ensure that accounting and other business codes are correct as you tab from field to field. Which text box event should you use to enforce this validation?

A. KeyPress

B. Change

C. LostFocus

D. Validate

51. Part of your company's check reconciliation software involves reading a large ASCII text file. This file is provided by your financial institution each month on CD-ROM. You've been asked to improve the interface by adding a ProgressBar control to the form, showing the percentage of the file that has been read. You have set the range properties in code and drafted the following:

```
Ô Open ASCII File
Open ÒD:\CHECKDUMP.TXTÓ For Input As #1

Ô Loop Until End of File, reading and
➥processing lines from the Þle
```

```
Do While Not EOF(1)
    Line Input #1, strLine
    Call ProcessLine(strLine)
    Loop

Ô Update ProgressBar to show 100% completion
```

The ProcessLine procedure will increment the ProgressBar, but to ensure that the bar shows 100% completion after the loop has finished, which line of code should you add to the bottom of this routine?

A. Progressbar1.Value = Progressbar1.Max

B. Progressbar1.Value = 100

C. Progressbar1.Max = True

D. Progressbar1.Value = 0

52. Your customer has requested that its application be delivered on floppy disks, but its computers have only the 5.25" high density disk drives. The compiled executable, however, is small enough to fit on a single disk. How should you use the Package and Deployment Wizard to deploy your application?

A. Use the wizard to create a single CAB file.

B. Use the wizard to create multiple CAB files.

C. Use the wizard to copy the .EXE to a floppy disk.

D. The wizard doesn't support that disk format.

53. With the new Package and Deployment Wizard, which of the following are true statements about Packaging scripts and Deployment scripts? Select all that apply.

A. Packaging scripts are used by Setup to list the files to be set up; Deployment scripts are used by Setup to determine where those files are to be installed.

B. Packaging scripts allow the Package and Deployment Wizard to run in silent mode; Deployment scripts are used by Microsoft System Management Server.

C. Packaging scripts are required for building CAB files; Deployment scripts allow the Package and Deployment Wizard to run in silent mode.

D. Packaging scripts allow the Package and Deployment Wizard to run in silent mode; Deployment scripts allow you to deploy the project again later, using the same settings.

E. Packaging scripts can be used to generate standard or Internet packages; Deployment scripts allow the Package and Deployment Wizard to run in silent mode.

54. What is the importance of the WithEvents keyword when declaring an object variable in a standard module?

A. It allows you to respond to events triggered by that object.

B. It allows you to implement events triggered by that object.

C. It allows you to raise your own events when using that object.

D. WithEvents is not valid in a standard module.

55. Recently, your users have been asking you to add small, pop-up notes that explain the purposes of the controls on the main data entry form. They claim that this type of help is found in many of the popular Microsoft Windows applications and all they have to do is hover their mouse pointer over a control and wait for the help to pop up. How can you implement this behavior in Visual Basic?

A. Set each control's ToolTip property.

B. Set each control's ToolTipText property.

C. Implement WhatsThisHelp .

D. Utilize the form's PopupMenu method.

56. You've been asked to build a customer phone list report. The phone list will be populated from a table named CUSTLIST, found in the SALES98 database on a SQL server named FARGO. In order to keep development time to a minimum, you are asked to use the Data Environment Designer. This report will be the only purpose of this application. After selecting a new Data Project template, what steps should you take to build this report?

    A. Configure the default connection, add a command to the connection, and drag the command to the detail section of the data report.

    B. Configure the default command, add a connection to the command, and drag the connection to the detail section of the data report.

    C. Add a new command, add a connection to the command, and drag the connection to the detail section of the data report.

    D. Add a new connection, add a command to the connection, and drag the command to the detail section of the data report.

57. How do you dynamically bind a text box control to an ADO Recordset object?

    A. Set the DataSource and DataMember properties.

    B. Set the DataSource and DataField properties.

    C. Use the Visual Basic data control.

    D. Use the Microsoft ADO data control.

    E. Use the Data Environment.

58. After evaluating an innovative ActiveX control, your programming shop has decided to purchase a license allowing all developers to use the control. Your manager gives you the path to the shared folder containing the OCX file and instructs you to copy it into your Windows system folder. After it has been copied to your hard drive, how would you publish the component to your local Visual Component Manager database?

    A. Run RegSvr32.exe.

    B. Run PubSvr32.exe.

    C. Drag the OCX into the VCM.

    D. Do nothing. VCM automatically publishes all components in the Windows system folder.

59. How would you go about adding a comment to the component published in your local Visual Component Manager database?

    A. Create a text file and republish the original component, including the new file.

    B. Create a text file and drag it into the VCM onto the component.

    C. Use the Add Annotation dialog box.

    D. Visual Component Manager does not support this.

60. You've designed a class module that contains the following method:

```
Public Sub DivideByZero()
    On Error GoTo ErrorHandler
    MsgBox ÒThe answer is: Ò & CStr(3 / 0)
    Exit Sub
ErrorHandler:
    MsgBox ÒDivision by ZeroÓ
End Sub
```

What happens when a standard EXE test project calls this method with its error-trapping behavior set to Break in Class Modules ?

A. Visual Basic enters break mode in the class module.

B. Visual Basic enters break mode in the test project.

C. A message box will display Division by Zero .

D. Nothing will happen.

61. The users of your database application have been complaining that a label control showing a grand total amount sometimes appears as 12345678.90 . How can you determine what part of your application is causing this peculiar amount to be displayed?

A. Use a Debug.Assert statement.

B. Monitor the Watch window.

C. Break when the value changes.

D. Break when the value is True .

Answers to Exam Questions

1. B. The Checked property is the correct choice. When Checked is True, a small checkmark will show next to the menu item, and vice versa. Neither Enabled nor Visible will work because when the user goes to toggle back, the menu option won't be able to click the option. Selected is a bogus answer.

2. B. Because the list box is on top of the form, you must add code to its MouseUp event rather than the form's. The form's MouseUp event will only fire when it is clicked on directly.

3. B. Only controls created at runtime can be passed to the Unload statement.

4. A, C. Double-clicking the toolbox icon or clicking the toolbox icon and drawing it on the form are the only two available choices that will place controls on a form.

5. B. Setting a label's UseMnemomic property to False displays the ampersand, whereas setting it to True defines the access key.

6. B. Orphaned event procedure code is moved to the general declarations section.

7. B. To trap function keys, you must use the KeyUp or KeyDown event. The KeyPress event procedure is ideal for trapping standard ASCII character keystrokes.

8. C. All of these controls, with the exception of ImageList , can display information and custom graphics, but only the Listview has a FindItem method for searching.

9. B. Use the ImageList 's overlay method.

10. A. Regardless of content, a toolbar has only one ButtonClick event.

11. C. For readability's sake, the Button.Index property doesn't tell us much. The Button.Key is a better property to use because you can use friendly, more descriptive names in your code. Using Button.Caption would be an option, but there is no guarantee that you will have unique captions or captions at all on your toolbar buttons.

12. B. Unlike other controls discussed in Chapter 4, "Creating Data Input Forms and Dialog Boxes," the StatusBar control cannot be bound to an ImageList .

13. A. The StatusBar control can display the current state of the Caps Lock key without writing any event procedure code.

14. D. Controls added dynamically are invisible at first. You must set their Visible property to True for them to appear.

15. A, E. Newly created controls automatically get the same property values as the first control in the array, except for Visible, Index, and TabIndex properties. Visible is always False, Index is one higher than the last control in the array, and TabIndex is one higher than the highest TabIndex property on the form.

16. D. Because the Forms collection contains only those forms that are loaded, there is no guarantee that it will contain all forms in your project. You will need to ensure that all forms are loaded first and then enumerate each form's Controls collection.

17. B. Because you were explicitly asked for an immediate halt and are able to throw caution to the wind, the End statement would be the ideal solution. The given code would be a safer choice, however, because it would cause all loaded forms to fire their QueryUnload and Unload events and any cleanup code placed there.

18. C. To disallow certain keystrokes, such as numbers, you can set the KeyAscii argument to 0 inside the KeyPress event procedure.

19. A. Having two parallel control arrays is the simplest approach to enabling/disabling the corresponding text boxes. The Boolean value of a text box's enabled property can be directly assigned from the value property of the corresponding check box. Value is the default property of a check box.

20. B. You only have to put the code in the Resize event. It will fire initially and whenever the user changes the form's dimensions.

21. A, D, E. The form's firing order will be Initialize, Load, Resize, Paint, and then the timer will fire, causing the form's QueryUnload, Unload, and Terminate events. The timer control doesn't have an initialize event and the form cannot receive focus because the command button has been placed on the form and will initially receive the focus.

22. C. The Activate event is appropriate here; it will fire when the user returns to the customer form from any other form in the application. The Load event fires only once—when the form is first loaded, and the Paint event can fire too sporadically to be useful.

23. B, D. Both the QueryUnload and Unload events have the ability to set their Cancel argument to True to prevent the form from unloading. The QueryUnload is the preferred event to use in that it can also determine the reason the form is closing.

24. B, C. DeActivate fires when focus is moved from one form to another within the same application. Closing a form does not cause the DeActivate event to occur.

25. C, D. Visual Basic can read WinHelp or HTML help files. The file extensions are HLP and CHM, respectively.

26. B. Only one help file can be referenced at design time, so you will need to assign the App.HelpFile property at runtime whenever appropriate. Forms do not have a HelpFile property.

27. B. GetObject is used to activate the application associated with a particular file.

28. B, C, E. During your practice you had a chance to visit this screen; you will find Visual Basic for Applications, Visual Basic runtime objects and procedures, Visual Basic objects and procedures, and OLE Automation.

29. C. Dim X as Drawing declares a variable that can later be assigned to an existing drawing object. If you had used the Dim X as New Drawing, a new instance would be created. Set is used for assignment, and the syntax is Set Object = [New] Class.

30. A. Because the variable was declared as Object, rather than an explicit type, Visual Basic is forced to use late binding and therefore won't be able to identify the error until runtime.

31. C, D. WithEvents cannot be used in a standard module. You aren't allowed to use the As New keyword. WithEvents is allowed in form and class modules and even for objects that don't support events.

32. A. The user should call the PropertyChanged method, so that the UserControl will fire the WriteProperties event. The property can then be persisted via the WriteProperty method.

33. D. You will need to close the ActiveX Control Designer before you can properly test your control in a test project.

34. B. OLE_COLOR and all color properties and methods are represented by a long integer.

35. C. This behavior can be controlled by using a Public variable defined in a standard module. Because you can't know the sequence in which your ActiveX documents are invoked, you'll need to use a Public variable to pass the document object references between them.

36. B. With the NavigateTo method of the Hyperlink object, you can provide the URL of a Web page or a local document that the container application can open as an Active document.

37. B, C. Debug.Assert suspends execution at the line on which the method appears, provided that the assertion expression evaluates to False. Stop statements suspend execution.

38. B. When you select this option, the project will automatically update ActiveX controls that are newer than those used in the project.

39. B. Variables declared using the Dim statement in a Form module are private, whereas variables declared using a Dim statement in a standard module are public. For this reason, you should always declare your Public variables with the Public statement.

40. B, E. An ActiveX document can be built as an out-of-process component (an .EXE file) or an in-process component (a .DLL file). In either case, when you compile the project, in addition to creating the .EXE or .DLL file, Visual Basic creates a Visual Basic Document file, which has the extension VBD.

41. B. Using the TypeName(UserDocument.Parent) function, you can determine the container of an ActiveX document. The appropriate event for determining the container is the Show event.

42. A. You can achieve this behavior by evaluating the Name property of the Extender object in your control's InitProperties event procedure. The InitProperties event occurs when a new instance of a control is placed on a container, such as a form.

43. D. By declaring a member (property or method) as a Friend, it becomes visible to other objects in your component. Friend members are not part of the class's interface, so they can't be accessed by client programs.

44. A. You can ensure that early binding is used by declaring variables of specific class types. It's the component that determines whether DispID or vtable binding is used. Components you author with Visual Basic will always support vtable binding. Visual Basic compiles type library information into its components.

45. B. When you set a class's Instancing property to GlobalMultiUse and then reference that component from another project, you can use the properties and methods of the class without having to explicitly create an instance of the class. The members of the class are added to the global name space of your project, so they will be recognized just as if they were part of Visual Basic.

46. C. Dynamically changing the HTML document from within the browser is a very powerful feature of DHTML. If you were changing only the text of the CustStatus message, you could have assigned to the innerText property, but since you were changing the text to include HTML tags (boldface on/off), you must assign to the innerHTML property, so that the HTML gets rendered.

47. D. DHTML allows you to change the style of any HTML element in a document. You can change colors, typefaces, spacing, indentation, position, and even the visibility of text. Style attributes can be set from the style sub-object for each element. Internet Explorer recognizes colors by constant names, such as Red or Dark Green.

48. A. When Visual Basic compiles an ActiveX control, it automatically registers it on your system so that any Windows application including VB can use the control. Other users that intend to use your control must register the component on their computers first. Regsvr32.exe will work great for them.

49. C, E. You can set a conditional compilation constant in code, using the #Const statement or by listing them in the Conditional Compilation Arguments field of the Make tab on the Project Properties. You can set a constant on the command line:

```
vb6.exe /make MyProj.vbp /d
➥conGermanVersion=1
```

50. D. Take advantage of the new validate event in VB6. It is superior to the LostFocus event in that you can designate some controls on the form to not fire the validate event, such as a Cancel button.

51. A. Typically, the ProgressBar's minimum property is 0 and its maximum is 100, but that doesn't have to be the case. For example, it might be easier to increment the ProgressBar from 0 to the number of records in a file. Because the question didn't specify what the ranges were, the safe choice would be to assign the Value property the value of the Max property when the loop was complete.

52. B. If you are deploying your application on floppy disks, you should choose multiple CAB files, specifying a size no larger than the disks you plan to use. The current sizes supported are 1.44MB, 2.88MB, 1.2MB, and 720KB. Don't forget that you will need to distribute the Visual Basic runtime library.

53. D, E. Scripts allow you to repackage and/or redeploy your projects at a later time. Either type of script can be selected the next time you run the Package and Deployment Wizard. Scripts can also be passed to the wizard executable (pdcmdln.exe) to enable the wizard to run in silent mode. Scripts enjoy the same functionality that an interactive user would, if he or she were running the wizard manually.

54. D. WithEvents allows you to respond to events triggered by an ActiveX object. The WithEvents keyword is valid only in an Object (Class or Form) module.

55. B. ToolTips are a great way to relay pertinent information to the user as they navigate the user interface. By setting the ToolTipText property of all desired controls, a small label will be displayed when the mouse pointer is held over that control for a set length of time.

56. A. When you start a Data Project, Visual Basic automatically adds the Data Environment and Data Report Designers to your project. Your first step should be to configure the existing connection in the Data Environment so that it connects to the SQL Server and database. Next, you should add a command under the connection and configure it to open the correct table or query. And finally, you can drag the finished command to the Data Report Designer, drop it in the details section of the report, and then format the report as you wish.

57. B. Many intrinsic controls, including the text box, support dynamic binding directly to Recordset objects at runtime. Data controls and the data environment are not needed. Set the control's DataSource property to the Recordset object and the DataField property to the specific field or column to bind the control to. DataMember would further clarify which set of data to use from the DataSource, but doesn't apply when binding to a Recordset object.

58. C. When you drag the OCXfile to the Visual Component Manager, you automatically launch the Publish Wizard, which walks you through the process of publishing a component for reuse. You can also launch the wizard from within the Visual Component Manager.

59. C. Use the Annotations tab of the Component Properties dialog box to enter textual annotations or comments on a component as well as the author's name. These annotations can be searched.

60. C. The Break in Class Modules option applies only to unhandled errors in class modules. Because this method has an error handler in place, no error is raised to Visual Basic, so no break takes place.

61. D. By adding watch expressions to the Watch window, you can direct Visual Basic to put the application into Break mode whenever an expression's value changes or becomes True. In this scenario, you know what the suspicious value is; you can set a Watch expression equal to "12345678.90" and tell Visual Basic to enter Break mode when the value is True.

PART

III

APPENDIXES

A Glossary

B Overview of the Certification Process

C What's On the CD-ROM

D Using the Top Score Software

E Visual Basic Basics

F Suggested Readings and Resources

APPENDIX A

Glossary

A

Access key A key that the user can press (usually in combination with the Alt key) to move focus to a control. Typically, it also fires the control's Click event. Access keys are accompanied by the visual cue of an underlined letter in a control's accompanying caption and are typically only available when the cue is actually visible. Almost all controls that have a Caption property in Visual Basic support access keys, including menus. A TextBox control can also have an access key, by associating it with the caption of a label. Compare with *Shortcut key*.

Active document An application whose interface can be displayed directly in ActiveX container applications. VB programmers can create Active document applications.

Active form The form where the focus currently resides in a running application. Focus is typically on one of the form's controls and not on the form itself. The form itself will only have focus if it currently contains no controls that can receive focus.

ActiveX Microsoft's standard for implementing objects and communicating between them. ActiveX is based on the COM specification and replaces the OLE standard.

ActiveX component Application elements developed using the COM standard. In VB6, you can develop the following ActiveX components: ActiveX EXE, ActiveX DLL, ActiveX control, ActiveX Document DLL, and ActiveX Document EXE.

ActiveX container An application that can host Active documents. Microsoft Internet Explorer and Microsoft Office Binder are ActiveX container applications.

ActiveX control A custom control for VB development provided in an OCX file. Custom controls can come from Microsoft or from third-party developers. VB6 programmers can create ActiveX controls using VB.

ActiveX Data Objects (ADO) A high-level, COM-based data access model for Windows programming that exposes OLE DB functionality in a Windows programming environment. ADO is the preferred data-access model for VB6. See also the definition for *OLE DB*..

Administrator role A default role that exists in the System package. When authorization checking is enabled for the system package, membership in this role determines which NT accounts have administrative access from MTS explorer. A user must be mapped to the Administrator role if he or she has a need to create, delete, and modify MTS packages.

ADO See *ActiveX Data Objects*.

ASP Active Server Pages, a technology that Microsoft provides as a supplement to its Web server, Microsoft Internet Information Server (IIS). ASP provides an enhancement to the HTML language that allows IIS (or any other Web server that can understand ASP) to dynamically generate the HTML that it sends to browsers.

The end product of ASP is still standard HTML, recognizable by all browsers. All ASP processing takes place on the server and so is completely transparent to browsers, which only see the standard HTML pages produced by ASP.

Assertion A device either provided by the programming environment or implemented by the programmer to test the validity of assumptions in a program. In VB, the Debug.Assert method provides built-in assertion capability.

Atomicity A quality of transactions that guarantees that for the transaction to be committed, all actions performed within the body of the transaction must succeed. If any aspect of a transaction fails, all activity associated with the transaction is rolled back.

Automation The part of the COM specification that specifies the rules for manipulating a component through its exposed objects by means of scripting. Formerly known as OLE automation.

Availability Optimum availability of a solution means that it can be reached from the most locations possible and during the greatest possible amount of time.

B

BMP The extension for bitmap files, a popular file graphics format supported in VB.

Branch The act of creating two copies of a Visual SourceSafe project or file that will henceforth have independent paths of modification.

Business object A logical and physical entity that represents a physical or conceptual entity connected with the business environment at hand. Examples of Business objects might be Customer, Employee, Sales Order, or Account. Developers often implemented Business objects as object classes in a VB environment.

C

CAB file See *Cabinet file*.

Cabinet file The standard compression format used in setup packages created by the Package & Deployment Wizard. You can use the MakeCab utility to create CAB files manually.

Call stack The hierarchical list of procedure calls that lead up to the currently executing line of code.

Class The formal definition or template used to create an object. The class acts as the template from which an instance of an object is created at runtime. The class defines the properties of the object and the methods used to control the object's behavior.

Collection An object that contains a group of member objects of the same type. Collections are somewhat similar to arrays, but they have more functionality (such as Add and Remove methods), and you can access their members with textual keys as well as numeric indexes. You can also use the special VB For each...Next construct to loop through a collection's members.

COM client An ActiveX document or VB program that uses a COM component.

COM component A program that provides objects that are implemented based on the COM specification. Types of COM components available in VB include ActiveX DLLs and EXEs, and ActiveX controls.

COM The Component Object Model. COM is a general industrywide specification for implementing objects in a computing environment. It is the basis for Microsoft's ActiveX standard.

Compile error An error that occurs during compile time because of mistakes in the programmer's use of the rules of the programming language.

Compiler constant In the VB environment, a specially declared name that can be assigned a value recognized by the compiler as it reads the source code. The compiler can then react to the value of the compiler constant when the compiler constant is used in compiler directives. Compiler constants are not considered part of the actual program.

Compiler directive A special logical directive to the compiler that the VB programmer can embed in source code. The compiler directive typically uses the value of a compiler constant to instruct the compiler about whether to compile a particular section of code. Compiler directives are syntactically similar to the `If..Then..Else..End If` structure of the VB language, except that each of the keywords must begin with a # character. Compiler directives are not considered part of the actual program.

Compiler A utility used in the development of computer programs that reads source code, applies to it the rules of the programming language, and then produces object code or executable files.

Composite control An ActiveX control that uses other controls (known as *constituent controls*) for all or part of its functionality.

Conceptual Design The phase of solution development that identifies user scenarios for the solution.

Constituent control A "building block" standard or third-party control used as part of a new ActiveX control. See *Composite control*.

Concurrency In software solutions that require more than one user to access data at the same time, the term concurrency refers to the problems that come up when two or more users or processes try to update the same data at the same time.

Consumer See *Data consumer*.

Context ID The term used to describe topic IDs in WinHelp Help File development.

Context object An object automatically created by the MTS runtime for every running instance of all MTS objects. It provides the state of its associated MTS object, including memory resources, transaction information, and identity of the objects creator.

Context-sensitive help Help in a Windows program that responds differently according to which control the user has currently selected.

Context-sensitive menu See *Pop-up menu*.

Controls collection A built-in VB collection whose members include all the controls on the current form.

Cursor See *Data cursor*.

D

DAO See *Data Access Objects*.

Data Access Objects (DAO) An object library that represents the functionality of the Microsoft Jet database engine (Microsoft Access). Programmers can use DAO to gain access to other types of data through ODBC as well. DAO was the principal data access model for version of VB before VB6. However, new VB6 development should be done with ADO.

Data consumer An object (such as a control) that refers or binds to the data provided by a data source. In VB, controls that are data consumers have `DataField` and `DataSource` properties. Also an object or application that uses a data provider to obtain a connection to data and to manipulate that data.

Data cursor A set of resources initiated by a data connection that provides a connection to a specific row in a set of data. The data cursor can change position to point to a different row of data. In the ADO environment,

data cursors can be implemented either as client-side or as server-side cursors. In addition, several different types of cursors provide differing degrees of flexibility, performance, and efficiency.

Data provider In an ADO programming environment, refers to a driver for data access running under the OLE DB data-access standard.

Data source An object (such as a control) that provides a data connection for data consumers.

Data Source Name (DSN) In ODBC, the name assigned by a developer or administrator to a specific data connection that has been defined in the ODBC environment. A DSN encapsulates specific information about a data connection which may include such information as the type of data driver, the location of the data, logon information, and other vendor-specific parameters. Programmers can use the DSN without knowing about the specific details of the connection that it represents.

Database Management System (DBMS) Software used to create, examine, organize, and modify information stored in a database. Examples of DBMSs are Access, Oracle, BTrieve, SQL Server, and Sybase.

DBMS See *Database Management System*.

DCOM A standard that extends the interobject communication specifications of COM across multiple machines over a network.

DDF file A text file read by the MakeCab utility to determine how to build the CAB files and which source files to use.

Declarative security An approach to MTS security in which the built-in features of MTS and Windows NT are exploited. In this model, MTS checks that clients are authorized to use a specific component or interface. It can be configured from the MTS Explorer, and does not require additional implementation code inside the component.

Delegated member A property, event, or method of a constituent control that is exposed to programmers through a corresponding custom member of an ActiveX control. Developers using the ActiveX component can't access the constituentís members directly, so the VB programmer creating the ActiveX control must create special wrapper members in the control that refer to the members of the underlying constituent controls.

DEP file See *Dependency file*.

Dependency file A text file with the same name as the file whose dependencies it describes. The contents of a dependency file look like an INI file: section headers set off by square brackets ([ ])and individual key entries of the format `Keyname=value` .

Deployment The act of placing setup package files in the location where they will be used to install the application.

Design time Refers to the time that the VB developer spends in the VB IDE when an instance of the application is not running.

Designer A window provided by the VB design-time IDE for creating and implementing VB's major container objects. VB provides designers for forms, user documents (ActiveX documents), user controls (ActiveX controls), and property pages.

DHTML (Dynamic HTML) An enhancement to the HTML standard that enables programmers to program HTML pages with methods, events, and properties and to free the HTML page from too much server-side processing.

DHTML application A type of VB application (new with VB6) that enables programmers to use the VB IDE to create applications that use DHTML. DHTML

applications are deployed on individual user machines as an ActiveX DLL and some associated runtime files.

Difference The act of comparing the differences between two Visual SourceSafe versions of the same file.

DOB The extension for a file containing the source code for a VB UserDocument.

DSN See *Data Source Name*.

E

Early binding The technique of programming with an object class so that the compiler is aware of the object class members and can verify the programmer's syntax when manipulating the object.

Enterprise Application Model Microsoft's latest framework for discussing application development. The Enterprise Application Model uses six complementary models for looking at a development project, known as the Development, Business, User, Logical, Technology, and Physical models. The Enterprise Application Model is not referred to in the certification exam.

Enterprise Development Model A slightly older set of Microsoft concepts for application development than the Enterprise Application Model. The Enterprise Development Model sees three phases of application design: Conceptual, Logical, and Physical design. The Enterprise Development Model is the basis for the design-oriented questions on the VB6 Certification Exam.

Error handling Code created to specifically deal with some types of errors at runtime.

Error number A number that indicates which error occurred. In Visual Basic, you can find the current error number through the `Err` object's `Number` property (`Err.number`).

Event An action performed by hardware, the user, or some other component of a computer system.

Event procedure A predefined VB procedure that runs in association with an event. The programmer can put code in the event procedure. Event procedures may run apart from their associated events.

Executable file The end product of the programming process that is distributed to users. Programmers of compiled languages must use compilers as part of the production process for executable files.

Extensibility 1) In the context of the VB development environment, extensibility means the capability to integrate other applications (such as Visual SourceSafe, Component Object Manager, Visual Data Manager, and many others) into the development environment. 2) In the context of the VB6 certification exam objectives, extensibility is best understood as the capability to use a core set of application services for new purposes in the future, purposes which the original developers may not have foreseen.

F–G

Fatal error A type of runtime error that generates an error message, stops execution, and closes the program.

Forms collection A built-in VB collection whose members include all the forms currently loaded by the application.

H

HTML (Hypertext Markup Language) The standard language recognized by all Internet Web browsers (such as Internet Explorer or Netscape). HTML files reside on

a server. A browser requests the HTML file from a server when the user navigates to a particular Web page representing the HTML file. The server sends the file to the browser, and the browser interprets the HTML script inside the file to display the Web page to the user.

HTML Help Microsoft's current format for Help files. HTML Help replaces the older format.

I

ICO The extension for icon files.

IDE Stands for Integrated Development Environment, that is, the programming interface for VB.

Identity Defines how the components in the package will be identified on the network. It will either be a Windows NT user, or default to the user who is interactively logged on to the NT system that is running MTS.

IIS (Internet Information Server) Microsoft's Web server.

IIS application A type of VB application (new with VB6) that enables a VB programmer to use the VB IDE to enhance HTML templates with ASP code before submitting them to the Internet Information Server in response to client requests. IIS applications run on Web servers. They require no change on client machines because their final output is pure HTML.

Immediate window A debugging window in the VB IDE that enables the programmer to query and set variables, and run single-line statements on-the-fly at runtime.

Implicit loading The action of loading an object (usually a form) into memory without an explicit command to load it. Implicit loading can happen when the program refers to any of the object's members (properties or methods) in code.

INF file A text file included in Internet download packages that contains dependency information needed by the end user's browser. Based on the information in the INF file, the browser will download and install other files that your application needs to run.

Input validation A general term describing the means by which an application examines user input to determine whether it is acceptable to the application and then takes steps to correct problematic input.

Instance Refers to an object created from a specified class. For example, the `xl` object might be an instance of the `Excel.Application` class.

Instantiate To create an instance of a class at runtime (that is, to create an object).

Interpreter A utility used to interpret source-code programs line by line and then run each line in the platform's native language. In the computing world, interpreters are the main alternative to compiled programs.

Intrinsic control A standard control always found in the VB environment.

J–L

Jet The database engine underlying Microsoft Access and DAO.

Late binding The technique of programming with an object class so that the compiler is unaware of the underlying object model and therefore does not verify the syntax that the programmer uses to manipulate the object. Instead, syntax errors in the use of the object model will cause runtime errors.

Library package Packages activated as library packages run in-process with their clients. Clients that call components in a library package must be running on the same machine as the MTS installation.

Load balancing The process by which workload is spread among two or more physical servers to prevent bottlenecks on a single machine.

Locals window A debugging window in VB that displays all variables available in the current context.

Locking In the context of database programming, record- or data-locking refers to the act by a process of blocking other processes or users from gaining access to the data in, for example, a record, a table, or an entire database.

Logical design The phase in solution development that uses the user scenarios of the Conceptual Design to identify business objects for the solution.

M

Macro In the context of Package & Deployment Wizard, a distinctive token that represents a standard path on a machine that has Windows installed. The macro (such as $(WinSysPathSysFile)) is embedded in the SETUP.LST file and when the setup routine runs, it is resolved into the actual path on the user's machine.

Maintainability The ease with which a software system can be changed without disrupting service and with the least and simplest efforts by software developers.

Merge The act of combining together in Visual SourceSafe two different versions of a file.

Modal Describes a process or object that causes the current thread to halt until the modal process ends or the modal object unloads. The MsgBox, for example, is modal. You can also cause a form to run modally by calling its Show method and passing vbModal as its argument. Note that MDI child forms cannot be shown modally.

Modeless Describes a process or object (such as a form) that allows the current thread to continue executing even as the modeless process runs or while the modeless object stays loaded. By default, forms operate modelessly when activated with the Show method—no parameter is necessary. For clarity, however, you may add the parameter vbModeless . When a Timer control's Timer event procedure runs, it is modeless with respect to the rest of the code and objects in your application.

MTS client package A setup program used to configure a client to use existing components in an MTS package. It registers the remote components, and copies support files necessary to call the components from a client machine.

MTS Explorer A Microsoft Message Console snap-in that provides a GUI to administer and view information about servers running Microsoft Transaction Server.

N–O

Native code Machine-readable instructions that have been translated by a compiler to the low-level language of the hardware or operating system platform on which they are intended to run. This is one of the two possible formats for a compiled VB executable file (the other is pseudocode). Native code is the default format for VB applications and has the same format as a compiled Visual C++ executable.

Object Browser A Visual Basic design-time dialog box that provides a list of available object libraries and information about the objects for a specified library.

Object code The end product of a compiler. In many programming environments (but not VB), object code must then be *linked* with other elements to produce the final program that can be distributed to users.

Object Model The complete list of a COM component's classes, their members (events, methods, and properties), their hierarchical relation to each other, and their behaviors.

OCX The extension for files that implement 32-bit ActiveX controls.

ODBC See *Open Database Connectivity*.

ODBC Resource Dispenser An MTS resource dispenser that allows objects to share ODBC connections. This reduces the overhead associated with obtaining database connections, and provides for greater scalability.

OLE (object linking and embedding) A standard that gives rules for defining interfaces between applications and objects. The ActiveX standard has superseded the OLE standard.

OLE DB Microsoft's newest standard for Windows data access, and the standard underlying ADO. Microsoft intends for OLE DB to eventually replace the ODBC standard. OLE DB is COM based and provides generally more efficient and more universal data access than ODBC. OLE DB is in turn based on Microsoft's Universal Data-Access Model.

Open Database Connectivity (ODBC) A standard for data-access drivers that Windows programmers can use to transparently access most major DBMSs, such as Oracle, BTrieve, or SQL Server. The vendors of each DBMS create ODBC-compliant drivers, and Windows programmers can then use another layer of software known as the ODBC driver manager to talk to the DBMS-specific drivers and so gain access to the data. ODBC will eventually give way to OLE DB.

Option pack Free add-on to Windows NT. It includes key development items such as MTS, IIS, and ASP.

P

Package Package has various meanings in different contexts. In MTS, a package is a logical grouping of components. Components in an MTS package utilize the same roles for security purposes, run in the same server process, and are grouped together for client deployment. Generally components in the same MTS package share responsibility in an application. A package in VB can also mean a deployment package. This is just a setup program or an Internet download that includes the distribution files for a VB program or ActiveX component.

Package & Deployment Wizard This VB wizard automates the steps necessary to create a setup package. The setup package contains the compiled version of a project, such as an EXE, OCX, or DLL, and all the necessary support files.

Package file (PAK) A file that is the result of exporting an MTS package. It can be imported into an MTS system to duplicate a package and its contents.

Performance The capability of a software system to perform critical operations rapidly and with the least-possible use of system resources.

Persist To cause a control's property to "remember" its value between design time and runtime. ActiveX component properties are said to persist when they retain their programmer-assigned values appropriately during the development/runtime cycle. The concept of persistence is used with custom properties in VB ActiveX document and ActiveX control applications. When an instance of a custom control is instantiated in a running copy of a component, the VB developer who created the component needs to ensure that the control's custom properties will retain the values that developers assign to them at design time. In addition, property values that a user assigns when running an ActiveX document in its container application need to persist so that they can be retrieved when the user re-enters the document later.

Physical design The phase in solution development that identifies the specific implementations of the logical design, including the specific hardware and software solutions.

Pin The act of locking in a particular version of a file in a SourceSafe project. That file may not be changed within the project where it's pinned. If the file is pinned before sharing, then the projects using the file cannot make changes to it.

Pixel A unit of measure in many graphics environments. A pixel's size depends on the system's graphics device settings because it corresponds to the size of a single video dot on the screen.

Pop-up menu A menu with subitems that can appear apart from a form's menu toolbar, usually after clicking the alternate mouse button. The pop-up menu may or may not be visible on the form's menu toolbar as well.

ProgID A unique identifying string that refers to an object type registered on a particular computer workstation's Windows operating system. You must use the ProgID of a control type when you want to add an instance of that control type to the Controls collection of a form.

Programmatic security An approach to MTS security where the component developer implements the security logic directly in the code of an MTS component.

Project group A technique used in VB to associate projects together in one instance of VB for testing, debugging, and project management.

Property Page A tabbed dialog box that enables a developer to specify properties in an organized format. VB developers can create property pages for their ActiveX controls.

Provider See *Data provider*.

Pseudocode 1) One of the two possible formats for a compiled VB executable file (the other is native code). A VB6 executable compiled as pseudocode contains a tokenized version of the original source code. The VB6 runtime libraries can interpret this code when the application runs. This first meaning is the usual one intended in a VB environment. 2) A format for writing design specifications for program logic, also known as "structured English." The writer of the specifications typically uses the control structures (such as looping and branching) of the programming language, but uses more natural language to specify actions and decisions.

R

Reader role A default role that exists in the System package in MTS. Users mapped to the Reader role can browse components in the MTS Explorer. However, they cannot create new packages or modify or delete existing ones.

Reference counting A technique used by COM's Object Management services that helps to control when object references and their objects are released from memory.

Remote support file A file with extension .VBR created when you compile a client or server project using DCOM.

Resource Dispenser A service of MTS that provides shared access to common resources for multiple instances of MTS objects. An example is the ODBC Resource Dispenser, which allows MTS objects to share ODBC connections.

Right mouse menu See *Pop-up menu*.

Role A grouping of users that can be used to apply security to components, and component interfaces

running in an MTS package. A role can be assigned to a component or component interface, which in turn defines who can use it.

Rowset The set of data behind a data cursor.

Runtime Refers to the time that a VB application is actually executing, either as a compiled application or from the VB IDE.

S

Scalability An attribute of a software system that refers to the capability of the system to run in a more demanding environment than the environment in which it was originally implemented. Examples of a more demanding environment might include more users, a higher-end DBMS (Oracle or SQL Server as opposed to MS Access), or more traffic.

Scope The area of a program over which a variable is known. In debugging, this applies to the area in which a Watch expression is considered to be valid.

Security In the context of software solution design, the capability of a business solution to deny unauthorized access to each of its components.

Server package Packages activated as server packages run in separate processes in the MTS runtime environment. Calls to components in server packages are marshaled across process boundaries.

SetAbort A method of the Context object. When this method is called, the MTS runtime environment is notified that the work associated with the MTS object's transactions did not complete successfully, and should not be committed. Resources used by the object are released when this method is called.

SetComplete A method of the Context object. When this is called, the MTS runtime environment is notified that all the work for an object's transaction is complete, and can be committed. Also, calling this method will release any resources used by the object without actually destroying the object.

Share The act of making one or more of a project's files available to other projects. The master files are stored in the Visual SourceSafe database and appear in other projects because of pointers to the masters. There is only one copy of each shared file, so any changes to it are seen in all projects that use this file.

Shared Property Manager An MTS resource dispenser that allows objects with different creators to share property values.

Shortcut key A key that the user can press (usually in combination with the Ctrl key) to perform an action. Shortcut keys are typically associated with items in a standard Windows menu and have a visual cue associated with the menu item on the menu line. Shortcut keys are available even when their corresponding menu item is not available. Compare with *Access key*.

Siting The action performed by an ActiveX Container application to instantiate and display an Active document.

Source code Human-readable instructions, typically in the form of text files and/or visual representations (such as VB designers) that follow the rules of a particular programming language, such as VB. For a computer to be able to run the programs specified in source code, the source code must either be translated by a compiler to native code or pseudocode, or else it must be interpreted on-the-fly by an interpreter.

SQL Server Microsoft's Enterprise-scale DBMS.

Standard control A control that is always included in the toolbox and contained within a VB EXE file. Some standard controls in VB include the CommandButton, TextBox, and the Label.

Startup object The object, such as a form or Sub Main procedure, that is the entry point for program execution when an application starts running. In VB6, you specify the startup object in the Project, Properties dialog box.

Submenu A menu item that is under a higher-level item in a Windows menu hierarchy. A submenu may in turn have other submenus items underneath it.

Syntax error An error in the use of a programming language's rules. Some syntax errors are caught by the VB editor as the programmer types; other syntax errors are detected as compile errors.

Syntax The defined order and punctuation of components that comprise a programming-language statement.

System package A package containing components used internally by MTS. The components in this package are used by MTS to provide administrative functionality and security features.

T

Toolbox A special window in the VB IDE that displays the VB controls currently available to the programmer in the design environment.

ToolTip A small message that appears when the user pauses the mouse over a control.

Top menu A menu item that is at the highest level in a Windows menu hierarchy. A top menu item appears on the form's menu toolbar.

Topic ID Unique numbers assigned to topics within a Help file. A VB programmer must know these numbers to enable a VB application to invoke their corresponding topics by assigning them as the values of various object's HelpContextID and WhatsThisHelp properties.

Transaction A combination of work performed that is grouped into a single unit. Transactions can be committed or rolled back.

Trappable error An error that can be trapped and handled by an error-handling procedure while an application is running.

Twip The default unit of measure in VB graphics programming. A twip is 1/20 of a point. Because a point is 1/72 of an inch, a twip is therefore 1/1440 of an inch.

U

UDA See *Universal Data-Access Model*.

Universal Data-Access Model (UDA) Microsoft's general specification for data access. UDA is supposed to encompass all types of possible data access, from traditional DBMSs to more "exotic" data formats such as bitmap files, text files, and spreadsheets.

User scenario A deliverable component of the conceptual design phase of solution development. A user scenario describes a user activity as well as the relevant traits of the user supporting the activity.

V

VBD The extension for a Visual Basic Document file. This is the file that an ActiveX container application uses to implement the PropertyBag object of an Active document that was programmed and compiled in VB.

VBR file See *Remote support file*.

Version control The part of computer software development that manages changes to source-code files made by developers. Version control tools, such as

Visual SourceSafe, enable developers and managers to identify and manage versions of a product in the source code. Such tools also help developers to avoid conflicts and confusion when multiple developers work on files in the same project at the same time.

Visual SourceSafe Microsoft's version control tool. Distributed with Visual Studio and Visual Basic's Enterprise Edition.

W

Watch A mechanism for monitoring variables when running programs from the VB IDE.

WebClass A container object in an IIS application that corresponds to an HTML template.

WebItem A named object belonging to an IIS WebClass that can have its own URL and custom-defined events.

WinHelp Microsoft's older format for Help files.

Working folder A physical folder where Visual SourceSafe will place the files for a project when requested by a Visual SourceSafe user. Each user must define a Working folder for a project that he or she wants to work with.

APPENDIX B

Overview of the Certification Process

You must pass rigorous certification exams to become a Microsoft Certified Professional. These closed-book exams provide a valid and reliable measure of your technical proficiency and expertise. Developed in consultation with computer industry professionals who have experience with Microsoft products in the workplace, the exams are conducted by two independent organizations. Sylvan Prometric offers the exams at more than 1,400 Authorized Prometric Testing Centers around the world. Virtual University Enterprises (VUE) testing centers offer exams at over 250 locations.

To schedule an exam, call Sylvan Prometric Testing Centers at 800-755-EXAM (3926) or VUE at 888-837-8616 (or register online with VUE at http://www.vue.com/student-services/). Currently Microsoft offers seven types of certification, based on specific areas of expertise.

Types of Certification

‡ **Microsoft Certified Professional (MCP).** Qualified to provide installation, configuration, and support for users of at least one Microsoft desktop operating system, such as Windows NT Workstation. Candidates can take elective exams to develop areas of specialization. MCP is the base level of expertise.

‡ **Microsoft Certified Professional+Internet (MCP+Internet).** Qualified to plan security, install and configure server products, manage server resources, extend service to run CGI scripts or ISAPI scripts, monitor and analyze performance, and troubleshoot problems. Expertise is similar to that of an MCP but with a focus on the Internet.

‡ **Microsoft Certified Professional+Site Building (MCP+Site Building).** Qualified to plan, build, maintain, and manage Web sites using Microsoft technologies and products. The credential is appropriate for people who manage sophisticated, interactive Web sites that include database connectivity, multimedia, and searchable content.

‡ **Microsoft Certified Systems Engineer (MCSE).** Qualified to effectively plan, implement, maintain, and support information systems with Microsoft Windows NT and other Microsoft advanced systems and workgroup products, such as Microsoft Office and Microsoft BackOffice. MCSE is a second level of expertise.

‡ **Microsoft Certified Systems Engineer+Internet (MCSE+Internet).** Qualified in the core MCSE areas, and also qualified to enhance, deploy, and manage sophisticated intranet and Internet solutions that include a browser, proxy server, host servers, database, and messaging and commerce components. A MCSE+Internet-certified professional is able to manage and analyze Web sites.

‡ **Microsoft Certified Solution Developer (MCSD).** Qualified to design and develop custom business solutions by using Microsoft development tools, technologies, and platforms, including Microsoft Office and Microsoft BackOffice. MCSD is a second level of expertise with a focus on software development.

‡ **Microsoft Certified Trainer (MCT).** Instructionally and technically qualified by Microsoft to deliver Microsoft Education Courses at Microsoft-authorized sites. An MCT must be employed by a Microsoft Solution Provider Authorized Technical Education Center or a Microsoft Authorized Academic Training site.

NOTE

For up-to-date information about each type of certification, visit the Microsoft Training and Certification World Wide Web site at http://www.microsoft.com/train_cert . You must have an Internet account and a WWW browser to access this information. You also can contact the following sources:

- Microsoft Certified Professional Program: 800-636-7544
- **mcp@msource.com**
- Microsoft Online Institute (MOLI): 800-449-9333

CERTIFICATION REQUIREMENTS

An asterisk following an exam in any of the lists below means that it is slated for retirement.

How to Become a Microsoft Certified Professional

Passing any Microsoft exam (with the exception of Networking Essentials) is all you need to do to become certified as a MCP.

How to Become a Microsoft Certified Professional+Internet

You must pass the following exams to become a MCP specializing in Internet technology:

‡ Internetworking Microsoft TCP/IP on Microsoft Windows NT 4.0, #70-059

‡ Implementing and Supporting Microsoft Windows NT Server 4.0, #70-067

‡ Implementing and Supporting Microsoft Internet Information Server 3.0 and Microsoft Index Server 1.1, #70-077

OR Implementing and Supporting Microsoft Internet Information Server 4.0, #70-087

How to Become a Microsoft Certified Professional+Site Building

You need to pass two of the following exams in order to be certified as an MCP+Site Building

- ‡ Designing and Implementing Web Sites with Microsoft FrontPage 98, #70-055
- ‡ Designing and Implementing Commerce Solutions with Microsoft Site Server 3.0, Commerce Edition, #70-057
- ‡ Designing and Implementing Web Solutions with Microsoft Visual InterDev 6.0, #70-152

How to Become a Microsoft Certified Systems Engineer

You must pass four operating system exams and two elective exams to become a MCSE. The MCSE certification path is divided into two tracks: Windows NT 3.51 and Windows NT 4.0.

The following lists show the core requirements (four operating system exams) for both the Windows NT 3.51 and 4.0 tracks and the electives (two exams) you can take for either track.

Windows NT 3.51 Track

The Windows NT 3.51 Track will probably be retired with the release of Windows NT 5.0. The Windows NT 3.51 core exams are scheduled for retirement at that time.

Core Exams

The four Windows NT 3.51 Track Core Requirements for MCSE certification are as follows:

- ‡ Implementing and Supporting Microsoft Windows NT Server 3.51, #70-043*
- ‡ Implementing and Supporting Microsoft Windows NT Workstation 3.51, #70-042*
- ‡ Microsoft Windows 3.1, #70-030*

  OR Microsoft Windows for Workgroups 3.11, #70-048*

  OR Implementing and Supporting Microsoft Windows 95, #70-064

  OR Implementing and Supporting Microsoft Windows 98, #70-098
- ‡ Networking Essentials, #70-058

Windows NT 4.0 Track

The Windows NT 4.0 track is also organized around core and elective exams.

Core Exams

The four Windows NT 4.0 Track Core Requirements for MCSE certification are as follows:

- ‡ Implementing and Supporting Microsoft Windows NT Server 4.0, #70-067
- ‡ Implementing and Supporting Microsoft Windows NT Server 4.0 in the Enterprise, #70-068
- ‡ Microsoft Windows 3.1, #70-030*

  OR Microsoft Windows for Workgroups 3.11, #70-048*

OR Implementing and Supporting Microsoft Windows 95, #70-064

OR Implementing and Supporting Microsoft Windows NT Workstation 4.0, #70-073

OR Implementing and Supporting Microsoft Windows 98, #70-098

‡ Networking Essentials, #70-058

Elective Exams

For both the Windows NT 3.51 and the 4.0 track, you must pass two of the following elective exams for MCSE certification:

‡ Implementing and Supporting Microsoft SNA Server 3.0, #70-013

OR Implementing and Supporting Microsoft SNA Server 4.0, #70-085

‡ Implementing and Supporting Microsoft Systems Management Server 1.0, #70-014*

OR Implementing and Supporting Microsoft Systems Management Server 1.2, #70-018

OR Implementing and Supporting Microsoft Systems Management Server 2.0, #70-086

‡ Microsoft SQL Server 4.2 Database Implementation, #70-021

OR Implementing a Database Design on Microsoft SQL Server 6.5, #70-027

OR Implementing a Database Design on Microsoft SQL Server 7.0, #70-029

‡ Microsoft SQL Server 4.2 Database Administration for Microsoft Windows NT, #70-022

OR System Administration for Microsoft SQL Server 6.5 (or 6.0), #70-026

OR System Administration for Microsoft SQL Server 7.0, #70-028

‡ Microsoft Mail for PC Networks 3.2-Enterprise, #70-037

‡ Internetworking with Microsoft TCP/IP on Microsoft Windows NT (3.5-3.51), #70-053

OR Internetworking with Microsoft TCP/IP on Microsoft Windows NT 4.0, #70-059

‡ Implementing and Supporting Microsoft Exchange Server 4.0, #70-075*

OR Implementing and Supporting Microsoft Exchange Server 5.0, #70-076

OR Implementing and Supporting Microsoft Exchange Server 5.5, #70-081

‡ Implementing and Supporting Microsoft Internet Information Server 3.0 and Microsoft Index Server 1.1, #70-077

OR Implementing and Supporting Microsoft Internet Information Server 4.0, #70-087

‡ Implementing and Supporting Microsoft Proxy Server 1.0, #70-078

OR Implementing and Supporting Microsoft Proxy Server 2.0, #70-088

‡ Implementing and Supporting Microsoft Internet Explorer 4.0 by Using the Internet Explorer Resource Kit, #70-079

How to Become a Microsoft Certified Systems Engineer+Internet

You must pass seven operating system exams and two elective exams to become a MCSE specializing in Internet technology.

Core Exams

The seven MCSE+Internet core exams required for certification are as follows:

- Networking Essentials, #70-058
- Internetworking with Microsoft TCP/IP on Microsoft Windows NT 4.0, #70-059
- Implementing and Supporting Microsoft Windows 95, #70-064

  OR Implementing and Supporting Microsoft Windows NT Workstation 4.0, #70-073

  OR Implementing and Supporting Microsoft Windows 98, #70-098
- Implementing and Supporting Microsoft Windows NT Server 4.0, #70-067
- Implementing and Supporting Microsoft Windows NT Server 4.0 in the Enterprise, #70-068
- Implementing and Supporting Microsoft Internet Information Server 3.0 and Microsoft Index Server 1.1, #70-077

  OR Implementing and Supporting Microsoft Internet Information Server 4.0, #70-087
- Implementing and Supporting Microsoft Internet Explorer 4.0 by Using the Internet Explorer Resource Kit, #70-079

Elective Exams

You must also pass two of the following elective exams for MCSE+Internet certification:

- System Administration for Microsoft SQL Server 6.5, #70-026
- Implementing a Database Design on Microsoft SQL Server 6.5, #70-027
- Implementing and Supporting Web Sites Using Microsoft Site Server 3.0, # 70-056
- Implementing and Supporting Microsoft Exchange Server 5.0, #70-076

  OR Implementing and Supporting Microsoft Exchange Server 5.5, #70-081
- Implementing and Supporting Microsoft Proxy Server 1.0, #70-078

  OR Implementing and Supporting Microsoft Proxy Server 2.0, #70-088
- Implementing and Supporting Microsoft SNA Server 4.0, #70-085

How to Become a Microsoft Certified Solution Developer

The MCSD certification is undergoing substantial revision. Listed below are the requirements for the new track (available fourth quarter 1998) as well as the old.

New Track

For the new track, you must pass three core exams and one elective exam. The three core exam areas are listed below as well as the elective exams from which you can choose.

The core exams include the following:

Desktop Applications Development (one required)

- Designing and Implementing Desktop Applications with Microsoft Visual C++ 6.0, #70-016

  OR Designing and Implementing Desktop Applications with Microsoft Visual Basic 6.0, #70-176

Distributed Applications Development (one required)

- Designing and Implementing Distributed Applications with Microsoft Visual C++ 6.0, #70-015

  OR Designing and Implementing Distributed Applications with Microsoft Visual Basic 6.0, #70-175

Solution Architecture (required)

- Analyzing Requirements and Defining Solution Architectures, #70-100

You must pass one of the following elective exams:

- Designing and Implementing Distributed Applications with Microsoft Visual C++ 6.0, #70-015

  OR Designing and Implementing Desktop Applications with Microsoft Visual C++ 6.0, #70-016

  OR Microsoft SQL Server 4.2 Database Implementation, #70-021*

- Implementing a Database Design on Microsoft SQL Server 6.5, #70-027

  OR Implementing a Database Design on Microsoft SQL Server 7.0, #70-029

- Developing Applications with C++ Using the Microsoft Foundation Class Library, #70-024
- Implementing OLE in Microsoft Foundation Class Applications, #70-025
- Designing and Implementing Web Sites with Microsoft FrontPage 98, #70-055
- Designing and Implementing Commerce Solutions with Microsoft Site Server 3.0, Commerce Edition, #70-057
- Programming with Microsoft Visual Basic 4.0, #70-065

  OR Developing Applications with Microsoft Visual Basic 5.0, #70-165

  OR Designing and Implementing Distributed Applications with Microsoft Visual Basic 6.0, #70-175

  OR Designing and Implementing Desktop Applications with Microsoft Visual Basic 6.0, #70-176

- Microsoft Access for Windows 95 and the Microsoft Access Development Toolkit, #70-069
- Designing and Implementing Solutions with Microsoft Office (Code-named Office 9) and Microsoft Visual Basic for Applications, #70-091
- Designing and Implementing Web Solutions with Microsoft Visual InterDev 6.0, #70-152

Old Track

For the old track, you must pass two core technology exams and two elective exams for MCSD certification. The following lists show the required technology exams and elective exams needed to become an MCSD.

Core Technology Exams

You must pass the following two core technology exams to qualify for MCSD certification:

- ‡ Microsoft Windows Architecture I, #70-160*
- ‡ Microsoft Windows Architecture II, #70-161*

Elective Exams

You must also pass two of the following elective exams to become an MSCD:

- ‡ Designing and Implementing Distributed Applications with Microsoft Visual C++ 6.0, #70-015
- ‡ Designing and Implementing Desktop Applications with Microsoft Visual C++ 6.0, #70-016
- ‡ Microsoft SQL Server 4.2 Database Implementation, #70-021*

  OR Implementing a Database Design on Microsoft SQL Server 6.5, #70-027

  OR Implementing a Database Design on Microsoft SQL Server 7.0, #70-029
- ‡ Developing Applications with C++ Using the Microsoft Foundation Class Library, #70-024
- ‡ Implementing OLE in Microsoft Foundation Class Applications, #70-025
- ‡ Programming with Microsoft Visual Basic 4.0, #70-065

  OR Developing Applications with Microsoft Visual Basic 5.0, #70-165

  OR Designing and Implementing Distributed Applications with Microsoft Visual Basic 6.0, #70-175

  OR Designing and Implementing Desktop Applications with Microsoft Visual Basic 6.0, #70-176
- ‡ Microsoft Access 2.0 for Windows-Application Development, #70-051

  OR Microsoft Access for Windows 95 and the Microsoft Access Development Toolkit, #70-069
- ‡ Developing Applications with Microsoft Excel 5.0 Using Visual Basic for Applications, #70-052
- ‡ Programming in Microsoft Visual FoxPro 3.0 for Windows, #70-054
- ‡ Designing and Implementing Web Sites with Microsoft FrontPage 98, #70-055
- ‡ Designing and Implementing Commerce Solutions with Microsoft Site Server 3.0, Commerce Edition, #70-057
- ‡ Designing and Implementing Solutions with Microsoft Office (Code-named Office 9) and Microsoft Visual Basic for Applications, #70-091
- ‡ Designing and Implementing Web Solutions with Microsoft Visual InterDev 6.0, #70-152

Becoming a Microsoft Certified Trainer

To understand the requirements and process for becoming a MCT, you need to obtain the Microsoft Certified Trainer Guide document from the following WWW site:

http://www.microsoft.com/train_cert/mct/

At this site you can read the document as Web pages or display and download it as a Word file. The MCT Guide explains the four-step process of becoming a MCT. The general steps for the MCT certification are as follows:

1. Complete and mail a Microsoft Certified Trainer application to Microsoft. You must include proof of your skills for presenting instructional material. The options for doing so are described in the MCT Guide.
2. Obtain and study the Microsoft Trainer Kit for the Microsoft Official Curricula (MOC) courses for which you want to be certified. Microsoft Trainer Kits can be ordered by calling 800-688-0496 in North America. Those of you in other regions should review the MCT Guide for information on how to order a Trainer Kit.
3. Take the Microsoft certification exam for the product about which you want to be certified to teach.
4. Attend the MOC course for the course for which you want to be certified. This is done so you can understand how the course is structured, how labs are completed, and how the course flows.

If you are interested in becoming a MCT, you can obtain more information by visiting the Microsoft Certified Training WWW site at http://www.microsoft.com/train_cert/mct/ or by calling 800-688-0496.

WARNING

You should consider the preceding steps a general overview of the MCT certification process. The precise steps that you need to take are described in detail on the WWW site mentioned earlier. Do not misinterpret the preceding steps as the exact process you need to undergo.

APPENDIX C

What's On the CD-ROM

This appendix is a brief rundown of what you'll find on the CD-ROM that comes with this book. For a more detailed description of the newly developed Top Score test engine, exclusive to Macmillan Computer Publishing, please see Appendix D, "Using the Top Score Software."

Top Score

Top Score is a test engine developed exclusively for Macmillan Computer Publishing. It is, we believe, the best test engine available because it closely emulates the format of the standard Microsoft exams. In addition to providing a means of evaluating your knowledge of the exam material, Top Score features several innovations that help you to improve your mastery of the subject matter.

For example, the practice tests allow you to check your score by exam area or category to determine which topics you need to study further. Other modes allow you to obtain immediate feedback on your responses, explanations of correct answers, and even hyperlinks to the chapter in an electronic version of the book where the topic is covered. Again, for a complete description of the benefits of Top Score, see Appendix D.

Before running the Top Score software, be sure that AutoRun is enabled. If you prefer not to use AutoRun, then you can run the application from the CD by double-clicking the START.EXE file from Explorer.

Exclusive Electronic Version of Text

As referred to above, the CD-ROM also contains the electronic version of this book in Portable Document Format (PDF). In addition to the links to the book that are built into Top Score, you can use this version to help search for terms you need to study or other book elements. The electronic version comes complete with all figures as they appear in the book.

Copyright Information and Disclaimer

Macmillan Computer Publishing's Top Score test engine: Copyright 1998 New Riders Publishing. All rights reserved. Made in U.S.A.

APPENDIX D

Using the Top Score Software

Getting Started

The installation procedure for the Top Score software is very simple. Put the CD into the CD-ROM drive. The AutoRun function starts and after a moment, you will see the opening screen. Click Exit to quit or Continue to proceed. If you clicked Continue, then you will see a window offering you the choice of launching any of the four Top Score applications.

At this point you are ready to use the Top Score software.

> NOTE
>
> **Getting Started Without AutoRun**
> If you have disabled the AutoRun function, you may start the Top Score Software suite by viewing the contents of the CD-ROM in Explorer and double-clicking START.EXE.

Instructions on Using the Top Score Software

Top Score software consists of the following four applications: Study Cards, Flash Cards, Practice Exams, and Simulator.

Study Cards serve as a study aid organized around the specific exam objectives, arranged in multiple-choice format. Flash Cards, another study aid, require responses to open-ended questions, testing knowledge of the material at a deeper level than simply recognition memory. Practice Exams simulate the Microsoft certification exams. Simulator emulates elements of the Windows NT interface in order to provide you with hands-on experience and practice with simulation questions like those now appearing in new and revised certification exams.

To start the Study Cards, Flash Cards, or Practice Exams applications, click the application you would like to use, then on the next screen click the button that appears centered near the bottom of the screen. The initial screen of the application will appear, and you will be ready to go.

To start Simulator, click the button, then follow the instructions to install it. Once Simulator is installed, it will appear in your Programs menu.

Further details on using the four specific applications follow.

Using Top Score Practice Exams

The Practice Exams interface is simple and straightforward. Its design simulates the look and feel of the Microsoft certification exams. If you followed the two steps above, you should see an opening screen similar to the one shown in Figure D.1.

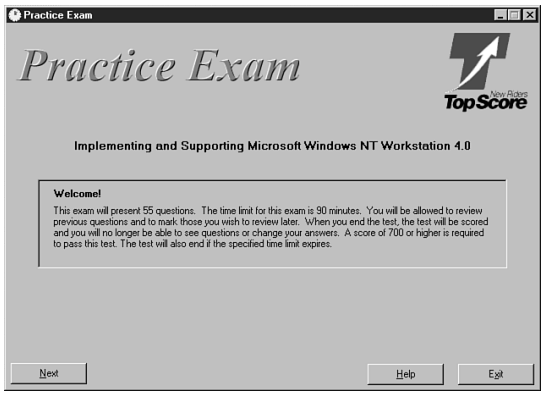


FIGURE D.1▲
Top Score Practice Exams opening screen.

Click Next to see a disclaimer and copyright screen, read the information, and click Top Score's Start button. A notice appears indicating that the program is randomly selecting questions for the practice exam from the exam database (see Figure D.2). Practice exams include the same number of items as the Microsoft exam.

> **NOTE**
> The number of questions will be the same for traditional exams. However, this will not be the case for exams that incorporate the new "adaptive testing" format. In that format, there is no set number of questions. See "Study and Exam Preparation Tips" for more details on this new format.

The items are selected from a larger set of 150 to 900 questions. The random selection of questions from the database takes some time to retrieve. Don't reboot; your machine is not hung!

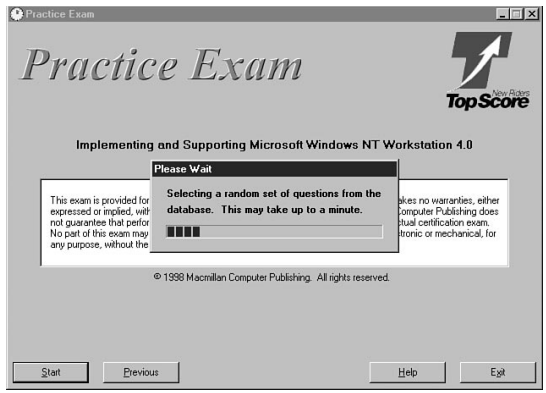


FIGURE D.2▲
Top Score's Please Wait notice.

After the questions have been selected, the first test item appears. See Figure D.3 for an example of a test item screen.

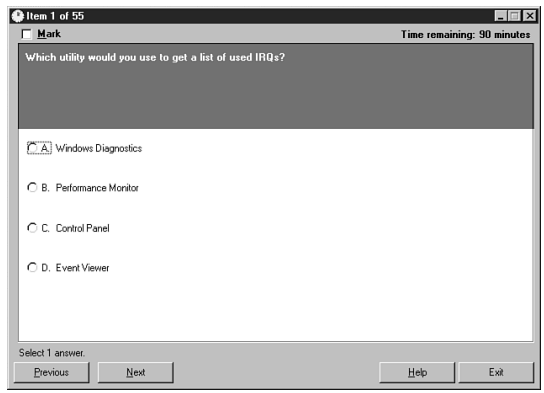


FIGURE D.3▲
A Top Score test item requiring a single response.

Notice several important features of this window. The question number, out of the total number of retrieved questions, is located at the top-left corner of the window in the control bar. Immediately below this is a check box labeled Mark, which enables you to mark any exam item as one you would like to return to later. Across the screen from the check box, you will see the total time remaining for the exam.

The test question is located in a colored section (gray in the figure). Directly below the test question, in the white area, are response choices. Be sure to note that immediately below the responses are instructions about how to respond, including the number of responses required. You will notice that questions requiring a single response, such as that shown in Figure 3, have radio buttons next to the choices. Items requiring multiple responses have check boxes (see Figure D.4).

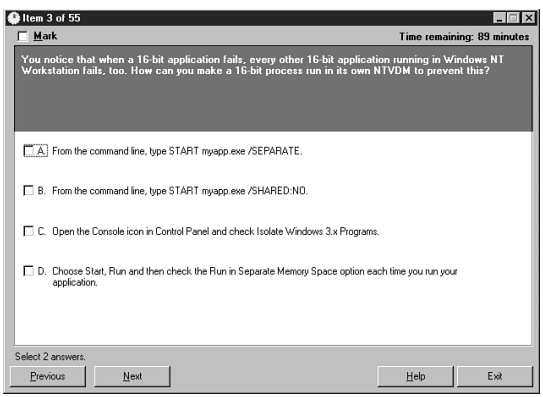


FIGURE D.4▲
A Top Score test item requiring multiple responses.

Some questions and responses do not appear on the screen in their entirety. In these cases a scrollbar appears to the right of the question or response. Use the scrollbar to reveal the rest of the question or response item.

The buttons at the bottom of a window enable you to return to a previous test item, proceed to the next test item, or exit Top Score Practice Exams.

Some items require you to examine additional information called *Exhibits*. These screens typically include graphs, diagrams, or other types of visual information needed to respond to the test question. Exhibits can be accessed by clicking the Exhibit button also located at the bottom of the window.

After you complete the practice test by moving through all the test questions for your exam, you will arrive at a summary screen titled Item Review (see Figure D.5).

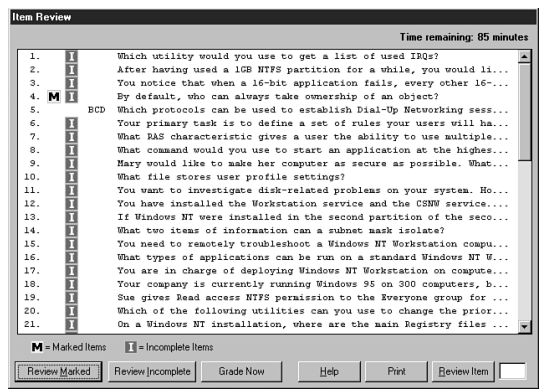


FIGURE D.5▲
The Top Score Item Review window.

This window enables you to see all of the question numbers, your responses to each item, any questions you have marked, and any left incomplete. The buttons at the bottom of the screen enable you to review all the marked items and incomplete items in numeric order.

If you want to review a specific marked or incomplete item, simply type the desired item number in the box at the lower right corner of the window and click the Review Item button. After you review the item, you can respond to the question. Notice that the item window also offers the Next and Previous options. You can also select the Item Review button to return to the Item Review window.

> **NOTE**
> If you exceed the time allotted for the test, you will not have the opportunity to review any marked or incomplete items. The program will move to the next screen.

After you complete your review of the practice test questions, click the Grade Now button to find out how you did. An Examination Score Report will be generated for your practice test (see Figure D.6). This report provides you with the required score for this particular certification exam, your score on the practice test, and a grade. The report also breaks down your performance on the practice test by the specific objectives for the exam. Click the Print button to print out the results of your performance.

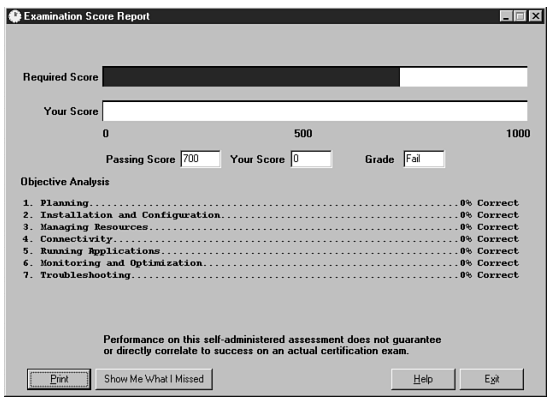


FIGURE D.6▲
The Top Score Examination Score Report window.

You also have the option of reviewing those items that you answered incorrectly. Click the Show Me What I Missed button to receive a summary of those items. Print out this information if you need further practice or review; the printouts can be used to guide your use of Study Cards and Flash Cards.

Using Top Score Study Cards

To start the software, begin from the main screen. Click on the Study Cards button, then on the smaller button displayed in the next screen. After a moment, an initial screen similar to that of the Practice Exams appears.

Click Next to see the first Study Cards screen (see Figure D.7).

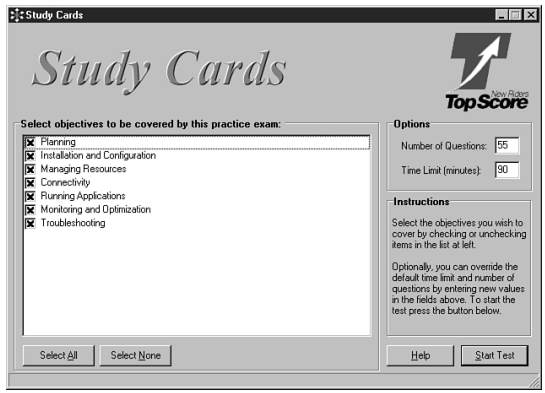


FIGURE D.7▲
The first Study Cards screen.

The interface for Study Cards is similar to that of Practice Exams. However, you have several important options that enable you to prepare for an exam. The Study Cards material is organized using the specific objectives for each exam. You can choose to receive questions on all of the objectives or use the check boxes to select coverage of a limited set of objectives. For example, if you have already completed a Practice Exam and your score report indicates that you need work on Planning, you can choose to cover only the Planning objectives for your Study Cards session.

You can also determine the number of questions to be presented by typing it in the option box at the right of the screen. You can also control the amount of time allowed for a review by typing the number of minutes into the Time Limit option box on the right side.

When you click the Start Test button, Study Cards randomly selects the indicated number of questions from the question database. A dialog box appears, informing you that this process could take some time. After the questions are selected, you will see a first item that looks similar to the one in Figure D.8.

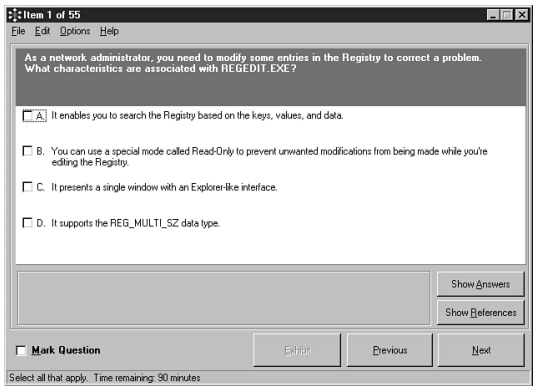


FIGURE D.8▲
A Study Cards item.

Respond to the questions in the same manner as you did to Practice Exam questions. Radio buttons signal that a single answer is required, whereas check boxes indicate that multiple answers are expected.

Notice the menu options at the top of the window. File pulls down to allow you an exit from the program. Edit allows you to use the copy function and even copy questions to the Windows clipboard. The Options pull-down menu allows you to take notes on a particular question. When you pull it down, choose Open Notes. After Notepad opens, type and save your notes. Options also allows you to start over with another exam.

This application provides you with immediate feedback as to whether you answered the question correctly. Click the Show Answers button to see the correct answer(s) highlighted on the screen as shown in Figure D.9.

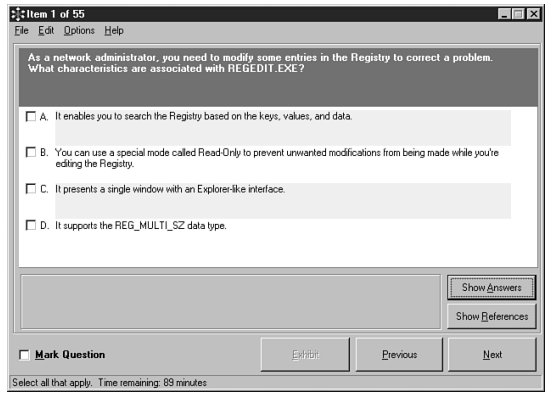


FIGURE D.9▲
Highlighting of the correct answer.

Study Cards also includes Item Review, Score Report, and Show Me What I Missed features that are essentially the same as those in Practice Exams.

Using Top Score Flash Cards

Flash Cards are a third way to use the exam question database. The Flash Cards items do not offer you multiple-choice answers; instead they require you to respond in a short answer or essay format. Flash Cards help you learn the material well enough to respond with the correct answers in your own words. If you have the depth of knowledge to answer questions without prompting, you will certainly be prepared to pass a multiple-choice exam.

Flash Cards are started in the same fashion as Practice Exams and Study Cards. Click the icon next to Flash Cards, then click Start the Program. Click the button for the exam you want and the opening screen will appear. It will look similar to the example in Figure D.10.

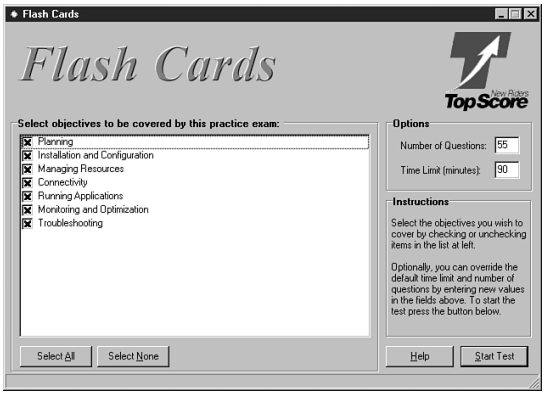


FIGURE D.10▲
The Flash Cards opening screen.

You can choose Flash Cards by the various objectives just as in Study Cards. Select the objectives you want to cover, the number of questions you want, and the amount of time you want to spend. Click the Start Test button to start the Flash Cards session; you will see a dialog box notifying you that questions are being selected.

The Flash Cards items appear in an interface similar to that of Practice Exams and Study Cards (see Figure D.11).

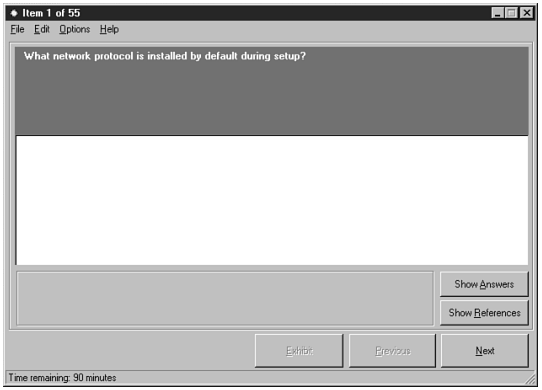


FIGURE D.11▲
A Flash Card item.

Notice, however, that although a question is presented, no answer choices appear. You must type your answer in the white space below the question (see Figure D.12).

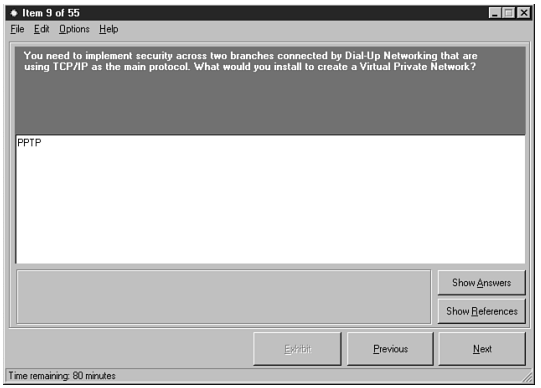


FIGURE D.12▲
A typed answer in Flash Cards.

Compare your answer to the correct answer by clicking the Show Answers button (see Figure D.13).

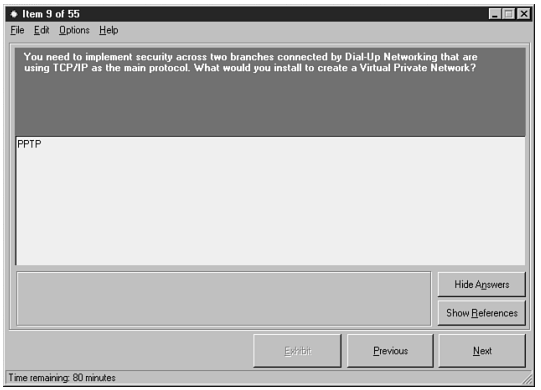


FIGURE D.13▲
The correct answer is shown.

You can also use the Show References button in the same manner as described earlier in the Study Cards sections.

The pull-down menus provide nearly the same functionality as they do in Study Cards, with the exception of Paste on the Edit menu rather than Copy Question.

Flash Cards provide simple feedback. They do not include an Item Review or Score Report. They are intended to provide an alternative way to assess your level of knowledge that will encourage you to learn the information more thoroughly than with other methods.

Using Top Score Simulator

Top Score Simulator is simple to use. Just choose Start, Program and click on the Simulator program name. After the application opens, go to Options, Question Set, and choose one of the three sets of questions. You will be presented with a task or question in the Task window and asked to type in an answer or choose the appropriate tool button to complete the task (see Figure D.14).

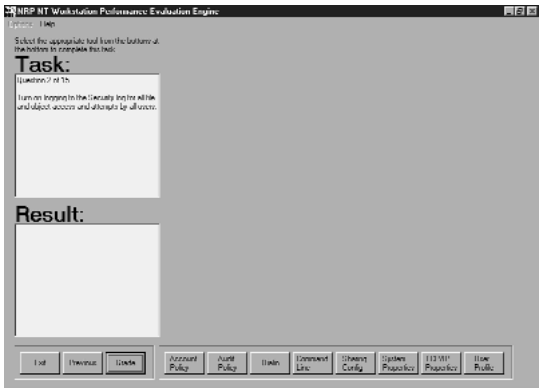


FIGURE D.14▲
An example of a Simulator task.

After choosing the tool, you will have to complete the task by choosing the correct tabs and settings or entering the correct information required by the task (see Figure D.15).

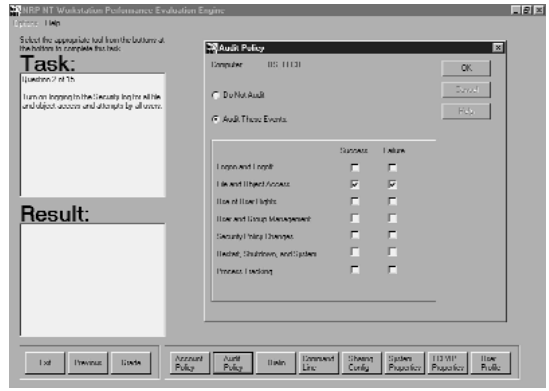


FIGURE D.15▲
Completing the task.

To find out if you chose correctly, click the Grade button. You will receive immediate feedback about your choice in the Result window (see Figure D.16). To move on to the next question, simply click Next. You can use the Previous button to go back over questions you may have missed or wish to review. The Exit button allows you to quit the program.

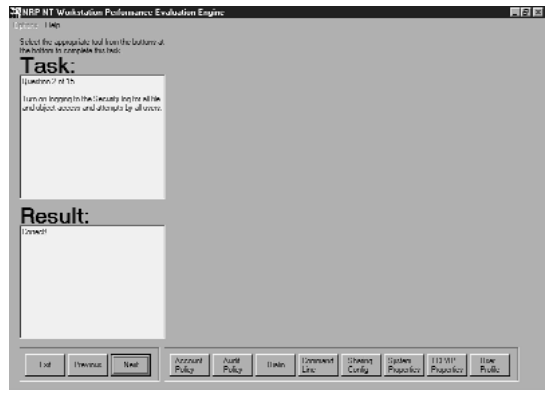


FIGURE D.16▲
You got it correct!

SUMMARY

The Top Score software suite of applications provides you with several approaches toward exam preparation. Use Practice Exams not only to assess your learning but also to prepare yourself for the test-taking situation. The same can be said of the Simulator application. Use Study Cards and Flash Cards as tools for more focused assessment and review and to reinforce the knowledge you are gaining. You will find that these four applications are the perfect way to complete your exam preparation.

APPENDIX E

Visual Basic Basics

The exam objectives are written assuming that you already have a basic working knowledge of the VB language. The objectives therefore do not explicitly mention fundamental VB programming skills.

If you are somewhat new to VB, you may find the discussion of VB fundamentals in this appendix useful as you prepare for the examination.

This appendix covers the following fundamental topics of VB programming:

- Programming with variables
- Programming with Sub and Function procedures
- Using control structures in VB

PROGAMMING WITH VARIABLES IN VB

Variables are storage locations in memory that are used for a wide variety of data. The variable receives a variable name from the programmer, which allows access to the memory location. The programmer does not have to know the physical memory address of information to retrieve and manipulate it. The variable name is used as a reference to the physical address and reduces the work of the programmer.

After information has been stored, the programmer, application, or user can affect the value held within a variable. Various calculations can be performed on the stored value, or the value can be used in other calculations.

For the operating system to allocate the memory required for a variable, the programmer must decide what type of data the variable will hold. Data comes in a wide variety of forms and sizes. Small numbers can take very little memory; however, decimal precision can require a lot of memory. Each type of data has a different physical memory requirement. Each programming language has a set of data types that describe the type of information and how much memory space will be used to hold it. A good programmer will always attempt to use the most effective data type and reduce the memory used by the application.

NOTE

Data Types Discussed at More Length in Later Sections For a fuller discussion of data types, see the subsections under "Understanding Visual Basic's Standard Simple Data Types" later in this appendix.

Declaring and Defining the Scope of a Variable

When the programmer requires a variable, Visual Basic allows two different ways of declaring them. The first method is called *implicit* variable declaration. This method enables a programmer to use a variable any time he or she requires one. The programmer just has to type a new variable name into the source code and the variable is automatically created by VB. The following line of code uses implicit variable declaration:

```
x = ÒHelloÓ
```

In this code sample, the *x* contains the String value ÒHelloÓ . Visual Basic automatically created *x* and selected a default data type for the variable.

The second method of using variables is called *explicit* variable declaration. To use this method, the programmer must first name the variable and describe what type of information will be contained within the variable. The following line of code is an example of explicit variable declaration:

```
Dim x As String
```

In this code sample, before the programmer can use *x* and assign a value to it, the *x* is declared. The Dim statement is used to dimension a variable called x. The variable will hold String information.

Explicit declarations have the following three parts. The first word is a keyword that determines the scope or lifetime of the variable. The second part is the variable name that will be used in the source code. The third part is the data type of the variable that tells the computer how much memory is required to hold the information.

Both methods of variable declaration can be used with Visual Basic. When working with large projects, or to help reduce source code problems, many programmers use explicit variable declaration. It may require a few extra lines of code, but can save many hours of frustration and debugging code. When a typo is made on a variable name, the compiler generates an error before running the code. This is due to a special statement that tells the compiler that all variables must be previously declared before they can be used.

The statement Option Explicit requires that the programmer inform the compiler of a variable's scope, name, and data type before the variable is used. This statement must appear at the beginning of any type of code module in the General Declarations section.

To assist the programmer with explicit declaration, VB has an option that will automatically insert the Option Explicit statement into any new code module that is created. Existing code modules will have to have the statement put in manually. To force variable declaration, ensure that the Tools, Options, Editor Tab check box Require Variable Declaration is checked.

Of the three key components to an explicit declaration, the third part—data type—describes the type of information stored within memory. Visual Basic contains 13 data types. Each data type has a specific name that is used to refer to the information it can hold, as well as to the amount of memory used by that data type.

> **NOTE**
>
> **VB Data Types** Again, for further discussion of data types, see subsequent sections in this appendix under the section titled "Understanding Visual Basic's Standard Simple Data Types."

Each data type has been designed to hold a particular piece of information. This allows for optimized memory use. It also enables the programmer to better control the memory requirements and performance of the application.

Visual Basic has many different data types that allow for storage of simple numbers, complex numbers, Strings , dates and times, as well as objects. One of the more complex data types in VB is the Variant . This is the default data type of Visual Basic. Variant s can be used to store information from any of the other data types. This makes the Variant a very flexible variable, but significantly increases the storage requirements. Listing E.1 gives example declarations of various data types.

LISTING E.1

VARIABLE DECLARATIONS

```
Dim iLoop As Integer
Dim lCounter As Long
Dim strFirstName As String
Dim dblRate As Double
```

The first example declares the variable name of iLoop to store Integer data. The second declares lCounter to store Long numbers. The third variable holds String data and the fourth allocates storage for a Double number. These samples all use a standard naming convention that describes the type of data they contain by using a special letter prefix followed by the name of the variable. This assists anyone who will be reading the source code; the readers will know the type of data being stored without having to read the declarations. All these code samples have also explicitly declared the type of data to be stored.

When variables are implicitly declared, they do not inform the compiler of which data type will be used. With Visual Basic, the compiler automatically assigns implicit variables with the default data type of Variant. This increases the amount of memory that the application will use. To reduce the storage of information in memory, programmers use explicit declarations. This enables the programmer to inform the compiler of the name and data type of the variable to be used. The compiler then verifies that the correct information is passed to variables.

When incorrect information is assigned to a previously declared variable, a runtime error occurs. The type mismatch is an error that new programmers get quickly accustomed to. To avoid these errors, the programmer must closely monitor how the program assigns information to the variables. When information is manipulated, the programmer must explicitly convert the information before assigning it to a different data type variable. This type of data control requires more attention, but ensures that the application is always doing exactly what it was designed to do.

Scope of Variables

Thus far, you have seen that a variable declaration contains the name of the variable as well as the data type. Another important aspect of a variable declaration is the *scope*. The scope of a variable determines where in the application that variable can be referenced. This allows only certain parts of the program to have access to information and provides security for the data.

The four different types of variable scopes are as follows:

- Local
- Static
- Private
- Public

In addition, a fifth variable scope, Global, is considered obsolete, having been replaced by Public. You may still encounter it, however, in code that was created in versions of VB before VB 4.

Defining the scope of a variable requires two things. The first is *where* in the project the variable is declared. The second is the *keyword* used in front of the variable name. Listing E.2 gives some code examples of different scopes in variable declarations.

LISTING E.2

VARIABLE DECLARATIONS OF DIFFERENT SCOPES

```
Dim iLoop As Integer
Static iLoop As Integer
Private iLoop As Integer
Public iLoop As Integer
```

The code samples all declare the same variable name and the same variable type. Notice, however, that the keyword to declare the variable is different for each one.

These four keywords allow the variable being declared to be restricted to certain areas of the application. This gives the programmer tighter control over what information is being assigned to variables. The scope also prevents other procedures from unexpectedly changing the value of the variables.

The keywords Public, Private, Static, and Dim are all used to set the scope. Each has a specific location where it can be used to declare the variable.

Declaring Local Variables

Local variables are declared within a Sub, Function, or Property procedure. The keyword Dim is used in front of the variable name and data type:

```
Dim strFirstName As String
```

Local variables are also referred to as *procedure-level* variables. These variables are created and exist only as long as the procedure is executing. When the procedure has completed, the variable is no longer available and will be reset upon the next execution of the procedure, as illustrated in Listing E.3. If the Sub procedure of the listing were to be called multiple times, the value of *X* displayed in the MsgBox would always be 1, because *X* would always be re-initialized to a default value of 0 and then would get 1 added on to its value.

LISTING E.3

A LOCAL VARIABLE

```
Sub SetX()
    Dim X As Integer
    X = X + 1
    MsgBox ÒThe value of X is Ò & X
End Sub
```

Local variables are only accessible to the code within the procedure. Program code outside the procedure cannot use or set a locally scoped variable. This protects the information stored within that variable. The variable name can be used by other variables and properties, or passed as an argument to other procedures.

The Dim keyword is primarily used within a procedure. VB will allow the use of the Dim keyword in the General Declarations section of a code module. When used outside a procedure, the Dim statement creates a variable with Private scope. The use of the keyword Private is preferred over Dim in the General Declarations section and has only been allowed for compatibility with previous versions of Visual Basic. The keywords Private and Public cannot be used within a procedure.

Declaring Static Variables

Static variables are created within a Sub, Function, or Property procedure. The keyword Static is used before the variable name and the data type:

```
Static X As Integer
```

Static variables are used within the procedure level. The main difference between local variables and Static variables is the life span. Static variables retain their values even after the procedure has finished executing, as illustrated in Listing E.4. If the procedure of the listing were called multiple times during the same session of the application, the value of the variable I would be greater by one on each call.

LISTING E.4

USING A Static VARIABLE TO "REMEMBER" A VALUE BETWEEN CALLS TO A PROCEDURE

```
Sub SetX()
    Static X As Integer
    X = X + 1
    MsgBox ÒThe value of X is Ò & X
End Sub
```

By contrast, local variables are reset every time the procedure executes, as explained in the preceding section, "Declaring Local Variables." Similar to local variables, a static variable is only accessible to the code within the procedure where the variable has been declared. Program code outside the procedure cannot "see" the variable. This restricts outside code from both using and setting the statically scoped variable. The variable can be used by other variables and properties, or passed as an argument to other procedures.

Declaring Private Variables

Private variables can be declared in the General Declarations section of any type of code module. The keyword Private is used in front of the variable name and data type:

```
Private iPage As Integer
```

A privately scoped variable is only accessible to procedures contained within the same code module. If a variable were declared in the General Declarations section of a form code module, only procedures from that form would have access to the variable. Procedures from other form code modules or standard modules would not be able to access the privately scoped variable.

The code sample in Listing E.5 assumes the previous declaration sample has been made in the General Declarations section of the same form code module.

LISTING E.5

USING A PRIVATELY SCOPED VARIABLE IN AN EVENT PROCEDURE

```
Sub Command1_Click()
    iPage = iPage + 1
    MsgBox ÒThere are Ò & iPage & Ò pages
currently.Ó, vbInformation
End Sub
```

The value of the variable is used and set within the Command1 Click procedure. The variable retains its value until the form is unloaded.

When using standard or class modules, variables can also be privately scoped by using a declaration for the variable in the General Declarations section. The privately scoped variable will only be accessible to procedures from the same standard or class module.

Declaring Public Variables

Public variables are created in the General Declarations section of any type of code module. The keyword Public is used in front of the variable name and data type:

```
Public iPage As Integer
```

Public variables are available to all components within the application. The variable is retained for the lifetime of the application.

Publicly scoped variables can be used and set by any code component whether it's a form, standard, or class module.

To reference a publicly scoped variable declared on a form code module, the name of the form must be referenced first, followed by the variable name.

When using standard modules, the name does not have to be referenced unless there are multiple modules with the same publicly scoped variable name. Then the module name must be referenced first, followed by the variable name.

Classes must be instantiated before publicly scoped variables of the class can be accessed.

Using the Appropriate Declaration Statement

To protect data that has been assigned to a variable, the programmer can use the scope of a variable to determine where in the application the variable can be accessed. This allows only required parts of the program to be able to change a variable.

One of the most important reasons for using the correct variable scope is to prevent logical errors in the application. A logical error occurs when the program is functioning, but not with the desired results. Logical errors can often be attributed to the programmer expecting a variable to have a certain value when it does not. This "unexpected" value could be caused from a certain procedure, within the application, changing the variable. Limiting which parts of the application can have access to certain variables can control these logical errors.

This appendix has already discussed the requirements for declaring a variable with certain scope. The following section expands on that information by showing specific uses of Local, Static, Private, and Public variables.

Using Local Variables

The keyword Dim is used within a procedure to declare a locally scoped variable.

> **NOTE**
>
> **A Limitation on Implicit Variable Declaration** When an implicit variable declaration is made, the variable will be a Variant data type and have local scope. This is another limitation of implicit variable declaration.

Local variables should be used when the value they contain will no longer be required after the procedure has completed execution. The Local variable will be reset when the procedure is executed again, and will thus lose its value.

Some examples of local variable use would be for looping counters, status flags, calculations, and property values, as illustrated in Listing E.6.

LISTING E.6

A Local Variable Used as a Loop Counter

```
Sub Command1_Click()
    Dim iLoop As Integer
    For iLoop = 1 To 100
        Print iLoop
    Next iLoop
End Sub
```

When the code in Listing E.6 executes, the iLoop variable is used to increment the For loop and then print the value to the default Form object. The variable cannot be altered from outside the Command1 Click procedure due to the local scope.

The code sample in Listing E.7 demonstrates assigning the value from a property to a locally scoped variable. Then the variable is used to manipulate the information.

LISTING E.7

Using Local Variables

```
Function CountFullName()As Integer
Dim strFirstName As String, strLastName As åString
    Dim iLenFirstName As Integer, iLenLastName As
➥Integer
    strFirstName = frmMain.txtFirstName.Text
    strLastName = frmMain.txtLastName.Text
    iLenFirstName = Len(strFirstName)
    iLenLastName = Len(strLastName)
    frmMain.txtFullName.Text = strFirstName & Ò Ò
➥& strLastName
    CountFullName = iLenFirstName + iLenLastName
➥End Function
```

This reduces the time needed to search for the object reference. The length of the first and last name are measured, then added together, and then used as the return value of the function.

Using Static Variables

The keyword Static is used within a procedure to declare a statically scoped variable.

Static variables should be used when the value they contain is required to remain after the procedure has executed, but the value is not to be altered outside the procedure.

The code in Listing E.8 compares the behavior of Static and Local variables.

LISTING E.8

COMPARISON OF LOCAL AND STATIC VARIABLE BEHAVIOR

```
Sub Command1_Click()
    Dim iLoop1 As Integer
    Static iLoop2 As Integer
    iLoop1 = iLoop1 + 1
    iLoop2 = iLoop2 + 1
    MsgBox ÒThe Þrst value is Ò & iLoop1 & vbCrLf
➥& ÒThe second value is Ò & iLoop2, vbInformation
➥End Sub
```

This code declares two variables with different scope. The first variable has local scope and the second has static. When the Command1button is clicked, a message box will appear with two lines of text. The first line of text is the Local variable, and the second line of text is the Static variable. Which one will continue to increase? The Static variable will, because it will retain the value after the procedure has completed execution and is called again.

Static variables can be used to set persistent status information within a procedure, but is not needed outside the procedure. This can allow the procedure to indicate how many times it has been executed, if processing is in progress, or other status information.

Using Private Variables

To declare a Private variable requires the keyword Private, and the declaration must be located in the General Declarations section of the code module.

> **NOTE**
>
> Dim **Can Also Be Used in General Declarations** The keyword Dim can also be used in the General Declarations section of the code module. This will also create a privately scoped variable. The preferred method is to use the keyword Private, but Dim has been included for backward compatibility with code originally written in earlier versions of VB. Private cannot be used to declare variables in VB procedures, although it can be used in a General Declarations section

Private variables should be used when you need to be able to access a variable's value throughout the entire code component. Private variables can be used in form code modules, standard code modules, and classes, including specialized classes such as UserControls (ActiveX controls) or UserDocuments (Active documents). All procedures of the code module will have access to the Private variable.

Private variables require more attention from the programmer to ensure that the value can be changed at any time without a specific value being required by a procedure.

Listing E.9 uses a form-level Private variable declared in the General Declarations section.

LISTING E.9

USING Private VARIABLES IN EVENT PROCEDURES

```
Private iPage As Integer
Sub Command1_Click()
    iPage = iPage + 1
    MsgBox ÒThe current page count is Ò & iPage,
➥vbInformation
End Sub
Sub Command2_Click()
    iPage = iPage – 1
    MsgBox ÒThe current page count is Ò & iPage,
➥vbInformation
End Sub
```

Two command buttons are placed on the form and coded as shown:

Command1 Click increases the value of the variable by one, and then displays a message box indicating the current page count.

Command2 Click decreases the value of the variable by one, and also displays a message box with the current count.

Both procedures can manipulate the privately scoped variable. If other procedures required the information contained within the variable, they could also use it.

Using Public Variables

To declare a Public variable requires the keyword Public, and the declaration must be located in the General Declarations section of the code module.

Public variables should be used when a value is required in all parts of the application for the entire lifetime of the application. Public variables require a good deal of consideration by the programmer because any procedure in the application has access to that variable.

When a publicly scoped variable is declared on a form code module, other forms can reference the variable by indicating the form name followed by the variable name. The Public variable then looks very much like a custom property of the Form object.

The code in Listing E.10 uses two different Form objects. Form1 contains a single command button called cmdForm1. Form2 contains a single command button called cmdForm2. Both command buttons have access to the publicly scoped variable of Form1.

LISTING E.10

ACCESSING A PUBLIC VARIABLE FROM TWO DIFFERENT FORMS

```
Public strAppTitle As String
Sub Form_Load()
    strAppTitle = ÒBig Sample AppÓ
    Form2.Show
End Sub
Sub cmdForm1_Click()
    MsgBox ÒThis application is called Ò &
➥strAppTitle
End Sub
Sub cmdForm2_Click()
    MsgBox ÒThis application is called Ò &
➥Form1.strAppTitle
End Sub
```

Notice how Form2's command button must use the name of the form first, and then the variable name. Form1 does not require the same qualification, but it is commonly used anyway.

If Form2 does not qualify the variable name with the form name in front, a compiler error will occur (Variable not deÞned).

When publicly scoped variables are used with standard code modules, the module name does not have to be included to reference the variable. The exception to this rule is if more than one code module is using the same publicly scoped variable name. In this case, the reference must indicate which module is to be referenced.

Too many publicly scoped variables can cause logical errors to occur in the application. Most applications will use very few Public variables if possible. Another method of passing information from one procedure to another is to use arguments. This is another alternative to using too many Public variables and offers better protection of the information.

Understanding Visual Basic's Standard Simple Data Types

A simple data type stores just one basic type of information at one place in your computer's memory. Other data types, which are discussed later, aren't so easy to characterize. They usually represent more complex data divided into a number of areas with special functions.

To be able to select a data type properly, you should know the following:

- What kind of information each data type can hold
- How much memory each data type takes up
- What range of data a data type can hold

Numeric Data Types

Programmers who haven't used languages supporting multiple numeric types may wonder why they need more than one type to store numeric values. The answer is efficiency in storage and access.

Two kinds of data type (Integer and Long) hold whole-number information, and two more types (Single and Double) hold floating-point numbers—that is, numbers that store a movable decimal point. There are two of each type to accommodate different storage needs. You would use the larger of each pair (Long for whole numbers, and Double for floating point) when you had to and the smaller of each pair (Integer and Single) whenever possible. The following sections on each data type give specific information about when to use the various numeric data types.

NOTE

Numbers as Numeric or Text Data
Don't store Social Security numbers, phone numbers, or customer ID numbers as numeric data. These are really textual information masquerading as numbers. A good general rule is this: If you don't plan to do math with it, it isn't really a number.

Integer

Integer data have the following characteristics:

Type of information	Whole number
Amount of memory required	2 bytes
Range of data	–32,768 to 32,767 (64K total)

Integer is a good choice for most incrementing counters (unless you plan to increment beyond the 32K limit) and for a lot of "every day" kinds of information. Examples would include things such as the following:

- Age to the nearest year (for human beings, at least)
- Number of the day or month
- Number of tax-deductible dependents (let's hope it is not more than 32K)

Long

Long data have the following characteristics:

Type of information	Whole number
Amount of memory required	4 bytes
Range of data	–2,147,483,648 to 2,147,483,647

Longs are appropriate for whole-number information that is too big to fit in an Integer (outside the 32K range, in other words). Examples of good Long data type candidates include the following:

‡ Salary in whole dollars

‡ Population of a city

Single

Single data have the following characteristics:

Type of information	Floating-point (fractional) number
Amount of memory required	4 bytes
Range of data	–3.402823E38 to –1.401298E-45 and 1.401298E-45 to 3.402823E38

Just about any non-whole number can fit in a Single . After all, 10^{-45} is a pretty small number and 10^{38} is pretty big. No business application will probably ever require anything bigger than a Single .

Double

Double data have the following characteristics:

Type of information	floating-point number
Amount of memory required	8 bytes
Range of data	–1.79769313486231E308 to –4.94065645841247E-324 and 4.94065645841247E-324 to 1.79769313486232E308

When to use: when you need to store astronomically large or small values. Scientific and engineering applications might require you to use Double . Examples of values requiring Double include the following:

‡ Number of elementary particles in the Universe

‡ Number of seconds (estimated) since the Universe began

‡ Size of a quark (a subatomic particle), in millimeters

The most critical difference between Single and Double is really in the number of significant digits (7 versus 15), which can be critical in almost any type of application, including financial calculations.

Currency

Currency data have the following characteristics:

Type of information	Number with four fixed decimal places
Amount of memory required	8 bytes
Range of data	–922337203685477.5808 to 922337203685477.5807

WARNING

Passing Currency Data Type Outside of VB Most applications outside of Visual Basic (such as C routines in DLL calls) don't recognize Currency , and so you must convert a Currency -type variable to Single or Double before you pass it.

When to use: Currency is best used with applications in which you need to keep track of money. Unlike the Single data type, Currency eliminates small rounding errors that can, in some situations, creep into calculations with Single and Double . Accountants will thank you for using Currency .

String **Data Type**

String data have the following characteristics:

Type of information	Textual information
Amount of memory required	10 bytes plus 1 byte for each character if the String is a variable-length String . 1 byte for each character if the String is fixed length
Range of data	0 to approximately 2 billion characters for Win95 and NT, and 0 to approximately 65,400 for earlier versions of Windows

A String can have fixed or variable length. You declare a variable-length String as you would any other variable type:

```
Dim MyString As String
```

You declare a fixed-length String by specifying the number of characters the String will hold:

```
Dim MyString As String * 20
```

If a String has variable length, you can add or delete characters, or re-initialize it at runtime as shown in the three examples of Listing E.11.

LISTING E.11

MANIPULATING A VARIABLE-LENGTH STRING

```
MyString = MyString & Ò.Ó
MyString = Left(MyString,3)
MyString = ÒÓ
```

In each of the cases in the listing, the String's size changes to reflect the new number of characters in the String.

If a String is of fixed length, you can't change its size. Any statement that appears to change the size of the String still leaves the String with the same number of bytes. In Listing E.12, the Debug.Print statement will always display a value of 11 for the String's length, because the String has fixed length.

LISTING E.12

MANIPULATING A FIXED-LENGTH STRING

```
Dim str1 As String * 11
str1 = ÒHello WorldÓ
Debug.Print Len(str1)
str1 = Left(str1, 3)
Debug.Print Len(str1)
```

WARNING

Fixed-Length String s as Public Variables You can't declare a fixed-length String as a Public variable in a form module. If you try to do so, you will get a compiler error.

You should, of course, store any type of data that contains variable textual information in a String. It is important to notice that it was termed "variable textual information." If a certain piece of information can always be represented by three values ("Married," "Single," "Divorced") for instance, perhaps that information should be stored as an Integer with values from 0 through 2 or 1 through 3 (1=Married, 2=Single, 3=Divorced). If the information is always one of two values ("Male," "Female"), consider implementing such dual-valued information as a Boolean data type (call the variable blnIsFemale, give it a True value if dealing with a female and a False value if dealing with a male). See "Boolean Data Type" in this appendix. Much information that appears to be numeric is actually better off as a String. This is because String s are the most pliable type of data and can be manipulated in numerous ways. In the example

```
MyString = Ò312-555-3245Ó
NewString = Mid(MyString,5,3)
```

NewString would end up holding the value Ò555Ó. The example shows that it is trivial to extract a certain range of characters from a String, but try doing that with a numeric data type sometime.

NOTE

Numbers Stored as String s In general, if you don't plan to perform mathematical calculations with a number, store the number as a String.

Date Data Type

Date data have the following characteristics:

Type of information	Date and time information—stored as floating point, where the fractional part represents time, and the whole number part represents date
Amount of memory required	8 bytes
Range of data	January 1, 100 to December 31, 9999, including fractional days (time of day)

Use it when you need to store or manipulate a date or time.

> **NOTE**
> Date **Data and Arithmetic Operations** You can do arithmetic operations, such as addition and subtraction, with variables stored as Dates . The DateDiff() function returns the amount of time elapsed between two Date -type variables.

It is easy to determine *when* to use a Date type. Now let's discuss *how* to use a Date -type variable.

Using the Now Keyword to Get Current System Date and Time

You can store the current date and time in a Date -type variable using the Now system function:

```
datToday = Now
```

You can use the value of datToday with the Format() function to extract readable String information about the date, day of the week, or time.

In addition, the Date function will return the current date only (no time component).

> **NOTE**
> **System Date and Time** You can set the computer's system date and time from a Visual Basic program with the Date and Time statements.

Using Special Date / Time Functions to Extract Information from a Date Variable

If you need to extract components of a date (year, month, day of the month, hour, minute, or second), use a separate function for each of these pieces of information.

For more information on using these functions, see the VB Language Reference and the Programmers Reference Guide for the following functions: Year() , Month() , Day() , Hour() , Minute() , and Second() . Note also the Timer() function, which returns the number of seconds elapsed since midnight.

Using Special Date / Time Functions to Store Information in a Date Variable

What if you need to store information in a Date -type variable other than the current date-time stamp of your system?

The DateSerial() function is one way to directly store date information in a Date -type variable. It takes three arguments: a year, a month, and a day. (All three values must work together to produce a valid date. 95,2,29 would not be valid, for instance, because there was no February 29, 1995.)

You can directly store time information in a Date -type variable with the TimeSerial() function. You can add the results of the DateSerial() and TimeSerial() functions together to store date/time information in a single variable.

LISTING E.13

DATE AND TIME MANIPULATION

```
Dim MyDate As Date
Dim intYear As Integer
Dim intMonth As Integer
Dim intDay As Integer
Dim intHour As Integer
Dim intMinute As Integer
intYear = CInt(txtYear.Text)
intMonth = CInt(txtMonth.Text)
intDay = CInt(txtDay.Text)
intHour = CInt(txtHour.Text)
intMinute = CInt(txtMinute.Text)
MyDate = DateSerial(intYear, intMonth, intDay) + _
TimeSerial(intHour, intMinute, 0)
Debug.Print MyDate
```

This example in Listing E.13 assumes that the user has typed desired values for year, month, day, hour, and minute into text boxes. The program converts these values to integers using the CInt function and stores them in Integer variables. Then, these Integer variables are used as arguments to DateSerial() and TimeSerial(). The program sums the return values of these two functions and stores the sum as a Date -type variable. Of course, a more robust application would validate the contents of the text boxes before trying to use them.

Byte **Data Type**

Byte data have the following characteristics:

Type of information	A single byte of data
Amount of memory required	1 byte
Range of data	0 to 255 (positive values only)

Byte is most useful when you are exchanging information that is in some binary format between Visual Basic and another environment, such as when your program calls a DLL routine or when it directly accesses a file containing binary information.

NOTE

Dealing with C Routines in DLL Files When a C routine in a DLL file needs C's char type in an argument, pass a variable of type Byte .

WARNING

Byte **Data Limitations** When you convert a variable to type Byte , make sure that the variable you are converting represents a number between 0 and 255 (positive values only). Any other value will cause an Overßow runtime error.

Boolean **Data Type**

Boolean data has the following characteristics:

Type of information	One of two values—True or False
Amount of memory required	2 bytes
Range of data	True or False

Whenever a piece of information will always take on two, and only two, values, and these two values are logically opposed to each other, use the Boolean data type. You might use Boolean to show the following, for example:

- Whether an employee is a 401K participant
- Whether data has changed

It is not a good idea to use a Boolean to represent the following:

- Employee type, when there are more than two types
- Marital status, if you need more than two possible choices (such as Married, Single, Divorced, and Widowed)

In such cases, you would want to use either an Integer to represent the different possibilities, or perhaps a String containing a description of the particular option.

> **NOTE**
>
> Boolean **Conversion** When converting from a numeric type to a Boolean, all nonzero values of the number will convert to Boolean True, and 0 will convert to False.
>
> When converting from a Boolean to a numeric type, True will convert to Ð1, and False will convert to 0.

> **WARNING**
>
> String **to** Boolean **Conversion** To successfully convert a String to a Boolean, the String can only contain ÒTrueÓ or ÒFalseÓ. Note that the conversion is not case sensitive, so ÒTRUEÓÒtrueÓ, and any other capitalization would work. Any other value in the String will cause a runtime error.

Variant **Data Type and the** Decimal **Subtype**

Variant data has the following characteristics:

Type of information	Any type of data
Amount of memory required	The amount of memory required to implement whatever type of data has been stored in the Variant, plus another 22 bytes of overhead
Range of data	Depends on the data type of its contents

Use this data type as little as possible. A Variant has a negative effect on memory resources and speed/performance issues; Visual Basic must perform internal conversions every time your program accesses a Variant variable.

You will need to use Variant when

- You are writing code for a routine that can take a parameter of ambiguous type (usually declared as Variant).
- You are using a For Each...Next loop to process elements of an array. The placeholder variable in such loops must be a Variant. (This is only true for *arrays*: When processing a *collection* with For Each...Next, you can use a placeholder whose type is compatible with the type of object in the collection.)
- You want to implement a ParamArray, which is an array of Variant. See "Array Arguments and the ParamArray Keyword" later in this appendix.

The Variant data type has a subtype known as Decimal. It is called a subtype because only a Variant can actually store Decimal-type data—you can't directly declare a variable as Decimal. Decimal is useful because it reduces rounding errors that can creep into numeric results, especially after division. If you run the code in Listing E.14, you will see that the result of a division of two numbers such as one and three correctly displays when displayed as Variant Decimal, but that the decimal places begin to wander when you use Single.

LISTING E.14

DIFFERENCES IN ACCURACY BETWEEN THE VARIANT'S DECIMAL SUBTYPE AND THE SINGLE TYPE

```
Private Sub Command1_Click()
    Dim varMe As Variant
    Dim sngSource As Single
    Dim Above As Long
```

```
    Dim Below As Long
    If IsNumeric(txtAbove.Text) And IsNumeric
➥(txtBelow.Text) Then
        Above = txtAbove.Text
        Below = txtBelow.Text
        sngSource = Above / Below
        varMe = CDec(Above / Below)
        MsgBox Above & Ò/Ó & Below & Ò  =Ó _
            & Chr$(10) _
            & Chr$(10) & ÒSingle:Ó _
            & Chr$(10) & Format(sngSource,
➥Ò##.################Ó) _
            & Chr$(10) _
            & Chr$(10) & ÒVariant Decimal:Ó _
            & Chr$(10) _
            & Format(varMe, Ò##.################Ó)
    Else
        Beep
        MsgBox ÒPlease enter numbers in both text
➥Þelds
    End If
End Sub
```

NOTE

The CDec Conversion Function The CDec conversion function is discussed in the section "CDec" in this appendix.

Checking the Data Type of a Variable

Although good programming practice encourages you to keep track of variable types, at times it is legitimately possible to be ignorant of a variable's type. This can happen in the following situations:

- You are writing code for a routine that can take a parameter of ambiguous type (usually declared as Variant).
- You are looking at object variables. You may know a variable is an object, but you need to know what kind of object it is. This could happen, for instance, when you are looking at a form's Controls collection (see the section in this appendix on collections).

IsNumeric() , IsArray() , and IsDate() Functions

The IsArray() function takes the name of a variable as its single parameter and returns a Boolean . It will return True if the variable is an array. For example, in the two lines

```
Dim MyNames() as String
If IsArray(MyNames) Then...
```

IsArray will evaluate to a True value.

IsNumeric() and IsDate() can actually do more than tell us whether a particular variable is one of the numeric types or whether the variable is of type Date . These two functions will also accept a String and determine whether Visual Basic can interpret the contents of the String as a number or a date. For example, in the code fragment

```
strIsIt = Ò897.90Ó
If IsNumeric(strIsIt) Then...
```

IsNumeric would evaluate to a True . You could then perhaps use the CDbl function (mentioned in the following section on the "C" functions) to convert the contents of the strIsIt variable in numeric computations. If you called CDbl without first making sure that strIsIt could be converted to a number, you could generate a runtime error.

TypeOf Statement and TypeName() Function

TypeName() provides a general way to determine the data type of a nonobject variable. TypeName() returns a String giving the data type's name. In the example of Listing E.15, you will see the word *Integer* in the Msgbox.

LISTING E.15

USING TYPENAME TO DETERMINE A VARIABLE'S DATA TYPE

```
Dim MyVar  Ôno explicit data type deÞned
                Ô- so this is a Variant
MyVar = 1
MsgBox TypeName(MyVar)
```

TypeOf will tell you the specific type of an object. It has the following special syntax:

```
If TypeOf MyControl Is TextBox then...
```

Because of its syntactic format, you can only use TypeOf in logical expressions.

Converting Between Data Types

Very often, you need to use different data types together in the same expression.

Visual Basic can often perform data-type conversion automatically, and you, as the programmer, need not concern yourself with what goes on behind the scenes in these cases.

At other times, however, you will need to convert a variable to another data type before you use it. This is necessary when you want to pass a variable as an argument to a function or procedure, and the procedure is expecting a parameter of a different data type.

> **NOTE**
> **Argument-List Compatibility** When you call a Function or Sub procedure with arguments, the data types of the arguments must match the data types in the parameter list.

Automatic Type Conversion

Visual Basic is very versatile in converting between data types, especially between String and other types. In many cases, you can just let Visual Basic do an internal conversion for you in numeric and String expressions without having to use any special conversion functions or other precautions.

Automatic Conversion Between Numeric Values

When you are dealing with different numeric data types, there is often no problem because Visual Basic automatically does the necessary conversion between types, including any rounding.

In Listing E.16, the actual result of the calculation is 87.8, but Visual Basic gracefully assigns a value of 88 to the Integer variable.

LISTING E.16

VB AUTOMATICALLY ROUNDS THE RESULT OF THIS CALCULATION TO AN INTEGER

```
Dim intFahrenheit as Integer
Dim dblCelsius As Double
dblCelsius = 31
intFahrenheit = (dblCelsius * 9 / 5) + 32
```

> **NOTE**
> **Beware of Incompatible Variable Sizes** You must also be wary of trying to stuff a value into a container (that is, a variable type) that isn't big enough to hold it. Always check the value you are trying to convert to make sure it fits the range allowed in the target data type.

Although you can assign a larger data type, such as Double, to a smaller data type, such as Integer, you must make sure the Double variable doesn't contain a number outside of the range –32K to 32K, which are an IntegerÕs limits. Otherwise, you are asking for a runtime Overßow error.

In general, you should always check the value of the larger of two types when converting between two numeric types (including Date and Byte).

Automatic Conversion of String Expressions to Numeric Values

If a String variable or control property of type String holds a String that can be evaluated as a number, it can be used as an element in a numeric expression, as in Listing E.17.

LISTING E.17

USING A STRING IN A NUMERIC EXPRESSION

```
Dim strRate as String
strRate = Ò11.50Ó
If IsNumeric(txtHours.Text) then
    intEarnings = strRate * txtHours.text
Else
    MsgBox ÒEnter a valid number in the Hours
➥ÞeldÓ
End If
```

Notice that in this example you need to check to see if the TextBox control's Text property holds a valid number. If you had performed the calculation without checking and txtHours.Text didn't hold a valid numeric expression, Visual Basic could have generated a runtime error.

As with numeric-to-numeric type conversions, you must be careful that a numeric String doesn't represent a value larger than the capacity of the type you want to convert to.

Automatic Conversion of Numeric Expressions to Strings

Visual Basic will also automatically convert any other data type into a String whenever non-String variables are used in a String assignment or other expression requiring a String, as in Listing E.18.

LISTING E.18

USING ANOTHER DATA TYPE IN A STRING EXPRESSION

```
Dim dblEarnings as double
txtEarnings.Text = dblEarnings
MsgBox dblEarnings
```

The Text property of a TextBox holds String-type data, and the MsgBox normally takes a String argument. In the example, VB converts a Double to a String both when assigning a Double to a TextBox control's Text property and when passing a Double to the MsgBox statement as its Prompt argument.

Conversion of Non-string Arguments in String Concatenation

In most programming languages, including Visual Basic, the + operator can add two numbers together and it can also concatenate two String expressions, as in Listing E.19.

LISTING E.19

THE + OPERATOR ONLY WORKS WHEN CONCATENATING TRUE STRINGS

```
strMine = ÒMineÓ
strYours = ÒYoursÓ
strOurs = strMine + Ò and Ò + strYours
```

Visual Basic also uses the & symbol for String concatenation.

> **NOTE**
>
> **Ampersand and String Concatenation**
> The & operator automatically converts its operands into Strings before using them.
>
> Microsoft recommends that you use the & operator rather than the + for String concatenation; & is more versatile.

The expression

```
strOurs = StrMine + Ò and Ò + 0
```

is illegal, because the + operator can take only String s as its arguments.

However, the expression

```
strOurs = StrMine & Ò and Ò & 0
```

is legal, because the & operator is more intelligent and converts its non String arguments to Strings . In this example, strOurs would end up holding the String ÒMine and 0Ó .

Asc

The Asc() function takes a String argument and returns an Integer -type value representing the ASCII code of the String 's first character. No matter how long the String argument is, Asc() only pays attention to the first character in the String .

In the following example, the MessageBox will display a 65, which is the ASCII value of a capital *A*, the first letter in the String :

```
Dim strName As String
strName = ÒAliceÓ
MsgBox Asc(strName)
```

The Asc function ignores the other letters in the name.

AscB

The AscB() behaves just like Asc() , except that its return value is of the Byte type. AscB() would be useful, for instance, when an external routine such as a DLL call needs a single-byte parameter referring to a character. Usually, such routines will have been written in C and will require a variable of the char data type, which is specific to C.

The Chr and ChrB Functions

Chr() and ChrB() take an Integer data type as their argument and return a one-character String corresponding to the ASCII code of the argument.

> **WARNING**
>
> **Range Restrictions with Chr() and ChrB()** You must remember to pass only Byte values or Integers in the range 0–255 as the argument to Chr() and ChrB() . If you pass a number outside this range, Visual Basic will generate an Overßow error.

The "C" (C for Convert) Functions

Use these functions when you want to be sure that Visual Basic will convert some variable or other expression to a specific data type. Each of the "C" functions work in the following format:

```
NewValue = CFunction(any expression)
```

where NewValue has the data type to which CFunction converts.

CBool

CBool will convert any non-zero numeric expression to True and will convert any expression containing zero to False , as in Listing E.20.

LISTING E.20

CONVERTING NUMERIC VALUES TO BOOLEAN

```
Dim blnHasProblem As Boolean
Dim intProblemCount As Integer
intProblemCount = 30
blnHasProblem = CBool(intProblemCount)
```

CBool will also convert String s to Boolean s, as long as the String s hold ÒTrueÓ or ÒFalseÓ or any alternative capitalization of those words. Otherwise, it will generate a runtime error.

CByte

CByte will convert any valid numeric String expression or any numeric type to a byte, as long as the range of the number or numeric expression is 0–255. If you supply a number outside of this range, you will get an Overßow runtime error from Visual Basic. If you supply a String that is not numeric, you will get a Data Type Mismatch error.

LISTING E.21

EXAMPLES OF CORRECT AND INCORRECT USAGE OF CBYTE

```
Dim bytWing As Byte
bytWing = CByte(200)  ÔOK: argument is number
➥1Ð255
bytWing = CByte(Ò100Ó) ÔOK: string is numeric
➥1Ð255
bytWing = CByte(256)  ÔERROR: number is > 255
bytWing = CByte(Ð200) ÔERROR: number is < 0
bytWing = CByte(ÒASDÓ) ÔERROR: string doesnÕt
➥evaluate
                                  Ôto a number
```

Listing E.21 gives examples of successful and unsuccessful attempts to use CByte.

CCur, CDbl, CInt, CLng, and CSng

Each of these functions will take any other numeric type (including Date and Byte) or any numeric String expression and convert it to Currency, Double, Integer, Long, or Single, respectively.

CStr

CStr() will convert any argument to a String. You don't have to worry about something "not fitting" into the return value, because a String can be very long.

> **WARNING**
>
> **Converting from "Larger" Data Types to "Smaller" Ones** You must be careful when doing a numeric conversion from a larger data type (one that has a larger size in memory) to a smaller one. If you are using CInt or CSng, be especially careful to check that the value you are passing to the function doesn't exceed the bounds for an Integer or Single variable.

CVar

Like CStr(), CVar() will take any argument and convert it to a Variant.

CDec

CDec returns a Variant of subtype Decimal, so you use CDec to assign a value to a Variant data type. You can find an example of the use of CDec in the section of this appendix titled "Variant Data Type and the Decimal Subtype."

CVErr

CVErr() takes a number representing any valid VB error number as its argument. CVErr() returns a Variant, as in these two lines:

```
Dim MyVar As Variant
MyVar = CVErr(200)
```

The Format Function

The Format function is very useful for putting the finishing cosmetic touches on an application, because it takes any expression and converts it to a formatted String. The Format function's second argument is a String representing a formatting template, which instructs the function how to display the String. Listing E.22 provides a few examples.

LISTING E.22

SOME EXAMPLES OF THE FORMAT FUNCTION

```
datToday = Now
dblMucho = 1000000.003
Msgbox Format(datToday,Óhh:mm:ssÓ)
MsgBox Format(dblMucho,Ó###,###,###.####Ó)
MsgBox Format(dblMucho,Ó000,000,000.0000Ó)
```

The first MessageBox in the listing would display a time such as 17:27:53.

The second MessageBox would display 1,000,000.003.

The third MessageBox would display 001,000,000.0030.

There are numerous ways to specify formatting Strings to the Format function, including many Visual Basic constants for built-in types of formatting. Check Visual Basic's online help or the Visual Basic 6 Language Reference for more details.

Common String -Manipulation Functions

As previously mentioned, the String is one of the programmer's favorite data types because it is so easy to chop, slice, dice, and splice Strings.

The following sections discuss some of the workhorse functions used to manipulate String expressions.

Left

The Left function takes two parameters—a String expression to parse, and a number representing how many of the leftmost characters you want returned from the String.

```
strMyName = ÒBill M. SmithÓ
strFirst = Left(strMyName,4)
```

After this code runs, the value of strFirst is ÒBillÓ.

Right

The Right function works like the Left function, but instead of giving the leftmost characters, it returns the rightmost characters:

```
strMyName = ÒBill M. SmithÓ
strLast = Right(strMyName,5)
```

After this code runs, the value of strLast is ÒSmithÓ.

Mid

Use the Mid function in those ticklish situations where you need to pick out something from neither the right nor the left side of a String, but from the middle. Mid takes three arguments:

- The String to examine
- The position of the starting character from the left (1-based)
- The number of characters to parse (starting with the character position specified in the second argument)

Suppose you know that the ProductID field in a database table represents specific information about the product. In the following example, the product color is stored in strProductID. The characters at positions 5–7 represent the product color. You can parse out the color code by passing three arguments to the MID function. The variable strProductColor in the example contains the String ÒBLUÓin positions 5–7. The second argument, 5, represents the first character position of the expression you want to parse. The third argument represents the number of characters you want to parse from the starting position.

```
strProductID = Ò423-BLU-099Ó
strProductColor= Mid(strProductID, 5, 3)
```

The Mid function is even more versatile than that, however. It not only dices—it slices! And you can accomplish this amazing feat with the Mid function just by leaving off the third parameter. If you don't tell Mid how many characters you want after the designated position, it returns all characters through the end of the String.

The following example will return everything in the String from the ninth position through the end of the String :

```
strMyName=ÓJohn J. ThomasÓ
strLast = Mid(strMyName,9)
```

strLast will return ÒThomasÓ because the T in Thomas is the ninth character in the String and therefore Mid will return everything from the ninth character on.

Instr

The Instr function doesn't return a String as do the other String -manipulation functions. Instead, it returns an Integer pointing to a position in a String where a character pattern was found. You can therefore use Instr to find things out about Strings and then manipulate them with this information.

Instr takes two required parameters, both Strings :

- The String to examine
- The String expression for which to search

Listing E.23 checks for a space in the String variable strName . If Instr returns a nonzero value representing the starting character position of the String expression for which you are searching, you parse all characters to the left of the space as a first name.

LISTING E.23

USING INSTR TO HELP PARSE A STRING

```
strName=ÓJohn ThomasÓ
intSpacePos = Instr(strName ,Ó Ò) Ôlook for a
➥space
If intSpacePos <> 0 Then        ÔThereÕs a
space
➥       Ôand everything to its left is the Þrst name
      strFirst = Left(strName, intSpacePos-1)
End If
```

Instr takes an optional first argument, which is an Integer representing the position from which you want Instr to start scanning the target String in its quest for a match. Therefore,

```
Instr(9,MyString,Ó Ò)
```

will only tell you about spaces it finds from the ninth character on in MyString . The positional number it returns, however, will be with respect to the entire original String . For example, the lines

```
strMyString = ÒABCDEÓ
intPos = Instr(3,strMyString,ÓDÓ)
```

will return the value 4 to intPos .

Len

The Len function (see Listing E.24) tells you how many characters a String contains.

LISTING E.24

USING THE LEN FUNCTION

```
If Len(Trim(txtResponse.Text)) = 0 Then
      MsgBox ÒYou must enter a responseÓ
      txtResponse.SetFocus
End If
```

> **NOTE**
> **Len Function and Data Types** You can use the Len function with variables of any data type, not just Strings . When used with these other types of variables, Len will return the amount of memory the variable uses.

A Parsing Example Using String -Manipulation Functions

Let's see how you could use String -manipulation functions to parse a String out into individual elements, placing those elements in an array for later use.

Listing E.25 shows a function called ParseOut and Listing E.26 shows a call to that function from elsewhere in code.

ParseOut takes three arguments: the String to be parsed (strSource), the character to use as a parsing character (strParseChar), and the array to hold the parsed results (strTarget), one parsed element per array element.

ParseOut returns an Integer giving the number of elements parsed.

The basic strategy of the function is to traverse the original String looking for the parsing character. Every time you find the parsing character, take everything to its left and put it into a new array element. Then chop the String down to just what remains after the parsing character and repeat the operation on what remains.

When nothing is left, you know you are done. See the comment on each line of code in Listing E.25 for more detail about how this works.

LISTING E.25

THIS FUNCTION PARSES A STRING INTO WORDS, PUTTING EACH WORD INTO A SEPARATE ARRAY ELEMENT

```
Public Function ParseOut(ByVal strSource As
➥String, _
    strParseChar As String, _
    strTarget() As String) As Integer
    ÔCounter for number of elements parsed:
    Dim intElements As Integer
    ÔHolder for the currently parsed-out element:
    Dim strCurElement as String
    ÔWhere we found the parsing character
    'in the string:
    Dim intParsePos As Integer
    ÔMake sure the array is zeroed out
    Redim strTarget(0)
    ÔLoop until weÕve run out of material to parse:
    Do
        ÔFind position of parsing character in
➥string:
        intParsePos = Instr(strSource,
➥strParseChar)
        ÔIf we found the parsing character then
        If intParsePos > 0 Then
            Ôcurrent elementÕs everything to left of
            ➥Ôwhere we found parsing character:
            strCurElement = _
              Left(strSource,intParsePos - 1)
            Ônow chop element we just used off of
➥the
            Ôsource string:
            strSource = _
              Mid(strSource,intParsePos + 1)
        Else  Ôthis is the last element
              Ôso next parsed element is
              'everything that remains:
              strCurElement = strSource
              Ôand so nothing else remains to parse:
              strSource = ÒÓ
        EndIf
        ÔSo count one more element:
        intElements = intElements + 1
        ÔMake another array element to hold it:
        ReDim Preserve strTarget(intElements)
        Ôand assign it to the new array element:
        strTarget(intElements) = strCurElement
    Loop Until strSource = ÒÓ
    ÔWhen done, the number of elements is the
    Ôreturn value of this function:
    ParseOut = intElements
End Function
```

You could call the ParseOut function as you do in Listing E.26, passing it the String you wanted to parse and an empty String array.

LISTING E.26

CALLING THE PARSING FUNCTION

```
Dim strWords() As String
Dim iWords As Integer
Dim iCount As Integer
iWords = ParseOut(strSentence,Ó Ò,strWords)
For iCount = 1 To iWords
    Ô. . . Do something to each word
Next iCount
```

You could then use the return value of ParseOut to traverse the resulting array.

Using Arrays

An *array* is a dimensioned version of a basic data type. That is, instead of an array variable name referring to a single Integer, it can refer to many Integers. You can distinguish between the array's elements with a subscript or index argument to the array.

Therefore,

```
MyCounter = 7
```

refers to a single-dimensioned or nonarray variable. On the other hand,

```
MyCounter(17) = 7
```

refers to one of the elements of the MyCounter array.

An array has bounds; that is, it has a highest and a lowest position among its elements. As you might imagine, these are called the upper bound and lower bound, respectively.

> NOTE
> **Existence of Array Elements** Visual Basic will generate an error if you try to refer to an array element that does not exist.

Declaring Static and Dynamic Arrays

An array can take the same data types as other variables, but when you declare it, you must specify that it is going to be an array. To do so, you must put two parentheses after the array's name. You also can specify the upper bound (highest-numbered index) of the array within the parentheses when you declare the array, as in this declaration:

```
Dim MyNames(10) As String
```

The 10 tells Visual Basic the array's upper bound, or highest-numbered element in the array.

By default, Visual Basic starts the lower bound (lowest-numbered element) of its array elements at 0. In the preceding example, therefore, MyNames has 11 elements because its lower bound is 0 (the default) and its upper bound is 10.

> NOTE
> **Array's Default Lower Bound** If there are no further specifications in your code, Visual Basic starts an array's default lower bound at 0. This means an array dimension declared with a single number will have one more element than the upper bound.

> NOTE
> **Declaring an Array** You can't declare an array variable by using the Public keyword in a form module. You will get a compiler error.

When you specify an array's upper bound, you are committed to the specified number of elements for the duration of the program. However, you can change the number of elements of a dynamic array at runtime. You declare a dynamic array by omitting the specification of the bounds in the parentheses of the declaration. For example,

```
Dim MyNames() As String
```

will declare an undimensioned array with no initial elements. You can dynamically resize the array in your code. Later in this appendix, the section titled "Dynamically Resizing an Array with Redim" explains how to add and remove elements in a dynamic array.

Specifying the Bounds and Dimensions of Arrays

You can actually specify the lower bound of an array as well as the upper bound. For example,

```
Dim MyNames(5 to 10) As String
```

would tell Visual Basic to start the lower bound at 5 and put the upper bound at 10. This would yield six elements in the array (the array would have elements 5 through 10, inclusive).

You also may start the lower bound at a negative number. For example,

```
dim MyNames(Ð5 to 10) As String
```

would tell VB to start the lower bound at –5 and put the upper bound at 10. This would yield 16 elements in the array (0 would be included in the range of indices).

> **NOTE**
> **Array Bounds** Array bounds are inclusive and you must take them into account when you reckon the number of elements an array has. See the section titled "Calculating Total Elements in an Array" in this appendix.

Determining the Default Lower Bound of an Array and the Option Base Statement

As mentioned in previous sections, the default lower bound of an array is 0. This means that the first element in the array will have an index of 0.

You can change the default base for an array's lower bound to be 1 on a file-by-file basis by inserting this statement

```
Option Base 1
```

at the top of the General Declarations section of the module.

Therefore, the declaration

```
Dim MyNames(10) As String
```

would yield an array with 10 elements if the Option Base 1 statement were at the top of the General Declarations section in its module; and it would yield 11 elements if there were no Option Base statement or if the statement were Option Base 0 .

> **NOTE**
> Option Bases Only Option Base 1 and Option Base 0 are possible. Option Base 0 is the default, but many developers include it in their modules anyway to make their intentions clear.

> **WARNING**
> **When to Change the Option Base**
> It is not a good idea to change the Option Base on a module after you have declared array variables, because this may change the lower bound of each array and cause unforeseen problems.

Multidimensional Arrays

An array might also have more than one dimension. That is, each element may actually point to another "mini array" of elements. If each element of an array has one other set of elements associated with it, the array is "two-dimensional." You usually think of such an array as being made up of rows and columns, where the first dimension represents the rows and the second dimension represents the columns. Although it is a common way to visualize two-dimensional arrays, the row-column approach is a purely arbitrary way of looking at a two-dimensional array.

You might declare a two-dimensional array as in the following example:

```
Private Values(1 To 10, 1 To 3) As Integer
```

The array Values would then contain 10 "rows" in the first dimension and each "row" would hold three "columns" (the second dimension).

If each element of an array's second dimension has another "mini array" associated with it, the array is "three-dimensional." For instance, the declaration

```
Private Values(1 To 10, 1 To 3, 1 To 2)
```

creates an array with 10 "rows," three "columns" per row, and two values per "column."

In theory, this process of creating dimensions within dimensions can go on for many dimensions in Visual Basic, so that you can have "*n*-dimensional" arrays. Although the human mind has a little trouble visualizing arrays beyond three dimensions, many applications do use such arrays. In practice, an array with a huge number of dimensions would exhaust the available memory of even the most memory-rich systems.

To refer to an element of a two-dimensional array, a line of code might read

```
MsgBox Squares(1,7)
```

to tell Visual basic that you wanted the first element of the first dimension, and then within that first element you wanted to access the seventh element.

Each dimension of an array has upper and lower bounds, just like a single-dimensional array.

If you want more than one dimension in an array, you can specify each dimension after the first with a comma-delimited list.

```
Dim MyNames(5,6,7)
```

tells Visual Basic to allocate memory for a three-dimensional array of Variant (remember, Variant is the default) with upper bounds of 5, 6, and 7, respectively. Recall that the lower bound of each dimension will be 1 if the Option Base 1 statement has been issued. Otherwise, it will be 0.

You can specify lower and upper bounds separately for each dimension of a multidimensional array. For example,

```
Dim MyNames(5 to 10, 21 to 40, 21 to 40)
```

specifies three dimensions, and none of them start at the default of 0 (or possibly 1, if you have set Option Base 1).

You don't need to use the same specification style for all the dimensions in an array.

```
Dim MyNames(5 to 10, 40, 21 to 40)
```

specifies both the lower bound and upper bound for the first and third dimensions, but only the upper bound for the second dimension. The lower bound of the second dimension would be either 0 or 1 (depending on the specification in the Option Base statement).

Resizing a Dynamic Array

You can resize an array if it is declared as a dynamic array. Visual Basic recognizes a dynamic array if you specify no bounds or dimensions in its Declare statement.

```
Dim MyFriends() As String
```

declares a dynamic array that you can later resize. Fixed arrays cannot be resized. For example,

```
Dim MyNames(10) As String
```

declares a fixed array which you cannot later resize.

You may resize your dynamic array later in a line of code with the Redim statement.

```
Redim MyFriends(17)
```

would give 18 elements to the MyFriends array you declared a few lines previously (assuming the default array base has been left at 0).

There is no limit to the number of times you can resize an array with Redim. A few lines later in your array, you could make the statement

```
Redim MyFriends(19)
```

to resize the array again.

There is one little drawback—each time you use Redim, you re-initialize the data held in the array, and thereby destroy anything you had stored there earlier.

Not to worry—you save the contents of an array when you resize it by inserting the Preserve keyword after the word Redim.

```
Redim Preserve MyFriends(22)
```

would have kept data in the first 22 elements intact.

> **NOTE**
>
> **What Redim Preserve Actually Preserves** Redim Preserve preserves the existing values in array elements only when you resize the last dimension of an array. Resizing other dimensions or changing the number of dimensions with Redim Preserve will destroy the contents of existing array elements.

Determining the Number of Elements and Bounds of an Array

To compute the total number of elements in an array, just multiply the number of elements in all the array's dimensions.

That wasn't hard, was it?

But wait—how many elements are there in a given array dimension?

Remember to take into account the fact that array dimensions begin by default at element 0 (unless respecified with Option Base 1).

Also, remember that if the array's lower bound is specified, you must include the lower-bound element when counting number of elements in a dimension.

Remember that without Option Base 1 specified,

```
Dim MyNames(5,10) As String
```

specifies 66 elements (6×11) and not 50, because you must include element 0 in each of the dimensions.

Also,

```
Dim MyNames(5 to 20, 3 to 4) As String
```

specifies 32 elements (16×2), because you must include the lower bound when reckoning the size of a dimension.

Working with Collections

A *collection* is a special set of items in Visual Basic. Visual Basic itself implements some collections as a part of any program's running environment. All collections have a group of contained items and a Count property. In addition, most collections enable you to remove an item from them without regard to the item's position in the collection. Most collections also have a Key property that enables you to refer to collection elements either by numeric position or by a unique String identifier.

Some of Visual Basic's important collections are the Controls collection, which refers to all the controls on the current form, and the Forms collection, referring to all the forms currently loaded in memory in the running application.

You can refer to an item in a collection by its array index.

```
MsgBox Controls(0).Text
```

would, for instance, display the Text property of element 0 of the Controls collection.

The array of items of a collection is zero-based.

In Visual Basic 5 and 6, you can also loop through the items of a collection with the For Each loop. See the section on the "For Each_Next Loop" later in this appendix.

You will be getting a little preview of how to use the For Each_Next loop in the following sections, because that is the preferred way to traverse a collection in Visual Basic 6.

NOTE

Custom Collections In Visual Basic 5 and 6, the programmer can also implement custom collections.

PROGRAMMING WITH SUB AND FUNCTION PROCEDURES

Programs in most modern structured programming languages are implemented by the programmer as collections of subroutines that interact with each other and, if you are a Windows programmer, with the Windows operating environment.

Procedure is VB's term for a program's subroutine, and VB uses two different types of subroutines, the Sub procedure and the Function procedure, as discussed in the following sections.

Sub Procedures

A Sub procedure is a named section or block of code within the project that can be called by another part of the project to perform its instructions on demand. The Sub procedure may accept parameters, and it may even act on them, but it does not have any defined return value, which is the main difference between a Sub and Function procedure.

All Sub procedures use the keyword Sub on their first line (or declaration) and terminate with the line End Sub :

```
Private Sub ProcedureName(list of parameters)
    Ôcode goes here.
End Sub
```

The name of the Sub procedure is followed by parentheses, even if there are no parameters.

You can call a Sub procedure from somewhere else in your code (providing the procedure is in scope) just as you would call a built-in VB statement. To call the Sub procedure ProcName, just type the Sub procedure name at the appropriate point in your code:

```
ProcName list of arguments
```

or type:

```
Call ProcName (list of arguments)
```

Sub procedures are classified into two distinct types—Event procedures and General procedures.

Event Procedures

An Event procedure is different from other procedures in a couple of respects:

- Event procedures are triggered by an event associated with a control or form.
- All Event procedures are provided by VB and are associated with a particular control or form. VB uses the following syntax when naming the Event procedure:

  ```
  Sub ControlName_EventName
  ```

- Event procedures are stored only in form modules.

An Event procedure for the Click event of a command button named cmdQuit might look like this:

```
Private Sub cmdQuit_Click()
    UnLoad Me
End Sub
```

The programmer has put code (Unload Me) inside the procedure stub provided by VB.

NOTE

Event **Procedures** Event procedures are always Sub procedures, never functions.

You can find an Event procedure's code by following these steps:

1. While in Design mode, double-click a control or form.
2. The code window opens and displays one of the Event procedures belonging to the selected control or form.
3. To view all of the Event procedures belonging to the control or form, click the drop-down arrow to the right of the Proc combo box located at the top right of the code window.

If you double-click on a control or form and no code exists in any of the Event procedures, VB first displays the Event procedure it considers the object type's most commonly used procedure (for example, Click for a CommandButton and many other controls, Change for a TextBox, Load for a Form). If code exists in some of the control's Event procedures, VB displays the first Event procedure in alphabetic order that contains code.

By default, VB lists only one procedure at a time within the code window. If you prefer to look at all procedures in the code window as a single scrollable listing, select Tools, Options, and then the Editor tab. Check the Default to Full Module View box in the Window Setting frame.

When in the code window, you can also toggle full module view on and off with the two small text icons in the extreme lower-left corner of the window.

Typically, Event procedures run when their respective events are triggered by the user (Click, KeyDown, MouseMove). Some Event procedures, however, run when their events are triggered by the system (Load, Unload, Timer).

In addition, you can force an Event procedure to run by explicitly calling it as you would a General procedure. To run the Click Event procedure for cmdQuit, for example, use the following:

```
cmdQuit_Click n
```

General Procedures

A General procedure is designed to run only when you explicitly call it from somewhere in code. In addition, you must create a General procedure from scratch (unlike an Event procedure).

If you create a Sub procedure called CleanUp in a form, standard, or class module, it might look like Listing E.27.

LISTING E.27

A GENERAL PROCEDURE

```
Private Sub CleanUp(blnUnload As Boolean)
    Dim frmCurForm As Form
    If blnUnLoad Then
        For Each frmCurForm in Forms
            Unload frmCurForm
        Next frmCurForm
    End If
    End
End Sub
```

To call the CleanUp Sub procedure, just type the procedure name followed by any parameters:

```
CleanUp True
```

or use the alternative syntax:

```
Call CleanUp(True)
```

Notice that the syntax of the first example resembles a call to a built-in VB statement.

Initializing Procedure Code from the Insert Menu

General procedures are always stored in the General section of the form, standard, or class module in which they are created.

You can create a General procedure stub in one of two ways—through a menu-accessible dialog box or just by typing the procedure stub.

To create a General procedure using the menu, follow these steps:

1. Open the code window for the desired form, standard or class module. (Opening the code window is required. It is not important what the code window displays when it is opened.)
2. From the VB menu, choose Tools, Add Procedure to access the Add Procedure dialog box.
3. Type the desired name in the Name text box.
4. Choose the procedure's type from the Type group—Sub, Function, or Property.
5. Choose the procedure's scope from the Scope group.
6. Click OK.
7. VB places the general procedure stub in the General Declaration section. (It was not important what the code window displayed when it was opened because all General procedures will be stored in the General Declaration section.)

Many programmers prefer to avoid yet another dialog box, and instead use the technique discussed in the following section to initialize a new procedure.

Initializing Procedure Code by Typing in the Code Window

You also can initialize a new Sub procedure or Function by typing it within the code window:

1. Open the code window of the desired form, general, or class module.
2. Move to the General Declaration section (or position the cursor after the End Sub or End Function of an existing procedure).
3. Type the first line of the procedure declaration. You need not include the parentheses for the parameter list because VB includes them automatically.
4. Press the Enter key and VB completes the procedure stub with End Sub or End Function.

Although less "correct," this technique is faster and more widely used than the technique discussed in the preceding section.

Functions

A Function is a General procedure that returns a value. Unlike a General procedure of the Sub type, a Function procedure includes some extra components in its procedure stub:

- A Function must include a data type for its return value. You declare the return value's data type with an As DataType clause at the end of the Function's declaration.
- You must specify the Function's return value within the Function's code block. In VB, you do this by treating the Function's name as if it were a variable and assigning a value to it.

In this example, the function Tomorrow() has a return type of Date, and the return value is assigned within the Function's code block:

```
Private Function Tomorrow() As Date
    Tomorrow = Now + 1
End Function
```

To call the Function named Tomorrow somewhere within code, type this:

```
datDeadline= Tomorrow()
```

In this example, the variable datDeadline stores the return value of the Function named Tomorrow.

Declaring the Data Type of a Function's Return Value

Because a Function has a return value, it needs a data type just like a normal variable. To specify the data type, append the following:

```
As DataType
```

at the end of the Function's declaration line, where DataType is a standard VB data type. For example,

```
Private Function Tomorrow() As Date
```

would be a valid Function declaration, specifying Date as the data type of Tomorrow's return value.

You may, when necessary, declare the return type to be Variant:

```
Private Function Tomorrow() As Variant
```

Because Variant is VB's default data type, the declaration

```
Private Function Tomorrow()
```

is legal and is equivalent to an explicit Variant declaration.

Setting the Return Value

As previously stated, you set a Function's return value by assigning a value to a variable with the same name as the Function. Because most Functions require more than a single line to complete their task, you might consider declaring a Local variable inside the Function to hold the return value, finally assigning that variable's value to the Function name at the end, as in the example of Listing E.28.

This example uses datRetVal as an internal placeholder for computing the return value of the Function Tomorrow. The example then assigns datRetVal as the return value of the Function in the very last line.

LISTING E.28

USING A Local VARIABLE TO HOLD A Function'S RETURN VALUE

```
Private Function Tomorrow() As Date
    Dim datRetVal As Date
    Ôinvolved procedure to assign a value to
➥datRetVal
    Ô.
    Ô.
    Tomorrow = datRetVal
End Function
```

Passing Arguments to General Procedures

Arguments enable dynamic information to be passed to a procedure. A procedure declaration specifies the number and type of arguments required by the procedure within the parentheses that follow the procedure name. Typically, the complete list of arguments from the procedure's point of view is called the parameter list. The general format for specifying a parameter list for a Sub or Function procedure is this:

```
Private Sub ProcedureName(list of parameters)
Private Function FunctionName(list of
parameters) as DataType
```

When you specify a parameter, you assign it a name by which it will be known inside the procedure, and a data type:

```
datToday As Date
```

This name is known only inside the body of the procedure. In the following example, the Function procedure called NextDay expects a single Date-type parameter which will be known as datToday inside the procedure:

```
Function NextDay(datToday As Date) As Date
    NextDay = datToday + 1
End Function
```

When you want to call this function procedure from elsewhere in your code, you must pass a Date -type value as an argument, as illustrated in Listing E.29.

LISTING E.29

PASSING A DATE-TYPE ARGUMENT TO A FUNCTION

```
Dim datTomorrow As Date
Dim datStartDate As Date
datStartDate = Now
datTomorrow = NextDay(datStartDate)
```

WARNING

The Scope of a Parameter is Local
Remember, the parameter's scope as a variable is Local to its procedure.

Don't be misled, therefore, by many examples that use the same name for a variable passed as an argument and for its corresponding parameter.

Also, don't be misled by an application- or file-wide variable or a Local variable in a calling routine that has the same name as a parameter: As stated, the parameter is a Local variable to the procedure, and therefore, only the value of the parameter changes. This change does not affect the values of any other variables, even if those variables' names are the same as the name of the parameter whose value was changed.

In this example, when the NextDay function procedure is called, the value of datStartDate is passed to datToday , a Date -type parameter referenced within the NextDay function.

Notice that the argument, datStartDate , has a name different from datToday , the parameter for which the value of datStartDate is passed. This is because an argument is passed to its corresponding parameter by position and data type, not by name.

Procedure Calls and the VB Stack

Many programming languages, including VB, set aside a special area of memory, known as the *stack*, for holding temporary values. Among the temporary values stored on the stack are all Local variables and the parameters of procedures, as well as the return values of Function procedures.

Whenever VB calls a procedure, it initializes space on the stack for the following:

- The memory address of the caller to this procedure
- The memory addresses of any parameters
- Any Local variables in the procedure
- Return value of the routine, if it is a Function procedure

After VB encounters the End Sub or End Function in the procedure, it returns to the address of the caller, uses any return value or modified parameter as needed, and then frees that part of the stack for other uses. This is the reason Local variables have their limited lifetimes: Their references on the stack are destroyed after the routine is over.

You will find it useful to learn about the stack when answering some of the questions on the certification exam and for better understanding the reasons for the way VB passes arguments to routines.

By Reference and By Value Arguments

Because VB passes memory addresses on the stack from caller to subroutine, you can see that it doesn't matter that the argument names used by the caller don't match the parameter names in the procedure.

Both caller and subroutine are looking at the same memory address when the caller passes an argument that the subroutine sees as a parameter.

In Listing E.30, the function NextWorkingDay takes as its parameter a Date -type variable, which it calls datToday . Notice that in the course of executing the code block, NextWorkingDay modifies the value of datToday , incrementing it by a value of either two or three.

LISTING E.30

THIS FUNCTION TAKES A SINGLE PARAMETER, WHICH IT MODIFIES

```
Private Function NextWorkingDay(datToday As Date)
➥As Date
    ÔIn this function, we use the datToday
➥parameter
    Ôpassed to the function as a working
    Ôvariable for computations inside the function
    Dim intDayOfWeek As Integer

    Ôthe Weekday function returns an integer
    Ôfor day of week
    intDayOfWeek = WeekDay(datToday)

    Ôand we can compare its results with VB
    Ôconstants for the weekdays
    If intDayOfWeek = vbFriday Then
        datToday = datToday + 3
    ElseIf intDayOfWeek = vbSaturday Then
        datToday = datToday + 2
    Else
        datToday = datToday + 1
    End If

    ÔFinally, we assign the return value of the
➥function
    Ôto be the new computed value of datToday
    NextWorkingDay = datToday
End Function
```

In Listing E.31, you call the NextWorkingDay Function , passing a Date -type variable, datStartDate , to the function. When you display the value of datStartDate after the Function has run, you see that it has changed.

When the Function changed the value of its parameter, it changed the contents of the memory location of datStartDate .

LISTING E.31

ANOTHER PART OF THE CODE MAKES A CALL TO THE FUNCTION

```
Dim datTomorrow As Date
Dim datStartDate As Date
datStartDate = Now
datTomorrow = NextWorkingDay(datStartDate)
? datStartDate      Ôvalue was changed in the
➥function
```

When VB passes the original address of an argument to a procedure as a parameter, it is said that VB has passed the argument *by reference*. Passing by reference is the default method for argument passing in VB. Changes made to the parameter by the procedure will appear in the variable passed as an argument from the calling routine. This is because both the argument and the parameter point to the same memory location.

Sometimes, however, you would rather not allow changes to be made to the variables you pass as arguments to your function or procedure. In such cases, you can force VB to make a copy of the argument at another memory location and pass the copy instead of the original. Therefore, the subroutine gets the same value as the original argument, but doesn't get its hands on the underlying variable. This style of argument-passing is known as passing *by value*.

You can specify that a parameter always be passed by value by putting the keyword ByVal before its name in the subroutine's declaration:

```
Function NextWorkingDay(ByVal datToday As Date)
➥As Date
```

In the preceding example, VB always uses a copy of any variable that a caller passes to datToday , instead of using the original memory address.

> **NOTE**
>
> **Specifying by Value** Notice that the ByVal keyword is placed in front of the specified parameter within the procedure declaration. An alternative way to specify by value passing is to use parentheses () around the variable name in the call instead of putting the keyword ByVal in the procedure declaration.

See "Putting Extra Parentheses Around Each Argument."

Multiple Arguments

So far, the examples of procedures with parameters have shown only a single parameter in the subroutine declaration. If you want to pass more than one value at a time to a subroutine, you need only specify several parameters separated by commas in the parameter list.

In Listing E.32, the more generalized function NextDay takes a second parameter telling the function whether to return the next working day rather than just the next calendar date (sorry, holidays are included).

LISTING E.32

THE FUNCTION NOW TAKES A SECOND PARAMETER

```
Private Function NextWorkingDay(datToday As Date,
➥blnWorkingDay) As Date
    ÔIn this function, we use the datToday
➥parameter
    Ôpassed to the function as a working
    Ôvariable for computations inside the function
    Dim intDayOfWeek As Integer

    Ôthe Weekday function returns an integer
    Ôfor day of week
    intDayOfWeek = WeekDay(datToday)

    Ôand we can compare its results with VB
    Ôconstants for the weekdays

    ÔIf we want just working days , then
    Ôcheck for Friday or Saturday
    If blnWorkingDay Then
        If intDayOfWeek = vbFriday Then
            datToday = datToday + 3
        ElseIf intDayOfWeek = vbSaturday Then
            datToday = datToday + 2
        Else
            datToday = datToday + 1
        End If
    Else
    Ôotherwise, we always want the next day
        datToday = datToday + 1
    End If
    ÔFinally, we assign the return value of the
➥function
    Ôto be the new computed value of datToday
    NextWorkingDay = datToday
End Function
```

> **NOTE**
>
> **The Weekday Function and Weekday Constants** The Weekday function seen in the example is a built-in VB function, which returns an Integer indicating the day of the week (1 = Sunday, 7 = Saturday). VB uses constants such as vbSunday , vbMonday , and so forth to test these values.

When calling the NextDay function, provide two arguments as in Listing E.33 (note that the second argument in the example is a literal value rather than a variable).

LISTING E.33

ARGUMENTS TO THE NextDay FUNCTION

```
Dim datTomorrow As Date
Dim datStartDate As Date
datStartDate = Now
datTomorrow = NextDay(datStartDate, False)
```

Here, we pass the Date variable datStartDate , and the literal Boolean constant False .

Using Named Arguments

There are two ways to specify arguments—positional and named. Up to this point, you have exclusively used positional arguments.

Positional arguments are identified by their particular order in the argument list. Take, for example, the NextDay function seen in the previous sections. Assume that NextDay requires two values—datToday and blnWorkingDay , as illustrated in Listing E.34

LISTING E.34

FUNCTION WITH TWO REQUIRED PARAMETERS

```
Function NextDay(datToday As Date, _
  blnWorkingDay As Boolean) As Date
      Ôsome code
End Function
```

When you call the NextDay function, you are obligated to pass the Date variable first and the Boolean value as the second argument because this order has been defined for you in the NextDay function procedure:

```
datTomorrow = NextDay(datStartDate, False)
```

Visual Basic's own functions, procedures, statements, and methods support positional arguments. For example:

```
strResponse=MsgBox(prompt[, buttons][, title][,
helpÞle, context])
```

If you call the Msgbox function, you must pass the arguments in the order specified by the Msgbox function. This requires that you maintain a placeholder for the third argument, title , if you do not pass an explicit value for the title.

```
iSave=Msgbox(ÒSave current record?Ó,vbYesNo,
,ÓDEMO.HLPÓ,1000)
```

Named arguments eliminate the need to pass arguments in a predefined order. The VBA language engine included in VB6 enables you to pass arguments without concern for the order in which they are passed. Your call to MsgBox could look like Listing E.35.

LISTING E.35

NAMED ARGUMENTS FOR MSGBOX

```
iSave=MsgBox _
        Title:=ÓSAVEÓ, _
        Prompt:=ÓSave current record?Ó, _
        Buttons:=vbYesNo
```

This feature, however, requires some additional work to pass named arguments.

In the example at the beginning of this section, you defined the parameter names within the NextDay function as blnWorkingDay and datToday . When calling the NextDay function, you can use these parameter names to pass the arguments in any desired order. For example, calls to the NextDay function can now take more than one form, either

```
datTomorrow = NextDay(datToday := datStartDate, _
                      blnWorkingDay := False)
```

or

```
datTomorrow = NextDay(blnWorkingDay := False , _
                      datToday := datStartDate)
```

Remember, named arguments do not eliminate the need to pass required arguments. They just alleviate the need to pass the arguments in a particular order.

NOTE

Support for Named Arguments The use of named arguments also applies to some of VB's procedures, functions, and statements. However, named arguments aren't supported universally in Visual Basic. If you are not sure about the particular function or statement you are interested in, consult Help or the Object Browser on the specific topic.

The following list summarizes the advantages of named arguments:

- Your code is easier to read and maintain.
- You don't have to provide blank placeholders in your argument list for unused optional arguments.
- You don't have to pass arguments in the default order.

When they don't cause too much clutter in your code, named arguments will in general make it more readable.

Using Optional Arguments and the IsMissing() Function

Visual Basic provides a way to specify optional arguments. To define an optional argument, place the Optional keyword in front of the corresponding parameter. In addition, you need to define the parameter as a variant if you want to use the IsMissing() function as described later.

Of course, you will have to write extra logic in your procedure to determine whether the argument was passed by the caller. If the argument passed is a of type Variant, the procedure should use the IsMissing() function to determine whether the caller has passed the parameter. If the argument is any other data type, you should check for an appropriate blank value for that data type (0 for Numeric data types, ÒÓfor Strings).

The fragment in Listing E.36 has modified the example from the previous sections to indicate that the second argument is optional. The IsMissing() function tells whether the caller passed the Optional parameter. If not, the procedure assigns a default value for the Optional parameter.

LISTING E.36

AN OPTIONAL ARGUMENT

```
Function NextDay(datToday As Date, _
  Optional blnWorkingDay As Variant) _
  As Date
      If IsMissing(blnWorkingDay) Then
            blnWorkingDay = False
      End If
...
```

In this example, if the caller doesn't pass an argument for the blnWorkingDay parameter, blnWorkingDayÕs value defaults to False .

You can call the NextDay function by passing a value to the second parameter or by ignoring the second parameter entirely:

```
datTomorrow = NextDay(datStartDate, True)
datTomorrow = NextDay(datStartDate, False)
datTomorrow = NextDay(datStartDate)
```

NOTE

The Data Type of an Optional Parameter In versions of VB before 5, the data type of an Optional parameter had to be Variant . In VB 5 and VB 6, Optional parameters can take on any data type.

WARNING

IsMissing **Limitation** The IsMissing() function only works properly with Variant Optional parameters.

If you use IsMissing() to check on non-Variant Optional parameters, your code will still run, but IsMissing will always return False, even when you don't pass the parameter.

WARNING

Parameters **and** Optional **Keyword** All parameters listed after the first Optional keyword must be optional as well.

You must use the Optional keyword before each optional parameter.

Breaking either of these rules will cause a compiler error.

You can also specify a default value for an Optional parameter in the procedure header, as shown in Listing E.37. This often eliminates any need for the use of the AsMissing function, as the default value takes effect when the caller does not specify the Optional parameter.

LISTING E.37

AN OPTIONAL ARGUMENT

```
Function NextDay(datToday As Date, _
  Optional blnWorkingDay As  Boolean =  False) _
  As Date
...
```

Array Arguments and the ParamArray Keyword

What if you need to pass an undetermined number of Optional parameters?

VB enables you to pass a variable number of parameters with a ParamArray argument. The ParamArray argument must appear last in your parameter list in the procedure's declaration.

You must observe the following rules to implement a variable number of parameters in a procedure declaration:

- Specify the final parameter as an array by appending a set of parentheses to its name.
- Place the keyword ParamArray before the parameter name.
- Specify as Variant type (doesn't need to be explicit, because Variant is the default).

You can then put code within the procedure to process the array's elements.

In this example, you will write a procedure that can accept an indeterminate number of TextBox parameters and convert the contents of each TextBox control's Text property to all uppercase or all lowercase.

In Listing E.38, an initial parameter indicates whether to convert to upper- or lowercase, and then you specify a ParamArray that holds the TextBox controls whose Text properties you want to convert:

LISTING E.38

THIS FUNCTION TAKES A PARAMARRAY ARGUMENT

```
Private Sub ConvertText(blnUpper As Boolean, _
ParamArray boxes() As Variant)
      Dim varCurrBox As Variant   ÔFor loop
➥placeholder
      For Each varCurrBox In boxes()
            If blnUpper Then
```

```
            varCurrBox.Text _
              = UCase(varCurrBox.Text)
        Else
            varCurrBox.Text _
              = LCase(varCurrBox.Text)
        End If
    Next varCurrBox
End Sub
```

As seen in Listing E.39, you can call the routine with the names of as many TextBoxes as you want:

LISTING E.39

CALLING A FUNCTION THAT CAN TAKE A VARIABLE NUMBER OF ARGUMENTS

```
[Click event procedure of a command button]
Private Sub cmdConvert_Click()
    ConvertText True, Text1, Text2, Text3, Text4
    ConvertText False, Text2, Text3
End Sub
```

WARNING

ParamArray **and** Optional **Arguments** ParamArray arguments and Optional arguments can't appear together in the same procedure's parameter list.

The ParamArray argument must be the final argument in the parameter list.

Breaking either of these rules will cause a compiler error.

Using Exit Sub or Exit Function to Abruptly Terminate a Procedure

You can use Exit Sub or Exit Function to immediately cause Visual Basic to terminate the current procedure or function. No more code in this procedure or function will execute, the stack area allocated for this routine will be cleared, and control will return to the calling code.

Sticklers for well-structured programming techniques try to avoid using Exit Sub /Function as much as possible. Therefore, if you had a routine that needed to check some condition before running, you could put an If construct around all the code in the routine so that the routine would only do its business if some condition were true, as illustrated in Listing E.40.

LISTING E.40

AN IF CONSTRUCT SURROUNDS ALL THE CODE IN THIS ROUTINE

```
Sub MySub()
    If IsGoodIdea Then
        Ô. . . do the routine's work
    End If
End Sub
```

A slightly less awkward way to accomplish this is to use the Exit Sub statement if you encountered a bad condition, as shown in Listing E.41.

LISTING E.41

USING EXIT SUB

```
Sub MySub()
    If Not IsGoodIdea Then Exit Sub
    Ô. . . do the routine's work
End Sub
```

Using Exit Sub as in the preceding example is perhaps an aesthetic choice. Structured programming purists will argue that you shouldn't use it because it makes your code less maintainable; others will argue that having the entire procedure's code wrapped in some initial evaluation is logically cumbersome.

The following example, however, would be tedious to implement without Exit Sub /Function . In this example, something happens nested way down inside some internal logic that causes you to want to drop everything you are doing in the routine immediately. Although it is possible to set up a bunch of flags and checks to avoid using an Exit , the extra effort makes the code harder to follow and easy to break. Exit Sub/Function , on the other hand, provides a quick escape hatch to leave the whole mess behind and forget about it, as shown in Listing E.42.

LISTING E.42

EXIT SUB AS A QUICK WAY OUT OF A COMPLEX SITUATION

```
Sub MySub()
    Ô. . . do everything the routine is supposed
➥to do
    Ô. . . and then, maybe somewhere way down
➥nested
    Ô. . . a couple of levels into some loops or
➥If
    Ô. . . constructs, we just need to GET OUT
➥NOW:
    Do While something
        If somethingelse Then
            ÔTry doing this without Exit Sub —
            Exit Sub ÔweÕre outta here!
        EndIf
    LoopÕ. . . etc. etc.
End Sub
```

WARNING

Use Exit Sub **/** Function **Sparingly** Although an Exit Sub /Function can be handy, use it sparingly, because overuse can result in more difficulty at the debugging stage, as well as code maintenance nightmares.

Syntax for Calling Procedures

Thus far, you have reviewed the basics of how Sub and Function procedures are called.

The following text explores variations in calling Sub and Function procedures.

Using the Call Statement to Call Sub Procedures

There are two ways to invoke a Sub procedure:

```
ConvertText intLCase, Text1, Text2, Text3
```

or

```
CALL ConvertText(intLCase, Text1, Text2, Text3)
```

When you use the keyword CALL to invoke a Sub procedure, you must enclose the argument list in parentheses, whether there is one argument or multiple arguments.

WARNING

Call **Keyword and Parentheses** If you are not passing any arguments, do not use parentheses with the CALL keyword:

```
Call MySub()    ÔWrong
Call MySub    ÔCorrect
```

Putting Extra Parentheses Around Each Argument Passed By Value

You can always force an argument to a procedure to be passed by value, regardless of whether the procedure's declaration specifies ByVal or takes the default By Reference . All you have to do is put an extra set of parentheses around each individual argument that you want to pass by value. Even though the procedure's declaration might show that the parameter is passed by reference, the extra set of parentheses around an argument will cause the argument to be passed by value instead.

In the first call to the NextDay procedure in the following example, notice the extra set of parentheses around the first parameter, datToday . This specifies that datToday will be passed by value. The second call does not contain extra parentheses around its arguments, and so VB will pass each argument as specified by the procedure.

LISTING E.43

DIFFERENT MEANINGS OF PARENTHESES IN ARGUMENT LISTS

```
ÔdatToday will be passed by Value:
datTomorrow = NextDay((datToday), blnWorkingDay)

ÔdatToday will be passed by Reference:
datTomorrow = NextDay(datToday, blnWorkDay)
```

The Sub procedure calls in Listing E.44 would not normally use parentheses around their arguments. (Notice that these examples do not include the keyword CALL, which requires an outer set of parentheses around all of its arguments.) The first example indicates that the argument blnWantUnload is to be passed by value because the argument is surrounded by parentheses. Observe the Sub procedure call, which passes multiple arguments. Notice that the argument list itself takes no parentheses, but the parentheses around the argument, Text2 , indicate that Text2 will be passed by value.

LISTING E.44

ARGUMENTS PASSED BY VALUE AND BY REFERENCE

```
ÔblnWantUnload will be passed by Value:
CleanUp (blnWantUnload)

ÔText2 will be passed by Value:
ConvertText intUCase, Text1, (Text2)

ÔblnWantUnload will be passed as speciÞed by the
➥CleanUp procedure:
CleanUp blnWantUnload

ÔText2 will be passed as speciÞed by ConvertText:
ConvertText intUCase, Text1, Text2
```

The examples in Listing E.45 display Sub procedure calls that use the Call keyword to invoke a procedure. As you know, parentheses must surround argument lists when the keyword Call is used. However, notice a second parenthesis around any argument that will be passed by value.

LISTING E.45

PASSING ARGUMENTS BY VALUE AND BY REFERENCE WHEN USING THE CALL KEYWORD.

```
ÔblnWantUnload will be passed by Value:
CALL CleanUp((blnWantUnload))

ÔText2 will be passed by Value:
CALL ConvertText (intUCase, Text1, (Text2))

ÔblnWantUnload will be changed by the CleanUp
➥procedure:
Call CleanUp(blnWantUnload)

ÔText2 will be changed by the ConvertText
➥procedure:
CALL ConvertText (intUCase, Text1, Text2)
```

Note in the examples of the listing that any parentheses that would have been required without the ByVal invocation must still be supplied.

Ignoring a Function's Return Value

You can call a function without using its return value just by calling it as if it were a Sub procedure rather than a function.

Assume you have a function, MyFunc, which takes two parameters. In this instance, the return value of MyFunc is assigned to a variable called RetVal :

```
RetVal = MyFunc(MyParam1, MyParam2)
```

If you don't plan to use the function's return value, however, just call the function in the way you would call a Sub procedure, omitting a reference to the return value as well as parentheses around the argument list:

```
MyFunc MyParam1, MyParam2
```

Remember that there is an alternative style with the Call keyword for calling a Sub procedure. You can also call a function with this syntax to ignore its return value:

```
Call MyFunc(MyParam1, MyParam2)
```

Finally, if you don't need the return value but need to make sure an argument is passed by value, just remember to put the extra parentheses around any arguments you want to be passed by value. In the following examples, MyParam1 will be passed by value, and MyParam2 is passed by reference:

```
MyFunc (MyParam1), MyParam2
Call MyFunc((MyParam1), MyParam2))
```

WARNING

Return Values and Built-In Functions This rule works with built-in VB functions as well. For example, you can call the MsgBox function with or without using its return value:

```
intChoice =
MsgBox(ÒContinue?Ó, vbYesNo)
Call MsgBox(ÒThere are no
customers leftÓ)
MsgBox ÒThere are no
customers left.Ó n
```

Procedure Scope

Previous sections discussed the different types of scope, or visibility, assigned to a variable within a VB application. Recall that a variable's scope is determined, in part, by the keyword used to declare it.

Procedures follow scoping rules similar to the scoping rules for variables. You can use the keywords Private or Public to indicate the scope of a procedure.

Visual Basic offers two distinct levels of availability or scope for procedures within an application: Private (module-level) or Public (application-wide).

Private Scope

Private procedures can only be called from other procedures within the same form, standard, or class module. If you place the keyword Private before the procedure name, the compiler recognizes only the procedure's name from code that runs from within the same form, standard, or class module. Listing E.46 gives examples of Private procedure declarations.

LISTING E.46

PRIVATE PROCEDURE DECLARATIONS

```
ÔSub procedure with private scope
Private Sub MySub(MyParam as Integer)
    Ô....do something
End Sub

ÔFunction procedure with private scope
Private  Function MyFunc(MyParam1 as Integer) as
➥Long
    Ô....do something
End Function
```

The VB Editor gives an initial scope of Private to Event procedure stubs:

```
ÔEvent procedures begin with Private scope.
Private Sub Command1_Click()
```

Public Scope

Public procedures can be called from anywhere within the project. If you place the keyword Public before the procedure name, the compiler recognizes the procedure's name from anywhere in your application, as illustrated in Listing E.47.

LISTING E.47

PUBLIC PROCEDURES

```
ÔSub procedure with public scope
Public Sub MySub(MyParam As Integer)
Ô....do something
End Sub

ÔFunction procedure with public scope
Public Function MyFunc(MyParam1 As Integer) As
➥Long
Ô....do something
End Function
```

Although the Public keyword does not need to be explicitly stated, the default scope for general procedures is Public.

Remember, however, that even if it is Public, you must call a routine in a form or class module from elsewhere in your application with the form name or the class object variable's name prefixed to the routine name. When you call a form's Public routine from elsewhere in the application, in other words, you must call the routine as if it were a method of the form:

```
frmMain.MySub intPara1
```

> **NOTE**
> **Calling General Procedures** You may call general procedures stored in standard modules with the module name in front of the routine, but this is not required.

> **NOTE**
> **Scope in Earlier Versions of VB** In versions of Visual Basic before VB 4, all routines stored in a form automatically had Private scope and could not be changed to Public scope.

Friend scope

Another type of scope, Friend, is useful within COM component projects. The section titled "Using the Friend Keyword" in Chapter 12, "Creating a COM Component That Implements Business Rules or Logic" discusses the Friend keyword in greater detail.

Control Names and Event Procedure Names

Visual Basic provides you with an event procedure stub based on the control's current name. If you change the name of the control, VB will provide you with a set of new event procedure stubs based on the control's new name. The code you wrote under the previous name's event procedure still exists; however, it is no longer associated with the control whose name has been changed.

If you add a command button named Command1 to a form, write code in its Click event procedure, and then change Command1's name to cmdOK, for example, the event procedure name does not change and would still be named Sub Command1_Click. Therefore, the procedure would no longer be associated with the command button. (Sub cmdOK_Click would be the name of the current Click event procedure, and obviously, this event procedure contains no code.)

Conversely, if you happen to write a general procedure and later rename a control in such a way that one of its event procedure names happens to match the name of the existing general procedure, that general procedure becomes an event procedure for the control.

If you write a general procedure whose declaration looks like this, for example,

```
Private Sub Bozo_Change()
```

and then later rename a TextBox control as ÒBozoÓ, VB associates the Bozo_Change procedure with the TextBox named ÒBozoÓ.

WARNING

The Fate of Event Procedure Code After You Rename a Control When you rename a control, its event procedures are no longer associated with the control. If you rename a TextBox control named txtFName to txtCompanyName, the event procedures (txtFName_Change(), for example) do not get renamed and, therefore, any code you have written in the control's event procedures are no longer associated with the control.

NOTE

Finalize Names of Controls Before Writing Event Procedure Code Finalize the names of all your controls and rename them to the desired names before you put any code in their event procedures or write any other code that calls these event procedures by name.

PROGRAMMING WITH VB'S CONTROL STRUCTURES

The backbone of any modern structured programming language is its implementation of the control structures of looping (or iteration) and branching (or selection). These control structures enable the programmer to easily cause selective or repeated execution of lines of program code.

The following sections discuss how VB implements branching and looping control structures.

Branching or Selection

VB offers two basic types of branching constructs:

- The If structure is used most often when you want to implement one or two alternative execution routes in your code, or when there are multiple alternatives with disparate conditions.
- The Case structure is used when there are multiple alternatives for the same expression or variable value.

If

Use the If statement when you want to execute an action conditionally. The syntax for the If statement is this:

```
If [logical condition] then [action]
```

The If...End If Construct

In the If construct's simplest form, all the lines that follow the If statement through the End If statement will execute only if the condition is True. In case the If condition is not met, program control moves to the last line, End If. For example:

```
If UCase(Left(LastName,2)) = ÒMCÓ Then
    Ô. . . Do some stuff
End If
```

Else and ElseIf

The second simplest form of the If statement allows a second set of statements to execute when the condition is False. In such a situation, you would insert the Else statement between the If and the End If. Everything between the Else and the End If will execute only if the condition is False, as illustrated in Listing E.48.

LISTING E.48

IF...ELSE...END IF

```
If UCase(Left(LastName,2)) = ÒMCÓ Then
    Ô. . . Do some stuff
Else
    Ô. . . Do something different
End If
```

Finally, if you have several conditions and want to have something different happen in each case, you can put as many instances as you need of the ElseIf statement inside the If construct. Lines containing ElseIf statements must have a condition to evaluate, followed by the word *Then*. Everything after the ElseIf up to the next ElseIf, Else, or End If will execute, provided

- The condition of the ElseIf is True.
- No previous condition in the If construct was True. In other words, no more than one of the branches of an If construct will execute each time the program runs through the If construct.

You can put a final Else condition between the last ElseIf and the End If. Use a final Else clause if you need to catch any situation where none of the previous conditions are True (a "none of the above" condition).

Listing E.49 illustrate the use of ElseIf.

LISTING E.49

USING ELSEIF

```
If UCase(Left(LastName,2)) = ÒMCÓ Then
    Ô. . .
ElseIf UCase(Left(LastName,2)) = ÒOÕÓ Then
    Ô. . .
ElseIf UCase(Left(LastName,3)) = ÒVANÓ Then
    Ô. . .
ElseIf UCase(Left(LastName,2)) = ÒDEÓ Then
    Ô. . .
Else  ÔHandle all other types of last name
    Ô. . .
End If
```

In essence, the ElseIf construct enables you to turn the If construct from a "true/false" evaluation into a "multiple-choice" evaluation.

The If Construct Within One Line of Code

If there is only one line of code to run between the If and the End If, you can efficiently and clearly write the entire construct in one line:

```
If IsBadIdea Then MsgBox ÒBadÓ
```

Notice that the one-line construct does not require the use of End If.

Similarly, if there is also an Else clause with only one line of code as well, you can include the Else clause on the same line. As in the preceding example, the End If is not required:

```
If IsBadIdea Then MsgBox ÒBadÓ Else MsgBox
ÒGoodÓ
```

Case

The Case structure also offers conditional branching, but with more limited use. It is only practical to use a Case structure when you want to take different actions based on different values of the same variable or expression.

> **NOTE**
> **Why the Case Structure?** You might wonder why VB has the Case structure at all, because it is of more limited use than the If structure. The answer is that the Case structure will execute faster than an If_ElseIf structure that implements the same logic.

A Case structure begins with the words Select Case, followed on the same line by the variable or expression to be evaluated.

The Case structure ends with a line containing the words End Select . Between the Select Case and End Select statements, you can put any number of Case statements. After each Case statement, place a value for the expression or variable that is being evaluated. The lines of code following the Case and continuing until the next Case or the End Select will execute if the Select Case statement's expression matches the value provided by the Case.

If none of the Cases are fulfilled, you can put a final End Case clause between the final Case and the End Select . This provides a "none-of-the-above" option.

In Listing E.50, the comparison expression is UCase(Left(ProductID,3)) . Assuming that the first three characters of ProductID specify the product's color, the following example provides three possible values for UCase(Left(ProductID,3)) and a catch-all Case Else clause in case you run into unexpected values.

LISTING E.50

THE CASE SELECT STRUCTURE

```
Select Case UCase(Left(ProductID,3))
    Case ÒREDÓ
        Ô. . .
    Case ÒBLUÓ
        Ô. . .
    Case ÒYLWÓ
        Ô. . .
    Case Else
        Ôit wasnÕt one of the above options
End Select
```

> **NOTE**
>
> **Execution of the Case Construct** As with the If construct, no more than one of the Case construct's options will execute within a single pass through the construct. This means that, if two Cases are both True , only the first one will execute.

Enumerating Values in a Case Construct

You can go beyond enumerating one or more simple values in the Case statement (see Listing E.51):

- ‡ You can use the keyword Is before a comparison operator (<, …, >, , and so on) when evaluating literals. (Even if you omit the Is when typing, the VB Editor will automatically include it for you.)
- ‡ You can enumerate possible individual values in a comma-separated list.
- ‡ You can use the keyword To between two values to designate an inclusive range.
- ‡ You can combine any of the three previous options.

LISTING E.51

ENUMERATED VALUES IN CASE CONSTRUCTS

```
Dim MyStuff As String
MyStuff = UCase(Left(Text1.Text, 1))
Select Case MyStuff
    Case Is < ÒCÓ ÔA or B
        MsgBox ÒBefore CÓ
    Case ÒDÓ, ÒEÓ ÔD or E
        MsgBox ÒD, EÓ
    Case Is < ÒKÓ, ÒZÓ
        MsgBox Ò<K, ZÓ
    Case ÒKÓ To ÒMÓ
        MsgBox ÒK-MÓ
    Case ÒNÓ To ÒQÓ, ÒSÓ
        MsgBox ÒN-Q, SÓ
End Select
```

Although the Case construct isn't as flexible as an If...ElseIf construct, you can make it more pliable with these evaluation options.

Case Else , or "None of the Above"

Case Else is optional and is inserted as the last Case statement. It represents a "none of the above" option where none of the previous Case statements satisfied the evaluating variable or expression, as illustrated in Listing E.52.

LISTING E.52

USING CASE ELSE

```
Select Case iUserChoice
    Case 0
        Ôtake some action
    Case 1
        Ôtake some other action
    Ôother Case statements could go here...
    Case Else
        MsgBox ÒN/AÓ
End Select
```

You will almost always want to use a Case Else , because you want to explicitly handle this "none of the above" situation in your code.

Looping

VB offers the programmer a veritable cornucopia of looping constructs (at least seven at last count). Although you could get by with just one looping construct, it is nice to know that there is a looping construct for every need.

Do While...Loop for a Loop that Might Never Run

The Do While construct is useful when you want code to run only while a given condition is True .

The first line of a Do While construct begins with the words Do While followed by a logical condition (that is, an expression that evaluates to True or False).

All the lines of code between the Do While and the Loop statements will execute as long as the condition in the Do While is True . The loop terminates when the initial condition evaluates to False . Because the condition is at the beginning of the Do Loop , it is possible that program control will never pass through the loop if the initial condition is evaluated to False . Listing E.53 gives an example.

LISTING E.53

A DO WHILE...LOOP CONSTRUCT

```
Do While intOptions > 0
    . . .
    intOptions = intOptions - 1
Loop
```

The loop will run if the initial value of intOptions is 12, but will never run if the initial value of intOptions is Ð4.

Do...Loop While for a Loop that Runs at Least Once

If VB only offered the Do While... construct, you would have to go through some contortions if there were a loop that always needed to run at least a single time. You might have to "prime the loop" by artificially contriving to make the Do While condition True before the loop began.

For this type of situation, you can just include the condition within the Loop statement at the end of the loop:

```
Do
    . . .
    intChoice = MsgBox(ÒDo another one?Ó,
vbYesNo)
Loop While intChoice = vbYes
```

In this way, the condition will not be evaluated for the first time until the loop has already run once.

Do Until...Loop for a Loop on a Negative Condition

Many structured programming purists think that having to use a negative condition in a loop's While clause makes a loop harder to understand. To address this concern, VB provides the Do Until construct, which runs as long as the condition specified after the word Until is *not* True . That is, the loop terminates as soon as the condition is evaluated to True , as shown in Listing E.54.

LISTING E.54

DO UNTIL...LOOP

```
Do Until datQuestions.RecordSet.EOF
    . . .
    datQuestions.RecordSet.MoveNext
Loop
```

The same example with a negative condition in a Do While clause would read as in Listing E.55.

LISTING E.55

DO WHILE WITH A NEGATIVE CONDITION

```
Do While NOT datQuestions.RecordSet.EOF
    . . .
    datQuestions.RecordSet.MoveNext
Loop
```

Notice that the Do Until... construct is easier to read and understand.

Do...Loop Until for a Negative Condition Loop to Run at Least Once

As with the Do While loop, you can place the Until condition after the Loop keyword, as illustrated in Listing E.56.

LISTING E.56

DO...LOOP UNTIL

```
Do
    . . .
    strUserResponse = Trim(txtResponse.Text)
Loop Until strUserResponse = ÒÓ
```

This ensures that VB will always run the loop once before the Do...Loop Until statement gets a chance to evaluate the condition for the first time.

For...Next to Implement a Counter Loop

In contrast to the Do_Loop, which iterates based on a condition, the For_Next loop iterates through the loop a specified number of times. Therefore, a For_Next loop requires a counter variable to track the number of loop iterations.

The programmer must initialize a loop counter variable (usually an Integer type), which will be incremented upon each pass through the loop. The structure of the For statement looks like this:

```
For lcv = StartValue To StopValue
```

lcv variable, a common name in VB programming documentation, stands for "loop counter variable." Other customary names for such variables that you will find in a lot of existing code are i, j, iCount, iCounter, and so forth. This counter variable tracks the iterations through the loop. StartValue and StopValue are the starting and ending values for the loop counter variable. VB will automatically increment the loop counter variable by one on each successive pass through the For_Next loop.

The Next statement ends the For loop. You can put the name of the loop counter variable after the Next statement, as shown in Listing E.57. This improves readability, but is optional.

LISTING E.57

A FOR...NEXT LOOP THAT NAMES THE LOOP COUNTER IN THE NEXT CLAUSE

```
Dim lcv as Integer
For lcv = 1 To 10
    msgBox ÒSquare of Ò & lcv & Ò is Ò _
     & lcv * lcv
Next lcv    Ôuse of lcv name is optional here
```

The start and stop values in the For statement can be literal values typed directly into your code, as in the preceding example, or one or both of them can be a variable or constant:

```
For intCounter = 1 To intMaX
```

By default, the increment of the loop counter variable is 1. If you want to increment by some value other than 1, you can specify a Step quantity, using the following optional syntax:

```
For lcv = StartValue To StopValue Step
Increment
```

The value of Step can represent a negative or even a fractional number:

```
For intCounter = 10 To 0 Step Ð1
      txtCountDown.text = Str(intCounter)
Next intCounter
```

Because VB changes the loop counter variable each time it encounters the Next statement, the loop counter variable will increment (or decrement if Step is a negative value) one more time than there are passes through the loop. Therefore, in the code of Listing E.58, the value of intCounter displayed in the final line will be 11.

LISTING E.58

CODE THAT TESTS THE FINAL VALUE OF A LOOP COUNTER VARIABLE

```
Dim intCounter as Integer
For intCounter = 1 To 10
      msgBox ÒSquare of Ò & intCounter & Ò is Ò _
       & intCounter * intCounter
Next intCounter
MsgBox Òlcv = Ò & intCounter
```

For Each...Next to Loop Through Objects in a Collection or Array

You can traverse all the objects in a collection or array without knowing either their names or the number of objects. The For Each_Next construct takes a placeholder variable, similar to the For_Next loop. In contrast to the For_Next loop, however, this placeholder does not represent a counter, but rather an object of the type found in the collection. The object points to each element of the collection as you iterate through the loop. This enables you to evaluate and manipulate any of the collection's elements. The format of the For Each statement is this:

```
For Each ObjVarName in CollectionName
```

Just as you must first declare a variable to represent the counter in a For_Next loop, so you must first declare a variable of the appropriate object type to represent the For Each_Next loop's placeholder, as in Listing E.59.

LISTING E.59

A FOR EACH...NEXT LOOP

```
Dim ctrlCurrControl as Control
For Each ctrlCurrControl in Controls
      If TypeOf ctrlCurrControl Is TextBox Then
            ctrlCurrControl.Text = ÒÓ
      EndIf
Next ctrlCurrControl
```

Once again, notice that you can reference the placeholder's name after the word Next in the loop's final line. As is the case with the For_Next loop, this usage is optional. See the section titled "Working with Collections."

If you wanted to use a For Each loop to traverse an array, you need to declare a placeholder variable of type Variant and then loop through the array using that variable, as shown in Listing E.60.

LISTING E.60

TRAVERSING AN ARRAY WITH FOR EACH

```
ÔAssume we have an array of String
Ôcalled Names()
Dim varCurrent as Variant
For Each varCurrent in Names
    Me.Print varCurrent
EndIf
Next varCurrent
```

WARNING

For...Each Loop Control Variables for Arrays Must Be Variant If you declare the control variable to loop through an array as some type other than Variant, you will generate a compile error.

Control variables for collections of objects don't have to be of type Variant.

Terminating a Loop Abruptly with Exit

You can stop the currently executing loop in its tracks with an Exit statement. If you are in the middle of one of the Do loops (Do While/Until _Loop, Do_Loop While/Until), the statement

```
Exit Do
```

will cause VB to go immediately to the first line following the end of the current loop.

WARNING

Exit Do Inside While...Wend You can't use Exit Do inside the obsolete While...Wend loop construct. If you try to terminate a While...Wend loop with Exit Do, you will generate a compiler error. See the discussion of While...Wend in the corresponding section.

To exit a For_Each or For_Next loop, use the following statement:

```
Exit For
```

As with Exit Do, VB will immediately begin processing the first line after the For_Next or For_Each loop.

WARNING

Watch Use of Exit in Looping Use Exit in looping structures sparingly, because Exit can make your code harder to follow at maintenance time.

The With...End With Construct

Although the With_End With construct is not technically a control structure (it doesn't redirect flow of execution), it does affect all the lines of code inside it. This construct has two advantages—it saves you some typing and it helps your code to execute faster. With...End With can result in an enormous speed increase in a program that uses a lot of object references.

You can use this construct when you need to call several methods or properties belonging to the same object, such as a Form, Control, or other type of object (for example, an OLE server or a user-defined class). You begin the construct with this line:

```
With ObjectName
```

Then, everything between this line and the End With can refer to one of the object's properties without requiring you to type the object name. Instead of typing ObjectName.Method/Property, you need only to type .Method/Property inside the With_End With construct. You could, for example, replace the code in Listing E.61 with the code in Listing E.62.

LISTING E.61

FULL OBJECT REFERENCES IN EVERY LINE OF CODE

```
frmMain.Caption = ÒHello, WorldÓ
frmMain.Top = 0
frmMain.Show
```

LISTING E.62

USING WITH...END WITH TO REDUCE TYPING

```
With frmMain
    .Caption = ÒHello, WorldÓ
    .Top = 0
    .Show
End With
```

You can nest With constructs within each other. Therefore, in Listing E.63, you can refer to a Data Control's Recordset object using a nested With construct within the With construct for the Data Control:

LISTING E.63

NESTED WITH...END WITH CONSTRUCTS

```
With Data1
    .Caption = ÒCategoriesÓ
    .RecordSetType = dbOpenTable
    .RecordSource = ÒCategoriesÓ
    .Refresh
    With Data1.RecordSet
        .MoveFirst
        .Index = ÒIDÓ
    End With
End With
```

Notice that, even though the Recordset object belongs to the Data Control and appears inside the Data Control's With construct, you still had to make full reference to the Recordset (including the Data Control) when you started its own nested With construct.

Obsolete Techniques

There are many ways to modify flow-of-control in VB. Some of them are included for downward compatibility with earlier versions of BASIC and aren't recommended by Microsoft for new development. The two techniques discussed here are from earlier versions of the BASIC language—before VB came on the scene.

EXAM TIP

Watch Out for Obsolete Techniques on the Exams These obsolete constructs are included in this book because some of the examples in the certification exam might use them.

The While...Wend Construct

The While...Wend construct is similar to a Do...Loop with limitations. First, you are limited to placing the condition at the top of the loop. Second, the While...Wend construct does not recognize any type of Exit device. That is, While...Wend constructs do not enable you to exit the loop early.

In Listing E.64, you can see that the While...Wend construct most closely resembles a Do While...Loop construct.

LISTING E.64

WHILE...WEND

```
While intOptions > 0
    . . .
    intOptions = intOptions - 1
Wend
```

WARNING

Exit...Do **and** While...Wend If While...Wend you invoke Exit Do within a While...Wend construct, you will generate a compiler error.

The GoTo Statement

There was a time in the history of programming when just about the only control the programmer had over flow of execution was to evaluate a condition and then jump to another place in the program based on the result of the evaluation.

Constructs such as the If construct, the Case construct, and the various flavors of While and For that we find in languages like VB arose out of the difficulty programmers had in maintaining code with a lot of jumping back and forth.

As a throwback to this earlier time, VB still allows limited ability to jump immediately to another line in your application. You can use the GoTo statement to do this. Here are the rules for using GoTo:

- You can only jump to a location inside the current routine.
- You must jump to a named label. A named label is a line in VB that contains a unique name followed by a colon.

The first line in Listing E.65 directs the flow of execution to the label JumpPoint . The intervening code is not executed.

LISTING E.65

JUMPING AROUND WITH GoTo

```
GoTo JumpPoint
    Ô. . .this code gets skipped
JumpPoint:
    Ô. . .do some stuff after the jump
```

NOTE

Use of GoTo to Implement Error Handlers GoTo is used in VB to construct error handlers. Therefore, a statement like

```
On Error GoTo JumpPoint
```

is a standard approach to constructing VB error handlers.

APPENDIX F

Suggested Readings and Resources

Appelman, Dan. *Dan Appelman's Developing Active X/COM Components with Visual Basic 6*. Que, 1998.

Brierley, Eric. *Waite Group's Visual Basic 6 How-To*. Sams Publishing, 1998.

Cadman, John. *Waite Group's COM/DCOM Primer Plus*. Sams Publishing, 1998.

Conley, John. *Sams Teach Yourself OOP with Visual Basic in 21 Days*. Sams Publishing, 1998.

Fronckowiak, John. *Sams Teach Yourself OLE DB and ADO in 21 Days*. Sams Publishing, 1997.

Jennings, Roger. *Roger Jennings' Database Developer's Guide with Visual Basic 6*. Sams Publishing, 1998.

Jerke, Noel. *Waite Group's Visual Basic 6 Client/Server How-To*. Sams Publishing, 1998.

Jung, David. *Waite Group's Visual Basic 6 SuperBible*. Sams Publishing, 1998.

Kurata, Deborah. *Doing Objects in Visual Basic 6*. Que, 1998.

Mauer, Lowell. *Sams Teach Yourself More Visual Basic 6 in 21 Days*. Sams Publishing, 1998.

McManus, Jeffrey. *Database Access with Visual Basic*. Que, 1998.

McManus, Jeffrey. *Jeffrey McManus' Database Access with Visual Basic 6*. Sams Publishing, 1998.

Microsoft Corporation. *Microsoft® Visual Basic® 6.0 Programmer's Guide*. Microsoft Press, 1998.

Microsoft Corporation. *Microsoft® Visual Basic® 6.0 Reference Library*. Microsoft Press, 1998. Three-volume set includes the *Language Reference*, *Controls Reference*, and *Component Tools Guide*.

Norton, Peter. *Peter Norton's Guide to Visual Basic 6*. Sams Publishing, 1998.

Perry, Greg. *Sams Teach Yourself Visual Basic 6 in 21 Days*. Sams Publishing, 1998.

Perry, Greg. *Sams Teach Yourself Visual Basic 6 in 24 Hours*. Sams Publishing, 1998.

Rahmel, Dan. *Sams Teach Yourself Database Programming with Visual Basic 6 in 24 Hours*. Sams Publishing, 1998.

Reselman, Bob. *Using Visual Basic 6*. Que, 1998.

Sample Applications, *Microsoft® Visual Basic® 6.0* Assuming a default VB 6 directory structure, you should have a Samples directory that contains a number of subdirectories containing sample applications.

Shea, Brian. *Waite Group's Visual Basic Source Code Library*. Sams Publishing, 1998.

Sherrif, Paul. *Paul Sheriff Teaches Visual Basic 6*. Que, 1998.

Siler, Brian and Spotts, Jeff. *Special Edition Using Visual Basic 6*. Que, 1998.

Siler, Brian. *Visual Basic 6 Companion*. Que, 1998.

Smith, Curtis. *Sams Teach Yourself Database Programming with Visual Basic 6 in 21 Days*. Sams Publishing, 1998.

Spenik, Mark. *Visual Basic 6 Interactive Course*. Waite Group Press, 1998.

Thayer, Rob. *Visual Basic 6 Unleashed*. Sams Publishing, 1998.

Vaughn, William R. *Hitchhiker's Guide to Visual Basic and SQL Server, Sixth Edition*. Microsoft Press. 1998.

Webb, Jeff. *Microsoft® Visual Basic® 6.0 Developer's Workshop, Fifth Edition*. Microsoft Press, 1998.

Winemiller, Eric. *Waite Group's Visual Basic 6 Database How-To*. Sams Publishing, 1998.

www.cgvb.com . *Carl and Gary's Visual Basic Web Page*. A one-stop resource on the Web for VB developers. Contains lots of information as well as links to many other VB resources on the Web, including the various VB-related sites under microsoft.com.

www.microsoft.com . Search this site for the MSDN Library (which contains a complete reference to VB 6 and the Visual Studio 6.0) and the latest information and downloads on ADO, ActiveX, COM, and other VB-related standards and tools.

www.microsoft.com/workshop/author/htmlhelp/default.asp . You can obtain the latest version of the HTML Help Workshop here as well as links to other useful information about HTML Help.

Index

A

accelerator keys, *see* access keys
access keys, 82
  TextBox control, 98-99
accessing Object Browser, 453
Action menu commands, Export, 770
actions, ADO Data Control Recordset, 328
Activate event, 239-246
active context, Watch expressions, 852
Active Document
  applications
    creating, 690-691
    debugging, 722-724
    testing in VB environment, 722-724
    .vbd file, 706-707
  compiling, 724
  components, 686
  container applications
    behavior of, 692
    document navigation, 727-728
    executing, 687-688
    Microsoft Internet Explorer, 687
    Microsoft Office Binder, 687
    navigation code, 718-719
  containers, 693
  custom events, 704
  custom methods, 705
  custom properties, 705-706
  data preservation events, 707-708
  distributing, 724
  DLL documents, 691-692
  events, 694-698
  exam objectives, 683, 685
  EXE documents, 691-692
  Help menu, 714
  HTML Package and Deployment Wizard, 725-730
  Hyperlink objects, 716-717
  implementation overview, 688-689
  menus, 712-714
  methods, 697-712
  Microsoft's future intentions, 688
  modeless forms, 715-716
  multiple UserDocument objects, 719-722
  persistent properties, writing, 728-729
  properties, asynchronous downloading, 708-709
  Properties bag, 707-708
  scrolling, 698-701
  services, 686
  UserDocument objects, 689
  ViewPort, 701-704
  Web pages, 725-730
active learning, study strategies, 1,040
Active Server Pages (ASP), dynamic Web pages, 1,017-1,018
ActiveX
  Data Objects, *see* ADO
  DLLs, 903-905
    setup programs, 747-749
  Document Container (COM client), 448
  extension of OLE standard, 514-515
  Microsoft COM specifications, 445
  objects, 513
  OLE precursor, 449
  services, 450
ActiveX controls
  Alarm method, 625
  author, 615
  CanPropertyChange method, 647

component class features, 612
constituent controls, 613
building from, 639
delegated methods, 641
creating, 670
Fast Facts, 1,023-1,025
creation process, 616
currency amount entry example, 667-668
custom events, 625
custom methods, 624
data source controls, 647-650
Data Sources, 673-675
data-aware, 645
data-bound fields, 645-646
debugging, 661-670
breakpoints, 911
debugging via project groups, 907-909
declaring events, 626
default user interface event, 626-627
design-time features, 615
developer, 615
error trapping, 906
FoundOne event, 626
ImageList overview, 127
Initialize event, 631
InitProperties event, 631-634
licensing, 171-172, 613
ListView, 139-140
maintenance, 614
Microsoft Windows Common Controls Library 6.0, 128
.ocx file extension, 128
OCX files, 613
overview, 612
as project elements, 613
properties, 627-636, 671-672
Property Bag, 634-638
Property Pages, 652-661, 672
RaiseEvent statement, 626
ReadProperties event, 631
referencing, 170-171
retrieving persistent property values, 638
runtime features, 615
siting instances on containers, 628
StatusBar, 153
Terminate event, 632
testing, 661-670
threads, managing, 542
toolbar, 147-148
ToolBox
adding, 128
TreeView, 134
user-drawn, 614
Paint event, 623
UserControl container, 616-620
WriteProperties event, 632
adaptive-form exams, 1,042-1,046
Add Class Module command (Project menu), 521
Add File dialog box, 54
Add-ins, Package and Deployment Wizard, 747
Add method
Controls Collection, 162-163, 195-197
dependent collection class, 536
ImageList control, 131
ListView control, 142-143
TreeView control, 135-136
Add Procedure dialog box, 1,145
Add User dialog box, 51
Add Users and Groups to Role dialog box, 783
adding
Active controls to ToolBox, 128
Active Documents to Web pages, 729-730
ActiveX controls, 1,017
Class modules, 521
components to MTS packages, 775-777, 1,027
controls
Controls Collection, 168-169
via control arrays, 160-162, 193-194
via Controls Collection, 162-163, 195-197
controls to forms, 91-92
graphics, 130
MTS components to packages, 788
pop-up menus to applications, 85-89

projects to project groups, 900, 910
records
ADO Data Control setup, 325-326
Recordsets, 342-343
registered components, 776-777
VCM to Visual Basic toolbar, 553-554
AddNew method, adding Records to Recordsets, 342
adDoAddNew value (EOFAction property), 323
Administrator role, System package (MTS), 744
administrators, Visual SourceSafe, 45-51
ADO (ActiveX Data Objects), 305
cursors, 400, 403
Data Control, 318, 329-330
Connection object, 330-351
manipulating, 1,029
Recordsets, 352-356
data-access models, 378-394
data-binding tools, 308-317
Errors collection, 357
object model, 306-307
objects, 407
OLE DB data providers, 305
ADO Errors collection, handling, 1,031
adReason parameter
WillChangeRecord event, 350
WillChangeRecordset event, 351
WillMove event, 349
adStatus As ADODB.EventStatusEnum parameter (BeginTransComplete transaction method), 408
adStatus parameter
EndofRecordset event, 348
WillChangeField event, 350
WillChangeRecord event, 350
AdStavEOF value (EOFAction property), 323
adUseServer value (CursorLocation property), 401
advanced optimization, native code, 937-939
advantages of P-code, 927
Alarm method, ActiveX controls, 625
algorithms, testing in conditional blocks, 954
alignment property, 102
Allow Unrounded Floating-Point Operations option, 940
AllowCustomize property (ToolBar control), 152-153
Ambient object, 621-623
AmbientChanged event, 621
ampersand (&), 98-99
ampersand, double (&&), 103
answering simulation questions, 1,047
apartment-model threading, 541-542
appearance properties of menus, 83
Appearance property, 102
ToolBar control, 150-151
appearances of menus, 84-85
applications
Active Document
creating, 690-691
debugging, 722-724
DLL document, 691-692
EXE document, 691-692
existing applications, 690
multiple UserDocument objects, 719-722
testing in VB environment, 722-724
UserDocument objects, 689
.vbd file, 706-707
client, 451-452
conditional compilation, 1,033
deploying, 987-999
methods, 1,036-1,037
Package and Deployment Wizard, 1,035
developing, 18-19
VB Enterprise Development Model, 20-28
DHTML, 818-825
existing, 690
IIS WebClass, 799-817
loan processing case study, 465-465
MDI, 248-249
menus, 89
multitier, 15
n-tier, 27-32
online user Help, 265-269
HelpFile property, 270
HTML Help, 276-292
identifying files at design time, 269
identifying files at runtime, 270
WinHelp, 267-268

pop-up menus, 85-89
removing from projects, 985
returning records to, 396-399
see also menus
ApplyChanges event, ActiveX controls, 657
archive Þles, 969
archiving Visual SourceSafe databases, 48-49
arguments
AsyncRead method, 709-710
AsyncReadComplete method, 711-712
Button, 87
Find method (locating records), 346
passing to General procedures, 1,146-1,155
Shift, 87
Arrange property (ListView control), 145
array arguments, 1,152-1,153
arrays, 1,139
astrAlphabet, 860
astrName, 846
bounds, 1,140
dynamic, 1,139
elements, 1,142
menu controls, 89
multidimensional, 1,140-1,141
static, 1,139
Watch expressions, 860
arrows (Menu Editor), 83
Asc( ) function, data type conversion, 1,134
ASP (Active Server Pages), 800-804
Assert method, Debug object, 871
assertion failures, 872
assigning roles to MTS components, 784-792
assignment loops, debugging, 847-848
assume No Aliasing option, native code, 937
astrAlphabet array, debugging, 860
astrName array, debugging, 846
asynchronous
callbacks, 573-574
downloading, 709-712
processing, 1,020
AsyncRead method (Active Document), arguments, 709-710
AsyncReadComplete method (Active Document), arguments, 711-712
attributes, DHTML Web pages, 823-824
author, ActiveX controls, 615
authorization checking, MTS components, 785-786
authorization tracking, role-based security, 754
automatic data type conversion, 1,132-1,134
automatic program redeployment, 995
AutoRedraw property 242
AutoSize property, 103
StatusBar control, 156
availability
load balancing design implications, 33
logical design impact on physical design, 28

B

BackColor property, 100
Background Compilation switch, projects, 941
BackStyle property, 102
basic optimizations, native code, 929
BAT Þles, setup packages, 974
BeginTrans method, database transactions, 405
BeginTransComplete event, ADO Connection object, 333
BeginTransComplete method, 407
bidirectional text display, VB programs, 982
binding
ADO, 313-314
controls to ADO Data Control Recordset, 323-325
object variables, 456-458
binding and naming services, COM, 449
bindings, Data Sources to ActiveX controls, 673-674
BOFAction property, ADO Data Control Setup, 322-323
Bookmark property, Recordsets, 347-348
Boolean data types, 1,129-1,130
BorderStyle property, 102
bounds, specifying, 1,140
Branch dialog box, 64
branching (controls), 1,158-1,160

Break in Class Modules command, error handling, 480
Break mode, 854-863
see also Watch expressions
Break On All Errors command, error handling, 480
Break on Errors settings, debugging ActiveX controls, 912
Break On Unhandled Errors command, error handling, 481
Break When Value Changes Watch expression, 851
breakpoints
debugging ActiveX controls, 906-911
debugging programs, 855
browsers, testing ActiveX controls, 662
bugs in code
preventing, 846
see also debugging
building client applications, 451-452
projects, 902
business objects, 21-22
COM components, 518
business rules, 518-519
business-logic rules, 1,022-1,023
business-logic tier (COM components), 518-519
Button, 86-87
as Integer parameter, 107
ButtonClick event (ToolBar control), 151-152
buttons, 92-105
Buttons Collection (ToolBar control), 149-150
ButtonWidth property (ToolBar control), 149
Byte data types, 1,129

C

Cab Þles (cabinet Þles), 969
Call Stack window, tracing procedures, 888
Call statements, calling Sub procedures, 1,154
callback
objects, 577-594
procedures, 1,020
calling
MTS components from VB clients, 768-769
MTS objects, 780
procedures, 1,147
Sub procedures, 1,154-1,156
Cancel parameters, 243-245
Cancel property, 99
CancelAsyncRead method, 710
CancelDisplay parameter, ADO Data Control Error event, 329
canceling Recordsets, 341
CancelUpdate method, ADO Data Control Recordset, 327
CanPropertyChange method, ActiveX controls, 647
Caption property, 99, 103
captions, menus, 82
Case Else (Case statements), 1,160-1,161
Case statements, 1,159-1,161
case studies
data entry forms, 225-226
Displaying Customer Sales Information, 825-826
forms, 250-253
loan processing application, 465-466
Sales-Order Entry System, 34-35
CausesValidation property, 219-220
CBool function, data type conversion, 1,134-1,135
CD-ROM
applications, 1,036
contents, 1,107
electronic version of text, 1,107
program deployment, 999
Top Score test engine, 1,107-1,115
program deployment, 999
CDbl function, data type conversion, 1,135
CDs, deploying programs, 990
centralization (maintainability), 24
certiÞcation requirements, 1,099-1,106
Fields parameter (WillChangeField event), 350
Change events, 104-109
field-level validation 220
TextBox control, 642
ChangeFieldComplete event, 351
character cases, 213
checking
files in Visual SourceSafe projects, 56-57
variable data types, 1,131-1,132

Child property (TreeView control), 137
children (TreeView control), 134
Children property (TreeView control), 138
chm extension, HTML Help Þles, 267
Choose Project to Archive dialog box, 48
choosing ADO data-access models, 383-384
Chr( ) function, data type conversion, 1,134
Classes, 520-541
Clear method
  Err object, 485, 501
  ListBox control, 641
Click events, 83, 105-109
client applications
  asynchronous call notifications, 573-574
  callback objects, 573-577
  COM components, 451-452
  errors, 549-552
  in process server components (COM), 516
  Interface class, 567-572
  out-of-process server components (COM), 515
client speciÞc features, conditional compilation, 955
client-side cursors, 401
clients
  ActiveX Document Container, 448
  COM, 448
  configuring
    DCOM, 986
    with MTS components, 768-772
  DCOM, 1,036
  MTS client packages, 770
  MTS components
    calling, 768-769
    configuring, 789-790
    package installation/upgrades, 769-770
  MTS setup packages, creating, 789
  sending HTML text directly to programming WebClasses, 805-807
CLng function, data type conversion, 1,135
Close on the History dialog box, 63
closing
  Break mode, 859
  project groups, 901
codes
  Click event procedures of menus, 83
  CommandButton control, 99
  debugging, 855-859
  dynamic menus, 90
  error handling, 492-501
  forcing errors in code, 499
  in controls, 103-110
  in event procedures, 104-105
  menu appearances, 84
  MouseUp event procedure, 87
  pop-up menus, 85-88
  properties within, 94-95
  runtime menu items, 90-91
  simulating errors in code, 499
  tokenizing, 924
  see also listings; programs
coding, 697-712
container applications, merging, 714
Collapse event (TreeView control), 139
collections, 307, 1,142
ColumnClick event (ListView control), 147
ColumnHeaders Collection (ListView control), 146
COM (Component Object Model), 445-472
  Class modules, 520
  classes, 544-546
  components, 514-519
  error results, 590-591
  exam, 509-512
  GlobalSingleUse instancing, 547-548
  IDispatch interface, 585-586
  in process server, 515-516
  information, 513
  instancing, 590
  invoking, 1,019-1,020
  IUknown interface, 585-586
  messages, 595
  MultiUse instancing, 548
  object model, 519-520
  out-of-process server, 515-516
  procedures, 517-518
  programming, 513-514

publishing, 591-592
registering, 579-595
scalability, 584-585
SingleUse instancing, 546-547
threading model, 590
threads, 541-542
types, 583-584
unregistering, 579-595
user interfaces, 581
VB code, 1,026
VCM, 559-561
VCM component functions, 552
vtable binding, 585-586
CommandButton control, 105
command buttons, 92-103
Command objects, ADO object model, 307
adding to Data Environment Designer, 309-312
initializing, 334-335
Recordsets, 335-351
CommandButtons, 99-104
Properties, 96-98
commands
Action menu, 770
Debug menu, 858-859
Debug.Print, 889
Project menu
Add Class Module, 521
References, 452
scripting (COM automation), 448
CommitTrans method, database transactions, 405
CommitTransComplete event, ADO Connection object, 333
Compile on Demand option, 941
compiled
languages, 920
programs, 921
compilers, 919-926
compiling
Active Document, 724
to native code, 931-940
to P-code, 926-927
projects, 902
background compilation, 941
conditional compilation, 942
on demand, 941
with Class modules, 1,027-1,029
Complete events, 351
Component Object Model, see COM
Component wizard (MTS), adding, 776-777
components, 445
ActiveX controls, 612
adding to packages, 746
characteristics, 445
DCOM, 986
designing for *n*-tier applications, 29-32
events, 463-472
Object Library, 452
objects, 460-462
standalone applications, 451
components (MTS)
authorization checking, 785-786
clients, 789-790
developing, 772-773
in-process, 768
installing from packages, 769-770
multiple interfaces, 785
out-of-process, 768
packages, 775-788
roles, 784-792
transaction properties, 777-780
transactional options, 790-791
updating from packages, 769-770
Components dialog box, 308
composite controls, 613
conceptual design, 15, 20
developing, 1,007
Development model, 20-21
conditional blocks, 954
conditional compilation, 1,033
client specific features, 955
projects, 942
conditional compiler blocks, 952-954
conditional ßow control directives, 943

conÞguring
client computers, 1,011
clients, 789-790
DCOM, 1,036
on client/server computers, 986
dependent class, 534
MTS
for server computer, 1,010
from VB clients, 768-772
ConnectComplete event, ADO Connection object, 333
connection objects, 331-334
ADO, 309-312
ADO object model, 307
Const preprocessor directive (vbConst), 948
constants vbObjectError, 486
constituent controls, 613, 639-643
Contain relationships (business objects), 22
container applications
Active Document
behavior of, 692
testing applications, 723-724
documents, 716-717
examples, 687
executing, 687-688
Microsoft Internet Explorer, 687
object models, 718
container objects, see forms
containers, detecting, 693
content, DHTML Web pages, 823-824
contents Þle, HTML Help Workshop, 288-289
context options, Watch expressions, 852
context-sensitive help, 271-276
context-sensitive menus, 80-89
context-sensitive Topic Þles, 290
ContinuousScroll property (Active Document), 700-701
control arrays
controls, 160-162, 193-194
labels, 161-162
control containers, Ambient object, 621
controlling instancing in COM components, 590
controls
ActiveX, 450
creating, 670
see also ActiveX controls
ADO Data Control, 318, 329-349
branching, 1,158-1,161
Change events, 109
Click events, 105-106
code assigned, 103-110
CommandButton, 99-105
composite controls, 613
constituent, 639
constituent controls, 613
context-sensitive Help, 271-276
control arrays, 160-162, 193-194
Controls Collection, 168-169
data source, 647-650
DblClick events, 106
Default event procedures, 104
enabling based on input, 222-224
event procedure naming, 1,157-1,158
forms, 91-92
GotFocus events, 110
GoTo statements, 1,166
intrinsic, 164-165
Keystroke events, 110
Labels, 102-106
ListBox, 641
looping, 1,161-1,164
LostFocus events, 110
Mouse events, 108-109
MouseDown events, 106-108
MouseMove events, 109
MouseUp events, 106-108
names, 104-105
non-intrinsic, 165-167
properties, 94
referencing, 170-171
TextBox, 98-106
users, 914
While...Wend constructs, 1,165
with pop-up menus, 88-89
With...End With constructs, 1,164-1,165
Controls Collection
controls, 168-169
forms, 158-159
resetting fields, 190

methods, Add, 162-163, 195-197
ProgID, 163-164
Controls Collections, licensing, 171-172
contstants, 244
converting
existing applications, 690
variable data types, 1,132-1,136
core exams, 1,103
Count property
dependent collection class, 535
Forms Collection, 180
CPUs, machine code, 921
Creatable objects, 460
CREATE PROCEDURE query, stored procedures, 385-387
Create Project dialog box, 54
CreateObject, 801
CreateObject function, 458-460
creating
Active Document applications, 690-691
Active documents, 1,025
ActiveX controls, 1,023-1,025
callback procedures, 1,020
client setup packages, 789
COM components, 517-518
dialog boxes, 1,012-1,014
dynamic Web pages, 1,017-1,018
forms, 186
Interface class, 563-564
MTS client packages, 770
MTS components, 772-773
transactional options, 790-791
MTS packages, 1,027
objects, 469-470
roles for MTS packages, 791-792
MTS security, 782-783
server component applications, 517-518
StatusBar control, 189
toolbars, 188-189
Web pages, DHTML Page Designer, 1,018-1,019
cRecords parameter (WriteChangeRecord event), 350
Criterion argument, Find method, 346
CSng function, data type conversion, 1,135
CTL Þles, ActiveX controls, 613
currency amount entry example control, 667-668
Currency data type, 1,126
current system states, usage scenarios, Development model conceptual design, 20
cursors
ADO data-access, 400-403
executing statements without, 394, 396
options, 414-415
custom controls
building from constituent controls, 639
CanPropertyChange method, 647
constituent controls, 641
currency amount entry control, 667-668
data source controls, 647-650
data-aware, 645
Default user interface event, 626-627
events, 625
Initialize event, 631
InitProperties event, 631-634
methods, 624
properties, 627-636
Property Bag, 634
ReadProperties event, 631
retrieving persistent property values, 638
siting instances on containers, 628
Terminate event, 632
user-drawn, 623
WriteProperties event, 632
see also ActiveX controls
custom events
Active Document, 704
ActiveX controls, 625
Class modules, 528
WebItems, 814-815
Custom installation, MTS, 743
custom methods, 705
ActiveX controls, 624
custom properties
Active Document, 705-706
ActiveX controls, 627-636
Class modules, 524

Customize Toolbar dialog box, 152-153
customizing, 984-985
CVar( ) function, data type conversion, 1,135

D

data
  locking, 413-414
  structured storage services, 449
  types, 1,125-1,130, 1,146
  variables, 1,117
    arrays, 1,139-1,142
    checking data types, 1,131-1,132
    converting data types, 1,132-1,136
    scope, declaring and defining, 1,117-1,124
    strings, 1,136-1,138
data access components, 1,009
data binding, 1,019
Data control (ADO), 318
  programming, 329-356
  setting up, 319-328
  versus Data Environments, 318-319
data cursors, ADO object model, 306
Data Environment Designer, 308-317
data input forms
  exam objectives, 125-126
  reset fields, 158-159
    Controls Collection, 190
Data Link Properties dialog box, 320
data source controls, 647-650
data sources, 674-675, 1,029-1,030
data-access tier (COM components), 518-519
data-aware ActiveX controls, 645
data-binding tools (ADO), 308-317
data-bound Þelds, ActiveX controls, 645-646
databases
  error handling, 1,031
  stored procedures, 384-399
  transactions, 404-407
    recoverability of, 1,031
  Visual SourceSafe, 45-51
DataChange property 224
DataField property, ActiveX controls, 645-646
DataFormat property 224
DataSource property, ActiveX controls, 645-646
Date data types, 1,128-1,129
Date/Time functions, 1,128-1,129
DateTime project, 905
DblClick events, 106-109
DBMSs, 404-414
DCOM (Distributed Component Object Model), 986
  implementing, 998
  registering, 1,035
DDF Þles, building CAB Þles, 975
DeActivate event, versus PostFocus event, 246
DeActivate event, 240
debug code, conditional compiler blocks, 952-954
Debug menu, 857-859
Debug object, 865-874
debug status messages, 865-869
Debug.Assert method, saving breakpoints between sessions, 871
Debug.Print command, 889
debugging, 846
  ActiveX controls, 661-662, 670
    breakpoints, 911
    via project groups, 907-912
    what to look for, 666-667
  ActiveX DLLs via project groups, 903-905
  applications (Active Document), 722-724
  assignment loops, 847-848
  astrAlphabet array, 860
  astrName array, 846
  breakpoints, 871
  code, 855-859
  display loops, 847
  displaying data values, 869, 871
  procedures, 888
  project groups, 906
  projects, 849-850
  VB code, 1,026
  Watch expressions, 881-882

Decimal subtype (Variant data type), 1,130-1,131
declarative security (MTS), 784
declaring
  arrays, 1,139
  Class objects, 538-539
  compiler constants, 948-951
  data types, 1,146
  events, 528-529, 626
  intrinsic controls, 164-165
  non-intrinsic controls, 165-167
  objects, 455-456
  variable scope, 1,117-1,124
defaults
  lower bounds, 1,140
  properties, 94
  user interface events, 626-627
  values, 632
deÞning
  Callback objects, 574-575
  custom methods, 705
  menus, 712
  methods, 532-533
  object variables, 468
  pop-up menus, 85-86
  properties, 532-533
  variable scope, 1,117-1,124
  Watch variable, 1,034-1,035
delegated
  events, 642
  methods, 641
  properties, 630
Delete function, 876
Delete method, 343
DELETE statement, executing with stored procedures, 393
deleting
  controls, 160-162, 193-194
  records, 343
  Watch expressions, 850
delivering programs to clients, 987-999
DEP Þles (dependency Þles), 969, 981-984
dependency, ADO object model, 306
dependency Þles, 748, 978-981
dependent classes, 533-537
dependent objects, 460
deploying applications, 1,035-1,037
  CD method, 990
  CD-ROM method, 999
  floppy disk method, 988-998
  network method, 990
  updates, 994
  Web method, 992-999
deriving
  logical from conceptual designs (Development model), 21-22
  physical from logical designs (Development model), 22-23
descending For É Each loops, 183-184
Description parameter, ADO Data Control Error event, 328
Description property, Err object, 483
design
  conceptual, 15, 20
  logical, 20-22
  physical, 22-28
  specifications, 225-226, 253
  time, 84
design time, 94
  ActiveX controls, 615
  templates, 89
designing
  components for *n*-tier applications, 29-32
  data access components, 1,009
  menus, 712-713
  MTS components, 1,026
  properties, 1,009
desktop applications
  error handling, 1,021-1,022
  online help, 1,020-1,021
  Visual Basic installation/configuration, 1,010
detecting
  containers, 693
  object variables, 463
developers, ActiveX controls, 615
developing conceptual design, 1,007

Development Model (VB Enterprise), 20-28
DHTML (Dynamic HTML), 818-825
  Page Designer, 1,018-1,019
dialog boxes
  Add File, 54
  Add Procedure, 1,145
  Add User, 51
  Add Users and Groups to Role, 783
  Branch, 64
  Choose Project to Archive, 48
  Close on the History, 63
  Components, 308
  Create Project, 54
  creating, 1,012-1,014
  Customize Toolbar, 152-153
  Data Link Properties, 320
  exam objectives, 125-126
  Export Package, 770
  File Browse, 808
  Font property, 97
  History of Project, 62
  Label, 58
  Menu Editor, 81
  Project Components, 128
  Project History Options, 62
  Project Properties, 269, 805
  Property Pages, 320
    ToolBar control, 148
    Use of ImageList control, 129
  PropertyPages, 322
  RecordSource, 322
  References, 453
  Save, 806
  Share, 63
  Share From, 63
directives
  flow control (pseudocode), 923
  preprocessor, 943-948
disconnect event, ADO Connection object, 334
disconnected Recordsets, 352-353
display loops, debugging, 846
displaying
  Active document information, 1,022
  current variable values, 878-879
  data, 1,019
  Immediate window, 864
  information (TreeView control), 134
  modeless forms, 715-716
  pop-up menus, 88
  ViewPort properties, 702-703
DisplayName property, Ambient object, 622
distributed applications, 1,021-1,022
distributing Active Documents, 724
DLL documents (Active Document), selection criteria, 691-692
DLL Þles, VB programs, 925
Do Until loops, 1,161-1,162
docucentric operating systems, Microsoft Active Document future, 688
Document Container (ActiveX), 448
documents
  ActiveX, 450
  container applications, 716-717
Draw method (ImageList control), 132
dynamic
  arrays, 1,139-1,142
  controls, 160-162, 193-194
  cursors, 404, 1,031
  events, 816-817
  load balancing, 33
  menus, 80, 90
  Recordsets, 354-355
  Web pages, 1,017-1,018
Dynamic HTML, see DHTML
dynamically modifying menu appearances, 84-85

E

early binding
  IDE support, 457
  Microsoft Excel support, 458
  object variables, 456
early bound variables, 468-469

editing
canceling bound control changes, 327
existing records, 326-327
Package and Deployment scripts, 969
Property Pages, 653
Watch expressions, 850
efficiency, logical design impact on physical design, 24
ElapsedTime( ) function, 957
elective exams
MCSE certification, 1,102
MCSE+Internet certification, 1,103
elements
arrays, 1,142
DHTML Web pages, 824-825
Else and ElseIf construct, 1,158-1,159
Else preprocessor directive (#Else), 943-945
empty packages, creating with MTS Explorer, 752
Enabled property, 96
enabling
authorization checking, 785-786
controls, 222-224
VCM, 553-554
encapsulation (maintainability), 25
End If preprocessor directive (#End If), 943-945
EndofRecordset event, 348
ADO Data Control, 329
EnsureVisible method (TreeView control), 136-137
EnterFocus event (Active Document), 695
Enterprise Application Model, 18-19
VB Enterprise Development Model, 20-28
EntryText property, constituent controls, 643
enumerating values (Case statements), 1,160
EOFAction property, ADO Data Control Setup, 322-323
equi joins, 1,032
Err object, 481
Clear method, 485, 501
properties, 482-484
Raise method, 485-486
Error event, 328
Error statement, error handling, 499
error tracking, user controls, 914
error trapping, debugging ActiveX controls, 906
errors, 590-591
Errors collection
ADO, 357
Command ADO object, 307
establishing source-code version control, 1,009-1,010
event procedures
controls, 103-110
default for controls, 104
naming, 1,157-1,158
private variables in (code listing), 1,124
events
Active Document, 700-708
ActiveX controls, 625-633, 657
ADO, 317
Connection object, 332-333
Data Control, 328-329
AmbientChanged, 621
built-in, 530-531
Change, 104-109
Class modules, 528-529
Class objects
handling, 540-541
raising, 529-530
Click, 104-109
components, 1,009
data source controls, 648
DblClick, 106-109
DHTML, 821
EnterFocus (Active Document), 695
ExitFocus (Active Document), 698
field-level validation, 218-222
forms, 239-249
GotFocus, 110
handling from COM components, 463-472
Hide (Active Document), 698
Initialize (Active Document), 694
InitProperties (Active Document), 695
Keystroke, 110, 216-217
ListView control, 146-147
LostFocus, 110
mouse, 106-109
n-tier application components, 31-32

procedures, 86-87
ReadProperties (Active Document), 696
Recordset , 348-351
Show (Active Document), 695
Terminate (Active Document), 698
TextBox control, 642
ToolBar control, 151-152
TreeView control, 139, 139
WebItem, 814-817
WithEvents keyword, 464-465, 471-472
WriteProperties (Active Document), 697
exams
core, 1,103
electives, 1,102
formats, 1,041-1,046
objectives
Active Document, 683, 685
COM components, 447, 509-510
data input forms, 125-126
MTS applications, 765
pre-exam tips, 1,045
question types, 1,044-1,047
study tips, 1,039-1,041
taking, 1,046
EXE applications, adding, 521
EXE document (Active Document) selection criteria, 691-692
EXE Þles, P-code, 924
Execute Direct Model
ADO data-access, 379, 383
data sources, 1,029-1,030
ExecuteComplete event, ADO Connection object, 334
executing
container applications, 687-688
procedures, 875-876
exercises, 228-233
existing
applications, 690
packages, 755-757
Exit Function (terminating procedures), 1,153-1,154
Exit statements, terminating loops, 1,164
Exit Sub (terminating procedures), 1,153
ExitFocus event (Active Document), 698
exiting
Break mode, 859
project groups, 901
Expand event (TreeView control), 139
Expanded property (TreeView control), 138
Explorer
Visual SourceSafe, 51-65
Visual SourceSafe (workstations), 46
Export Package dialog box, 770
exporting packages, 755-757
exposed objects, COM components, 451
Expression not deÞned in context message , 852
expressions
preprocessor, 945
Watch, *see* Watch expressions
Extender object, 619-623
extensibility, 26
externally
classes (creatable), 544-546
objects (creatable), 513
extracting information from Date variables (Date/Time functions), 1,128

F

fast code optimization, native code, 929-930
Favor Pentium Pro option, 932
feature sets, conditional compilation, 955
FieldChangeComplete event (ADO Data Control), 329
Þelds
contents, 337-339
validation, 219-222
validation, 217
Fields parameter (WillChangeField event), 350
File Browse dialog box, 808
Þle delete function, 876
Þle I/O test, Optimize project, 934
File menu, 90
FileName classings, 459

Þles
CAB (cabinet), 969
checking in/out, 56-57
CTL, 613
DDF, 975
DEP, 969, 981
HTML, 976
HTML Help, 276-291
INF, 977
OBJ extension, 921
OCX, 613
Optimize.VBP, 932
PDM, 969
SETUP.EXE, 978
SETUP.LST, 979-980
VBG extension, 900
VBR, 987
Visual SourceSafe projects, 54
Find method, 346
Þnding COM components, 559-561
FindItem method (ListView control), 144
FirstSibling property (TreeView control), 137
Þxed-form exams, 1,042-1,046
Flash Cards application, 1,113-1,115
ßoppy disks
applications, 1,036
program deployment, 988, 998
ßow control directives
conditional, 943
pseudocode, 923
focus, Activate event 242
folders, Visual SourceSafe projects, 53-55
Font property, 96-97
For Each loops, 182-184
Controls Collection, 158
For Each ...Next loops, 1,163-1,164
For Next loops, 181-184
Controls Collection, 157
forcing errors in code, 499
Format function, 1,135-1,136
formats, MCP exams, 1,041-1,046
forms
case studies
design specs, 225-226
Load event procedures, 250-253
context-sensitive Help, 271-275
controls, 91-94
creating, 186
data processing, 1,015-1,017
debugging, 852
events, 239-249
hiding, 176-177
keystroke events, 212-217
loaded status, 182-183
loading, 173-174, 190-192
managing, 581-583
methods, 247-250
reset fields, 158-159
Controls Collection, 190
showing, 176-177
statements, 247-248
storage, 177-179
unloading, 173-174, 190-192
Forms Collection, 183-184
Forms Collection
contents, 179-180
forms, 179-184
item numbers, 180-181
loaded status, 182-183
looping techniques, 181-182
properties, 180
Forward-Only cursor, 403, 1,031
For_Next loops, 1,162-1,163
FoundOne event, 626
Friend keyword, 533
Friend scope, 1,157
.FRM Þle (forms storage), 177-179
FullPath property (TreeView control), 138
functions
checking variable data types, 1,131-1,132
CreateObject, 458-459
object creation, 469-470
data type conversion, 1,134-1,136
Date data type, 1,128-1,129

Delete, 876
ElapsedTime( ) , 957
error-handling example, 496-499
Error( ) , 499
file delete, 876
General procedure, 1,145-1,152
GetObject , 458-460
object references, 470
GetProcessTimes( ) , 958
GetTickCount( ) , 957
Lbound( ) , 939
Len( ) , 869
return values, 1,155-1,156
Spc( ) , 867
string manipulation, 1,136-1,138
Tab( ) , 867
Timer( ) , 957
TypeName, 693

G

General procedures (Sub procedures), 1,144-1,145
Generalize relationships (business objects), 22
GetDataMember event, 648
GetObject function, 458-460
GetProcessTimes( ) function, 958
GetTickCount( ) function, 957
GetVisibleCount method (TreeView control), 136-137
global scope, 884
global variables, 885
GlobalSingleUse instancing, 547-548
GotFocus event, 239
in controls, 110
debugging, 880
field-level validation, 221-222
versus Activate event, 246
Goto 0 clause, 489
GoTo statements (controls), 1,166
graphics, 129-133

H

handling
errors, 549-552
events, 463-472
in Class objects, 540-541
Height property, 97
help, online user, 265-269
context-sensitive, 271-276
HelpFile property, 270
HTML Help, 267-292
identifying files at design time, 269
identifying files at runtime, 270
WinHelp , 267-268
Help menu, merging, 714
HelpContext
ID property, 280-285
parameter, 329
property, 484
HelpÞle
parameters, 328
properties, 270, 484
Hide event (Active Document), 698
Hide method, 247-250
Hide statement (forms), 176-177
HideSelection property, 100
hiding forms, 176-177
hierarchy
of error handling, 494
menus, 82
History of Project dialog box, 62
hlp extensions, WinHelp Þles, 268
host pages, ASP, 804
HScrollSmallChange property (Active Document), 700
HTML (Hypertext Markup Language), 799-810
DHTML applications, 818-825
files, 976
Help files, 276-292
file structures, 276-277
Help Workshop, 277-292

extensive help source files, 280-285
Index file, 286
Package and Deployment Wizard, 725-730
Project header file, 279-280
WhatsThis source files, 289-292
Hyperlink object, Internet-aware applications, 716-717
Hypertext Markup Language, see HTML

I

icons
ListView control, 140
property (ListView control), 144
identifying
business objects (development model logical designs), 22
Help files, 268-270
Identity setting, Role-based security, 754
IDispatch interface, COM components, 585-586
If preprocessor directive, Def(), 943-945
If statement, branching, 1,158-1,159
If É End If construct, 1,158
ignoring errors in code, 501
IIS WebClass applications, 799-817
ImageList control
forms, 186
graphics, 129-130
ListImages Collection, 130-131
overview, 127
properties, 133
Property Pages dialog box, 129
ImageWidth property (ImageList control), 133
Immediate window, 864-880
Immediate windows, opening, 864
implementing
Active Document applications, 688-689
basic operations, 187-188
business logic rules in COM components, 588-589
business rules, 518-519
business-logic rules in COM components, 1,022-1,023
callbacks
in Class objects, 529-530
objects, 575-577, 592-594
Class modules with COM components, 520
COM components in client applications, 451-452
custom events in Class modules, 528
error-handling desktop/distributed applications, 1,021-1,022
Interface class, 565-572
load balancing, 1,036
navigational design, 1,011-1,012
object model, 519-520
online help, 1,020-1,021
polymorphism, 561-563
properties, 524-525
implicit loading, 249
importing packages, 755-757
in process
components (MTS), 768
server components (COM), 515-516
forms, 582-583
threads, 542
Index
files, 286
property, 89-90, 142
INF Þles, 977
information, displaying, 134
Initialize event
Class modules, 530-531
custom controls, 631-634
Initialize event (Active Document), 240, 694
initializing object variables, 468
InitProperties event, 631-634
Active Document, 695
inline error handling, 489, 500
input forms, 1,015-1,017
input validation 209-210
Insert menu, creating General procedures, 1,144-1,14
INSERT statement, executing with stored procedures, 391
installing
MTS components, 741-743
from packages, 769-770

VB programs, 983
Visual Basic for desktop/distributed applications, 1,010
Visual SourceSafe, 46
instances, releasing, 463
instancing
COM components, 584-585
controlling in COM components, 590
externally creatable classes, 546
Instancing property, 544-546
instantiating
Class objects, 538
object variables, 463
objects, 455-459
projects, 902
Instr( ) function, string manipulation, 1,137
integration, 26
interfaces
class, 563-572
MTS components, 785
Internet
browser-aware applications, 716-717
deploying programs, 992
Internet Explorer, 662
Internet Package, 748
Internet setup packages, 969-976, 996
interpreted languages, 920
interpreted programs, 921-922
Initialize event, 239
intrinsic controls, declaring, 164-165
InvisibleAtRunTime property, UserControl container, 620
invoking COM components, 1,019-1,020
IsArray( ) function, checking variable data types, 1,131
IsMissing( ) function, passing arguments to General procedures, 1,151-1,152
IsNumeric( ) function, checking variable data types, 1,131
Item method (dependent collection class), wrapper method, writing, 537
item numbers, referencing, 180-181
ItemClick event (ListView control), 146
IUknown interface, COM components, 585-586

J-K

JOIN clauses, connecting tables, 412-413
joins, 1,032

key combinations, assigning, 83
key events, debugging, 880
Key property (ListView control), 142
KeyAscii parameter, 212-213
KeyCode parameter, 214
KeyDown event, 214-216
KeyPress event, 212-213
KeyPreview property, 216-217
keys
access, 82
Controls Collection (license), 171-172
testing in Shift parameter, 87
Keyset cursor, 404, 1,031
keystroke events
in controls, 110
enabling with KeyPreview property, 216-217
KeyDown and KeyUp, 214-216
KeyPress , 212-213
KeyUp event, 214-216
keywords
Date data type, 1,128
ParamArray (array arguments), 1,152-1,153
Private , 882

L

Label control, 104-106
Label dialog box, 58
labeling Visual SourceSafe projects, 58
labels
access keys, 98-99
control arrays, 161-162
properties, 92-103
LastDLLError property, Err object, 484
LastSibling property (TreeView control), 137

late binding, 456-458
late bound variables versus early bound variables, 468-469
Lbound( ) function, 939
Left function, string manipulation, 1,136
Left property, 97
Len function, string manipulation, 1,137
Len( ) function, Debug object, 869
levels of menus, 82
Library packages, MTS components, 773
licensing
  ActiveX controls, 171-172, 613
  composite controls, 614
limiting
  users, MTS packages, 781
  Watch expressions, 885
LineStyle property (TreeView control), 138
ListBox control, Clear method, 641
ListImages Collection (ImageList control), 130-133
listings
  ADO data-access models, 380-383
  Alarm method, ActiveX controls, 625
  ApplyChanges event, 657
  arguments, 1,152-1,155
  ASP file, 800-802
  calling CanPropertyChange method, 647
  calling PropertyChanged method, 637
  Case statements, 1,160
  Change event, TextBox control, 642
  checking container property changes, 621
  controls
    GoTo statement, 1,166
    populating from copy buffer, 339
  cursors, 394-395
  customized Start event in new WebClass, 807
  Data Environment Designer, 316
  database transactions, 406
  date and time functions (Date data type), 1,129
  Decimal subtype (Variant data type), 1,130
  delegating Clear method of ListBox, 642
  design-time writeable property example, 622
  EditProperty event procedure, 660
ElseIf statements, 1,159
Exit Sub, 1,153-1,154
file delete function, 876
functions
  CBool, 1,134
  CByte, 1,135
  Format, 1,136
  Instr, 1,137
  IsMissing( ), 1,151-1,152
  Len, 1,137
  Parsing, 1,138
  TypeName( ), 1,131
General procedure functions, 1,146-1,147
GetDataMember event procedure, 649
Immediate window data display, 870
implementing custom control properties, 630
initializing delegated properties, 644
initializing properties via InitProperties event procedure, 633
Internet setup package, HTML code, 977
local variables, 1,120-1,123
loops, 1,161-1,163
managing persistent delegated properties, 644
MoveNext/MovePrevious methods, 345
numeric data types, 1,132
On Error Resume Next, ignoring errors via, 501
On Error Resume Next statement, 500
parts of typical dependency file, 982
passing General arguments, 1,155
private variables, 1,124
procedures
  General, 1,144
  Private scope, 1,156
  Public scope, 1,157
ProcessTag event procedure, 811
Property Let/Get procedures for delegated properties, 644
property value persistence via WriteProperties, 636
public variables, 1,124
records, 345
Recordsets
  adding and saving records, 342
  ADO, 336

canceling user changes, 341
CursorType property, 403
deleting records, 343
disconnected, 353-354
dynamic, 355
Find method, 347
persistent, 356
SQL statements, 409
updating records, 341
redrawing UserControl graphics via Paint event, 624
resizing constituent controls, 640
restoring persistent property values, 638
SelectionChanged event procedure, 656
SETUP.LST file example, 980
Standard HTML code, 810
static variables, 1,120-1,123
stored procedures, 392-399
String data, 1,127
type, 1,133
variable scopes, declaring, 1,119
WebClass Start code, 815
WebClass Start event procedure fragment (imbedding WebItem references), 812
WebItem dynamic events, 817
While …Wend construct, 1,165
ListItems Collection (ListView control), 141
ListView control, 139-147
literals, 946-947
load balancing, 16
implementing, 1,036
servers, 32-33
Load event, 239-252
Load statement, 89
forms, 174
versus Show method, 247-248
loading
forms, 173-174, 190-192
from memory, 249
graphics, 130
labels, 161-162
loan processing application case study, 465
local scope, Watch expressions, 882
local variables, 881
declaring scope, 1,120, 1,122
Watch expressions, 886
LocaleID property, Ambient object, 621
Locals window, 878-879
modifying program values at runtime, 889
locating
ProgID Controls Collection, 163-164
records, 346
locations, 400-402
Locked property, 101
locking data, 413-414
logging errors, 491
logical design, 15, 20-28
loop counters, local variables as (code listing), 1,122
looping, 1,161-1,164
Controls Collection, 157-158
Forms Collection, 181-182
loops, 847-848
LostFocus event, 240
in controls, 110
debugging, 880
field-level validation, 221-222
versus DeActivate event, 246

M

machine code, 920-921
macros, setup program install paths, 973
main project, 902-910
maintainability, 24-25
maintenance
ActiveX controls, 614
user (Visual SourceSafe Administrator), 51
managing
Active Document events, 695-698
forms, 581-583
threads, 541-544
manipulating
Callback objects, 577-579
Class objects, 539

data, 1,019
data sources, 1,029-1,030
objects, 461-462
String data types, 1,136-1,138
manual program redeployment, 995
mapping
files, 283-290
users, 783-784
MaxLength property, 101, 224
MCP (Microsoft CertiÞed Professional), 1099
exams, 1,041-1,047
MCP+Internet (Microsoft CertiÞed Professional+Internet), 1,099
MCP+Site Building certiÞcation requirements, 1,101
MCP+Site Building (Microsoft CertiÞed Professional+Site Building), 1,099
MCSD (Microsoft CertiÞed Solution Developer), 1,100
MCSE (Microsoft CertiÞed Systems Engineer), 1,099
MCSE+Internet (Microsoft CertiÞed Systems Engineer+Internet), 1,100
MCT (Microsoft CertiÞed Trainer), 1,100
MDI applications, 248-249
memory forms, 176-177
loading forms from, 249
menu bars, 80
Menu Editor, 80-83
menus, 81-90
Active Document, 712-714
design considerations, 716
merging
file versions, 65
Help menus with container applications, 714
messages, 595
meta-learning, 1,041
methods, 276
Active Document, 709-712
ActiveX controls
Alarm , 625
CanPropertyChange , 647
Class modules, 523-524
Class objects, 539
classifications, 532-533
constituent controls, 641
context-sensitive Help, 274
Controls Collection, 162-163, 195-197
database transaction, 405-407
Debug.Assert , 871
forms, 247-250
ImageList control, 131-133
ListBox control, 641
ListView control, 142-144
n-tier application components, 31
objects, 461-462
PopupMenu, 85-86, 88
Property Bag, 634-638
ReadProperty (Active Document), 696
ToolBar control, 151
TreeView control, 135-137
ViewPort , 704
WriteProperty (Active Document), 697
Microsoft
Active Document future, 688
ActiveX
development, 514
OLE precursor, 449
COM specifications in ActiveX standard, 445
Excel, 458
Internet Explorer, 687
Office Binder, 687
Office Suite COM components, 451
OLE development, 514
Microsoft Training and CertiÞcation Web site, 1,100
Microsoft Transaction Server, see MTS
Microsoft Windows Common Controls Library 6.0, 128
mid function, 1,136-1,137
minimal installation, MTS, 742
MidWidth property (Active Document), 699-700
menu preÞx (menus), 82
modeless forms, 715-716
models
ADO (ActiveX Data Objects), 306-307
ADO data-access, 378-398
n-tier encapsulation, 25
VB Enterprise Development, 20-28

modifying menu appearances, 84-85
module scope, Watch expressions, 883
module variables, 882
  Watch expressions, 886
modules, debugging, 853
monikers, naming and binding services (COM), 449
monitoring array values, see Watch expressions
mouse
  Button parameter, 86
  buttons, 86-87
  Click events in controls, 105-106
  DblClick events in controls, 106
  Mouse events in controls, 108-109
  MouseDown events in controls, 106-108
  MouseMove events in controls, 109
  MouseUp events in controls, 106-108
Mouse button, 105
mouse events
  in controls, 108-109
  debugging, 880
MouseDown event procedures, testing, 106-108
MouseUp event procedures, 86-87, 106-108
  testing, 107
Move method, navigating recordsets, 344
MoveComplete event, 351
  ADO Data Control, 329
moving applications, 27-28
MTS (Microsoft Transaction Server), 739-754, 1,010
  client packages, 770, 789
  components
    authorization checking, 785-786
    calling from VB clients, 768-769
    client computer configuration, 1,011
    client configuration, 789-790
    configuring from VB clients, 768-772
    designing, 1,026
    developing, 772-773
    in-process, 768
    Library packages, 773
    multiple interfaces, 785
    out-of-process, 768
    package additions, 788
    package installation, 769-770
    role assignments, 784-785, 792
    Server packages, 773
    transaction properties, 777-780
    transactional options, 790
  exam objectives, 765
  Explorer
    packages, 750-757
    functions, 765
  objects, 780
  packages
    component additions, 775-777
  roles, 792
  runtime environment, 773-775
  security, 781-784
  server computers, 1,010
  stateful objects, 774
multidimensional arrays, 1,140-1,141
MultiLine property, 101
multiple arguments, 1,149
multiple instances, 533-534
multiple project setup, 904-905
multiple UserDocument objects, Active Document, 719-722
multitier applications, 15
  data access components, 1,009
  see also n-tier
MultiUse instancing, COM components, 548

N

n-tier
  applications, 27-32
  models, 25
Name property, 104
  Class modules, 521-522
  CommandButtons, 95-96
named arguments, passing to General procedures, 1,150-1,151
names of controls, changing, 104-105
naming
  binding service, 449
  event procedures, 1,157-1,158

menus, 82
packages with MTS Explorer, 752-753
native code
compiling, 928-940
differences from P-code, 920
machine code, 921
native-language compiler, 919
navigating
between DHTML pages, 821-822
container applications, 718-719
documents, 716-728
object models, 718
Recordsets, 344-345
user services, 1,011-1,012
NegotiatePosition property (Active Document), 713-714
nested database transactions, 406-407
networks
applications, 1,036
deploying programs, 990-999
New keyword, objects, 455-458
no optimization, native code, 931
NodeClick event (TreeView control), 139
nodes (TreeView control), 134-136
Nodes property (TreeView control), 138
non-intrinsic controls, 165-167
nonstring data types, string concatenation, 1,133-1,134
Now keyword, Date data type, 1,128
Number property, Err object, 482
numeric data types
automatic string value conversion, 1,132-1,133
Currency, 1,126
Optimize project, 936

O

OBJ Þle extension, 921
Object Browser, 452-455
COM components, 513
object Þles, compiled programs, 921
Object Library, 452
object models
COM components, 519-520
container applications, 718
ObjectEvent procedure, declaring, 165-167
objects
ADO (ActiveX Data Objects), 305
binding to Data Environment objects, 313-314
Command, 334-351
Connection, 307-312, 330-334
cursors, 400-404
Data Control, 318-356
data-access models, 378-398
data-binding tools, 308-317
Errors collection, 357
object model, 306-307
OLE DB data providers, 305
Recordset, 307
Ambient, 621-623
business
COM components, 518
development model logical designs, 21-22
identifying, 22
callbacks, 573-574
collections, 533-534
COM components, 453-455
component classes, 455-456
creatable, 460
creating CreateObject function, 469-470
Debug, 865
declarations, 455-456
declaring, 538-539
Dependent, 460
dependent class, 534
dependent collection class, 534-535
Err, 481
events, 540-541
exposed, 451
Extender, 623
GlobalSingleUse instancing, 547-548
hierarchy, 460-461, 513
instances, 463
instantiating, 538
CreateObject function, 458-459

late bound variables, 468-469
management service, 449
methods, 461-462, 539
MTS
calling, 780
context, 774
multiple instances, 533-534
Parent class, 537
persistence, 449
polymorphism, 561-563
properties, 461-462, 539
Public , 460
Public Not Creatable , 460
Recordsets , 352-356
reference counting, 449
references
Object Library, 452
obtaining with GetObject function, 470
sharing, 548
SingleUse instancing, 546-547
variables, 456-468
WebClass (IIS), 803-811
obtaining object references, 470
OCX Ples
ActiveX controls, 613
file extension (ActiveX controls), 128
ODBC Resource Dispenser, 774-775
OLE (Object Linking and Embedding), 449
ActiveX development, 514-515
OLE DB, data providers, 305
On Error Resume Next statement, 500
On Error statement, 489-495
online help
applications, 265-287
context-sensitive, 271-275
desktop/distributed applications, 1,020-1,021
operating systems, Microsoft Active Document concept, 688
operators, 946
Optimize project, 932-936
Timer( ) function, 957
optimizing
compiler, 919
native code, 929-940
Option Base statement, array default lower bounds, 1,140
Option Pack, 741-742
optional arguments, passing to General procedures, 1,151-1,152
Order By clauses, SQL Select statements, 411
out-of-process server components, 515-516
forms, 581
server classes, 549
threads, 542-544
outer joins, 1,032
outlines, 1,040
Overlay method (ImageList control), 132-133
overview of ActiveX controls, 612
Own relationships (business objects), 22

P

P-code, 919-927
Package and Deployment Wizard, 966-978
ActiveX DLL setup programs, 747-749
packages
Internet setup, 996
limiting usage, 781
MTS, 739-757
adding components, 788
role-based security, 1,027
role creation, 791-792
standard setup, 996
System security, 744
Page Designer (DHTML), 818-825
Paint event, user-drawn ActiveX controls, 623
Panels Collections (StatusBar control), 153-156
ParamArray keyword, passing to General procedures, 1,152-1,153
parameters
ADO Data Control events, 328
BeginTransComplete transaction method, 407
Button, 86
Button as Integer, 107
events, 348-351

KeyPress event, 212-213
Process Tag event procedure, 811
QueryUnload event, 243-245
Shift testing keys, 87
Shift as Integer, 107
Parameters collection (Command ADO object), 307
Parent class, object collection, 533-534, 537
Parent property (TreeView control), 137
parsing string manipulation example, 1,137-1,138
passing error result code client applications, 550-551
PasswordChar property, 101
PathSeparator property (TreeView control), 138
pConnection As ADODB.Connection parameter, 408
PDM Þles, 969
performance
load balancing design implications, 33
logical design impact on physical design, 24
pError As ADODB.Error parameter, 408
persistence
ActiveX control properties, 631-636, 671-672
compiler constants, 951
control properties, 629
data source controls, 650
persistent properties, 706-729
persistent Recordsets, 356
physical design, 20
deriving, 1,008
Development model, 22-28
physical keystrokes, detecting, 216
Picture Box control versus newer CoolBar control, 147-148
Picture property (StatusBar control), 156
pinning earlier versions of Þles, 61-64
polymorphism, 563-567
pop-level menus, 81
pop-up menus, 80-89
PopupMenu method, 88
positioning menus, 713-714
practice exams, Top Score test engine (CD-ROM), 1,107-1,112
pre-exam preparation tips, 1,045
pre-testing study strategies, 1,041
predeÞned compiler constants, 947-948
preferences, compiler settings, 926
preÞxes, programming Web classes, 811
preparation, exams, 1,039-1,047
Prepare/Execute Model, 1,030
preprocessor
directives, 943-948
expressions, 945
Print method, 866, 874
Private compiler constants, 951
private instancing, 545
Private keyword
methods, 532
module variables, 882
Private scope, Sub procedures, 1,156
private variables, declaring scope, 1,123-1,124
procedure-level variables, see local variables
procedures, 1,143
ADO, 317
calling passing arguments to General procedures, 1,147
debugging
Call Stack window, 888
Watch expressions, 853
error handling, 487
events
assigning code to controls, 103-110
changing control names, 104-105
naming, 1,157-1,158
executing Immediate window, 875-876
menus, 85
Mouse event, 105
MouseUp events, 86-87
properties, 525
Property, 629
stored, 384-399
Sub, 1,143-1,157
testing Immediate window, 875-876
processing form data, 1,015-1,017
processors, machine code, 921
ProcessTag event procedure code list, 811

proÞles, user, 20
ProgID, locating, 163-164
programmatic security (MTS), 784
programming
  ADO Data Control, 329-351
    versus Data Environments, 318-319
  COM components, 513-514
  Web classes, 804-811
  WebItems, 812-817
  within Data Environment Designer, 314-317
programs
  compiled, 921
  debugging, 855-859
  deploying, 987-999
  error handling, 479-501
  forcing errors in code, 499
  interpreted, 921-922
  removing from projects, 985
  simulating errors in code, 499
Project Components dialog box, 128
Project Explorer, setting startup project, 903
project groups, 899-912
Project History Options dialog box, 62
Project menu commands
  Add Class Module, 521
  References, 452
Project Properties dialog box, 269, 805
  declaring compiler constants, 950
projects
  ActiveX controls, 670
    UserControl container, 616-620
  adding or removing from project groups, 900
  adding to project groups, 910
  Background Compilation switch, 941
  building, 902
  class modules, 1,027-1,029
  Compile on Demand option, 941
  compiling, 902
    conditional compilation, 942
    native code, 928-940
    to P-code, 926-927
  DateTime, multiple project setup, 905
  DCOM components, registering, 986
  debugging, Watch expressions, 849-850
  implementing DCOM, 998
  including ActiveX controls, 613
  multiple setup, 904-905
  Setup1, 985
  startup, 902-903
  testing ActiveX controls, 663-667
  Visual SourceSafe, 46-64
properties
  Active Document, 699-714
  ActiveX controls
    connecting to Property Pages, 658
    connecting to standard Property Pages, 661
    custom, 627-628
    DataField, 645-646
    Default values, 632
    delegated, 630
    editing complex properties, 660
    flagging changes, 656
    persistence, 629-636, 671-672
    saving changes, 657
  ADO Data Control setup, 322-323
  Alignment, 102
  AutoRedraw, 242
  AutoSize, 103
  BackColor, 100
  BorderStyle, 102
  Cancel, 99
  Caption, 99-103
  Class objects, 539
  classifications, 532-533
  CommandButtons, 92-103
  components, 1,009
  constituent controls, 643
  context-sensitive Help, 271-275
  controls, 94
  data source controls, 648
  Default, 100
  Err object, 482-484
  field-level validation, 219-220
  form controls, 94
  Forms Collection, 180

HelpFile, 270
HideSelection, 100
ImageList control, 133
Index, 89-90
instancing COM component classes, 544-545
keystroke event, 216-217
Label control, 102-103
Labels, 96-103
ListView control, 142-146
Locked, 101
menus, 83
modifying Immediate window, 875
MultiLine, 101
n-tier application components, 30-31
Name, 104
Class modules, 521-522
objects, 461-462
PasswordChar, 101
procedures, 525
Property Let/Get procedure, 526-528
public variables, 524-525
Recordset, 347-348
ScrollBars, 101
SelLength, 102
StatusBar control, 154-156
Style, 100
Text, 102
text boxes, 92-103
ToolBar control, 148-153
TreeView control, 137-138
UseMaskColor control, 133
UseMnemonic, 103
UserDocument.Parent, 693
UserMnemonic, 98
Validation, 224
Value, 100
ViewPort, 701-702
visible menus, 85
within code, 94-95
WordWrap, 103
Properties collection (Command ADO object), 307
Properties window, 93
Property Bag, 634-638
Property Get procedure, 630
implementing in Class modules, 524-525
Property Let procedure, 630
Property Let/Get procedure, 526-528
Property Let/Set procedure, 524
Property Pages, ActiveX controls, 652-672
Property Pages dialog box, 320
ImageList control, 129
ToolBar control, 148
Property procedures, deÞning custom control properties, 629
Property Set procedure, 630
PropertyChanged method, 634-636
PropertyPage designer, 652
PropertyPages dialog box, 322
proposed system states, usage scenarios, 21
providers, 305
pseudocode, 923
Public
but Not creatable objects, 460
compiler constants, 951-953
keywords, 532
objects, 460
methods, 523-524
scope, 1,156-1,157
variables
declaring scope, 1,124
defining custom control properties, 629
properties, 524-525
publishing
Active Document Web pages, 725-730
COM components, 555-559

Q-R

queries
data values, 874
stored procedures, 385-387
QueryUnload event, 240-248
question types, MCP exams, 1,044-1,047

Quick Watch, displaying expression values, 863
quitting
  Break mode, 859
  project groups, 901

Raise method, Err object, 485-486
RaiseEvent statement, ActiveX controls, 626
raising
  events, 528-530
  hard errors, 551-552
read allowances, Property Let /Get procedure, 526-527
reading record contents into controls, 339
ReadProperties event
  Active Document, 696
  custom controls, 631
  method, 634-638
Real Speed tests, Optimize project, 932-933
recognizing terms, study tips, 1,040
RecordChangeComplete event (ADO Data Control), 329
records
  adding ADO Data Control Setup, 325-326
  existing, 326-327
  locating, 346
  Recordsets, 342-343
  returning to applications, 396-399
  updating Recordsets, 340
  user changes, 341
  writing controls to, 340
RecordsetChangeComplete event (ADO Data Control), 329
Recordsets, 323-328
  ADO Command objects, 335-348
  cursors, 400,-404
  disconnected, 352-353
  dynamic, 354-355
  persistent, 356
  SQL statements, 408-413
RecordSource dialog box, 322
reference counting objects, COM, 449
references
  COM components, 452
  dialog box, 453
  Object Library, 452
  objects, 470
References command (Project menu), 452
referencing
  controls, 170-171
  item numbers, 180-181
Regedit utility, ProgID, 163-164
registered components, MTS Component wizard, 776-777
registering
  COM components, 579-581
  DCOM components, 986, 1,035
registers, optimizing, 938
releasing objects, 463
remote support Þles (VBR), 987
Remove Array Bounds Checks option, native code, 939
Remove Floating-Point Error Checks option, native code, 940
Remove Integer Overßow Checks option, native code, 939
Remove method
  Controls Collection, 162-163, 195-197
  dependent collection class, 536-537
  ImageList control, 131
  ListView control, 142-143
  ToolBar control, 151
  TreeView control, 135-136
Remove Safe Pentium FDIV Checks option, native code, 940
removing
  controls, 168-169
  projects from project groups, 900
  runtime menu items, 91
renaming packages with MTS Explorer, 752-753
repository databases, VCMs, 553
requirements
  case studies
    data entry forms, 225
    form Load event procedures, 252
  certification, 1,099-1,100
resetting data entry Þelds, Controls Collection, 158-159, 190
Resize event, 639
resizing
  arrays, 1,141-1,142
  constituent controls, 639

Resource Dispenser functions, 774-775
Response object, 801
restoring Visual SourceSafe databases with Administrator, 48-49
Resume Next clause, On Error statement, 489
return values
  function, 1,155-1,156
  General procedure functions, 1,146
returning error results, 590-591
reusing COM components, 559-561
right function, string manipulation, 1,136
right joins, 1,032
right-mouse menus, 80-89
role-based security, 753-754
  MTS, 781
  packages, 1,027
roles, MTS, 782-792
RollbackTrans method, 405
RollbackTransComplete event, 333
Root property (TreeView control), 138
round-robin threading, 543
routines, 487-490
runtime, 86-89
  environment (MTS), 774-775
  features, 615
  menus, 89-91

S

Save dialog box, 806
saving
  project groups, 901
  property changes, 657
scalability, 27-33
Scope parameter, RDO Data Control Error event, 328
scope
  compiler constants, 951
  Sub procedures, 1,156-1,157
  variable, 1,117-1,124
  Watch expressions, 881-890
scripting command (COM automation), 448
scripts, Package and Deployment, 969
Scroll event (Active Document), 700-703
ScrollBars property, 101
  Active Document, 699-700
scrolling Active Documents, 698-701
searchDirection argument, Find method, 346
searching
  COM component information, 453, 455
  Controls Collection, 157
security
  logical design impact on physical design, 28
  MTS, 783-786
  packages, 753-754
Select statements (SQL), 409-413
SelectedControls collection, ActiveX control Property Pages, 654-655
selecting
  COM components, 583-584
  threading model, 1,026
selection, 1,158-1,160
SelectionChanged event, ActiveX control Property Pages, 655
SelLength property, 102
SelText property, 101
sending
  messages, 595
  user messages, 581
SendTags parameters (SendTags event procedure), 811
separator bars in menus, 83
server-side cursors, 402, 1,031
servers
  Callback objects, manipulating, 577-579
  classes, 549
  component applications, 517-518
  configuring DCOM, 986
  DCOM, 1,036
  errors, 549
  load balancing, 32-33
  MTS configuration, 1,010
service classes, COM components, 544-545
Set Next command (Debug menu), 859
Set Next Statement command (Debug menu), 858

settings
custom properties, 148
references, 452
return values, 1,146
threading model, 590
transaction properties, 777-780
Watch expressions, 1,033
setting up ADO Data Controls, 319-329
Setup & Deployment wizard, COM components, 579-581
setup packages, 966-996
SETUP.EXE
installing VB applications, 983
setup routines, 978
SETUP.LST Þle, 979-984
Setup1 project, 985
SetViewPort method (Active Document), 704
Share From dialog box, 63
Shared Property Manager (ODBC Resource Dispenser), 775
sharing, objects, 548
Shift arguments, 87
Shift as Integer parameter, 107
Shift parameter, 214
shifted keystrokes, 215
shortcut key combinations, 83
Show event (Active Document), 695
Show method, 247-250
Show Next Statement command (Debug menu), 859
Show statement (forms), 176-177
showing forms, 176-177
ShowWhatsThis method, 276
siblings (TreeView control), 134
signpost debug messages, 866
SimpleText property (StatusBar control), 154
simulating errors in code, 499
simulation questions, 1,044-1,047
Simulator application, 1,115
SingleUse instancing, 546-547
siting control instances on containers, 628
sizing graphics, 129
SkipRows argument, Find method, 346
small code optimization, native code, 931
SmallIcon property (ListView control), 144
Sorted property (TreeView control), 138
SortKey property (ListView control), 145-146
source code, see listings
Source parameter, ADO Data Control Error event, 328
Source property, Err object, 483
source-codes, 1,009-1,010
branching Visual SourceSafe projects, 60-61
sharing Visual SourceSafe projects, 59
version control, 45-65
Spc( ) function, Debug object, 867
speciÞcations
case studies, 253
data entry forms, 225-226
speed, logical design impact on physical design, 24
SQL statements, 390-408
writing data joins, 1,032
writing for data retrieval, 1,031
stacks, passing arguments to General procedures, 1,147
standalone ActiveX controls, 613
standard Property Pages, ActiveX controls, 661
standard setup packages, 969-996
standards, stored procedures, 385
Start argument, Find method, 346
Start events, custom (programming WebClasses), 807
starting Pack and Deployment wizard, 968
startup code, compiled programs, 921
Startup objects, loading, 173-174
startup projects, 902-910
stateful objects (MTS), 774
statements
form, 247-248
Load, 89
SQL, 408-413
stored procedures, 390-396
Unload, 89
static
arrays, 1,139
cursors, 403, 1,031
load balancing, 32
variables, 1,120-1,123
StatusBar control

creating, 189
functions, 153
overview, 127
Panels Collection, 153-156
Properties, 153-156
Step Into command (Debug menu), 857
Step Out command (Debug menu), 858
Step Over command (Debug menu), 857
Step to Cursor command (Debug menu), 858
stepping through code, 856
debugging ActiveX controls, 907
stopping asynchronous downloads (CancelAsyncRead method), 710
stored procedures, 384-399
storing
forms, 177-179
information in Date variables (Date/Time functions), 1,128-1,129
multiple instances, 533-534
properties, 526
String data types, 1,126-1,127
automatic numeric value conversion, 1,133
string manipulation test, 934
structures
Controls, 1,158-1,166
HTML Help files, 276-277
looping, 1,161-1,164
While…Wend constructs, 1,165
With…End With constructs, 1,164-1,165
Study Cards application, 1,112-1,113
studying
strategies, 512
tips, 1,039-1,041
Style property, 100
StatusBar control, 153-156
ToolBar control, 150
Styles, DHTML Web page elements, 824-825
sub-level menus, 81
sub-menus, 81-82
Sub procedures, 1,143
callings, 1,154-1,156
Event, 1,143-1,144
General, 1,144-1,153
scope, 1,156-1,157
SubItems property (ListView control), 146
subobjects, object hierarchy, 460-461
subroutines (procedures), 1,143
event naming, 1,157-1,158
Sub, 1,143-1,157
substitution tags, 809-811
suggested reading, 1,167-1,168
Support folder, 974
Symbolic Debug Info option, 932
syntax
compiler constants, 949
preprocessor directives, 943
properties within code, 94

T

Tab Stop property, Labels, 98
Tab( ) function, Debug object, 867
TabIndex property, 98
tables, 412-413
TabStop properties, 98
TagContents parameter (ProcessTag event procedure), 811
taking exams, 1,046
templates
design-time menus, 89
HTML, 804-809
Index property, 89
Terminate events, 240-246
Active Document, 698
class modules, 530-531
custom controls, 632
terminating
General procedures, 1,153-1,154
loops, 1,164
terminology, 81-82
terms, 1,040
test projects, 663-667
testing
ActiveX controls, 661-670
applications (Active Document), 722-724
procedures, 875-876
Shift mask in MouseDown or MouseUp event procedures, 107
text, HTML, 805-807

text boxes, 92-103
Text property, 102
TextBoxes , 96-99
threading COM components, 584-585
threads, 541-543
  models
    configuration, 590
    selecting, 1,026
tier integrity, running processes without cursors, 396
timeliness, logical design impact on physical design, 24
Timer( ) function, 957
tokenizing code, 924
ToolBar controls, 148-152
toolbars, creating, 188-189
Toolbox, ActiveX controls
  adding, 128
ToolBoxBitMap property, ActiveX controls, 619
Tools menu, Menu Editor, 82
ToolTipText property, context-sensitive Help, 272-273
Top property, 97
Top Score test engine, 1,109-1,115
top-level menus, 81-82
Topic Þles, 281-290
Topic IDs, 283-290
transaction properties, 777-791
TransactionLevelAsLong parameter, 407
transactions, database, 404-406
  ADO Connection object, 407
  BeginTrans method, 405
  BeginTransComplete method, 407
  CommitTrans method, 405
  nested, 406-407
  RollbackTrans method, 405
trappable errors, 492
trapping errors, 906
TreeView control
  basic operations, 187-188
  children, 134
  events, 139
  information hierarchy, 134
  methods, 135-137
  nodes, 134-136
  properties, 137-138
TypeName function, 693
  checking variable data types, 1,131-1,132
typing General procedures (in Code window), 1,145

U

Ubound( ) function, 939
Unload events, 240-248
Unload statements, 89, 247-248
unloading, forms, 173-192
UnloadMode parameter, 244
unregistering, 579-595
UPDATE statement, 392
updates to programs, deploying, 994
updating
  applications, 1,037
  MTS components, 769-770
  records, 340
Use Connection String source connection option, ADO Data Control setup, 320
Use relationships (business objects), 22
UseMaskColor property (mageList control), 133
UseMnemonic property, 103
user-deÞned types, 861
user-drawn ActiveX controls, 614-623
user-interface tier (COM components), 518-519
UserControl container, 616-634
UserDocument objects
  configuring (Active Document applications), 689
  multiple, 719-722
UserDocument.Parent property, 693
UserMnemonic property, 98
UserMode property, 622
users
  mapping, 783-784
  messages, sending (COM components), 581, 595
  MTS packages, 781
  online help, 265-292
  performance issues, 24
  profiles, 20
  records changes, 341
  roles, 792

setup and maintenance administrator, 51
validating input 209-210
utilities, SETUP.EXE, 983

V

Validate event, 218-219
validating
field-levels, 218-222
properties, 224
user input, 209-210, 1,015
Value property, 100
values, Watch expressions, 859
variables, 1,117
arrays, 1,139-1,142
Boolean types, 1,129-1,130
collections, 1,142
data types, 1,125-1,136
displaying current values, 878-879
local, 881
module, 882
objects, 456-468
scope, 1,117-1,124
strings, 1,136-1,138
Watch expressions, 881-886
Variant data types, 1,130-1,131
variants, preprocessor constants, 954
VBObjectExtender , non-intrinsic controls, 165-167
VBP Þle extension, 900
VBR Þles, remote support Þles, 987
VBScript, 801
VCM (Visual Component Manager), 552-561
verifying loaded forms, 182-183
version control, source-code, 45-65
establishing, 1,009-1,010
version numbers, Visual SourceSafe projects, 58
ViewPort , 701-704
View property (istView control), 142
viewing information, 513
views, runtime, 86
visibility, forms, 249-250
Visible property
CommandButtons , 96
Labels , 96
Visual Basic toolbar, adding, 553-554
Visual Component Manager, 1,029
Visual SourceSafe, 45-65
VScrollSmallChange property (Active Document), 700
vtable binding (COM components), 585-586

W

Watch expressions, 850-851
active context, 852
arrays, 860
debugging code, 849-850
deleting, 850
scope, 881-890
setting, 1,033
user-defined types, 861
variables, 885
viewing values, 859
Watch variables, scope of, 1,034-1,035
Watch windows, viewing Watch expression values, 85
Web applications, deploying, 1,036
Web browsers, displaying, 1,022
Web pages
Active Document, 725-730
ASP (Active Server Pages), 799
DHTML, 818-825
Page Designer, 1,018-1,019
dynamic, 1,017-1,018
HTML (Hypertext Markup Language), 799
IIS WebClass applications, 799-817
Web sites, 1,168
WebClass applications, 799-817
WebItems, 804-817
WhatsThis source Þles, HTML Help Workshop, 289-292
WhatsThisHelp Þles, 291-292

Where clauses, SQL Select statements, 410-412
While É Wend constructs, 1,165
Width property, 97
WillChangeField event
ADO Data Control, 329
Recordsets, 349-350
WillChangeRecord events, 329-350
WillConnect event, ADO Connection object, 332
WillExecute event, ADO Connection object, 332-333
WillMove event
ADO Data Control, 329
Recordsets, 349
Win16
compiler constant, 947
threads, 541-542
Win32
compiler constant, 947
threads, 541-542
windows
Code, 103-104
Immediate, 864
Locals, *see* Locals window
menu bars, 80
Properties, 93
Watch, 848, 859
Windows NT 3.51, MCSE certiÞcation, 1,101
Windows NT 4.0, MCSE certiÞcation, 1,101-1,102
Windows Registry
COM components, 513
controls, 163-164
WinHelp ÞIes, 267-268
With É End With constructs, 1,164-1,165
WithEvents keyword
intrinsic controls, 164-165
limitations, 464-465
WordWrap property, 103
working copies, 57
working folders, 55
workloads, see load balancing
workstations, Visual SourceSafe Explorer, 46
wrapper method, writing, 535-537
wrapper routines, 570-572
write allowances, 526
WriteProperties events, 632
Active Document, 697
method, 634-636
writing
Active Document application, 718-719
controls to records, 340
persistent properties, 728-729
SQL statements, 1,031-1,032
WWW (World Wide Web), deploying programs, 992
see also Web pages

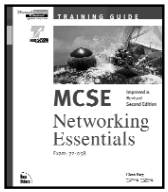
X-Z

X as Single parameter, 107

Y as Single parameter, 107

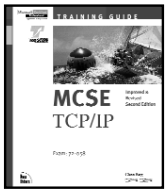
TRAINING GUIDES

NEXT GENERATION TRAINING



MCSE Training Guide: Networking Essentials, Second Edition

1-56205-919-X, $49.99, 9/98

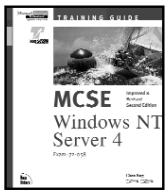


MCSE Training Guide: TCP/IP, Second Edition

1-56205-920-3, $49.99, 10/98

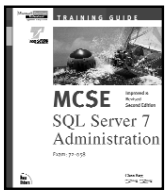
A+ Certification Training Guide

1-56205-896-7, $49.99, Q4/99



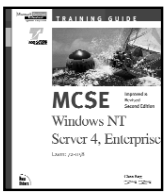
MCSE Training Guide: Windows NT Server 4, Second Edition

1-56205-916-5, $49.99, 9/98



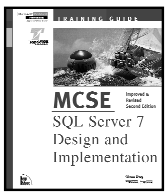
MCSE Training Guide: SQL Server 7 Administration

0-7357-0003-6, $49.99, Q2/99



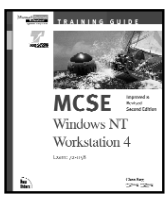
MCSE Training Guide: Windows NT Server 4 Enterprise, Second Edition

1-56205-917-3, $49.99, 9/98



MCSE Training Guide: SQL Server 7 Database Design

0-7357-0004-4, $49.99, Q2/99

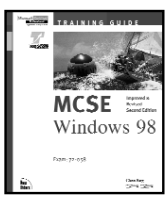


MCSE Training Guide: Windows NT Workstation 4, Second Edition

1-56205-918-1, $49.99, 9/98

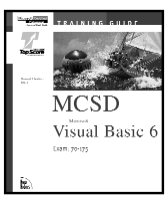
MCSD Training Guide: Solution Architectures

0-7357-0026-5, $49.99, Q2/99



MCSE Training Guide: Windows 98

1-56205-890-8, $49.99, Q2/99



MCSD Training Guide: Visual Basic 6, Exams

0-7357-0002-8, $59.99, Q3/99

TRAINING GUIDES

FIRST EDITIONS

MCSE Training Guide: Systems Management Server 1.2, 1-56205-748-0

MCSE Training Guide: SQL Server 6.5 Administration, 1-56205-726-X

MCSE Training Guide: SQL Server 6.5 Design and Implementation, 1-56205-830-4

MCSE Training Guide: Windows 95, 70-064 Exam, 1-56205-880-0

MCSE Training Guide: Exchange Server 5, 1-56205-824-X

MCSE Training Guide: Internet Explorer 4, 1-56205-889-4

MCSE Training Guide: Microsoft Exchange Server 5.5, 1-56205-899-1

MCSE Training Guide: IIS 4, 1-56205-823-1

MCSD Training Guide: Visual Basic 5, 1-56205-850-9

MCSD Training Guide: Microsoft Access, 1-56205-771-5

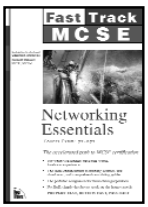
FAST TRACKS

The Accelerated Path to Certification Success

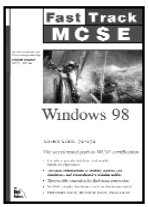
Fast Tracks provide an easy way to review the key elements of each certification technology without being bogged down with elementary-level information.

These guides are perfect for when you already have real-world, hands-on experience. They're the ideal enhancement to training courses, test simulators, and comprehensive training guides.

No fluff—imply what you really need to pass the exam!



MCSE Fast Track: Networking Essentials
1-56205-939-4,
$19.99, 9/98



MCSE Fast Track: Windows 98
0-7357-0016-8,
$19.99, 12/98

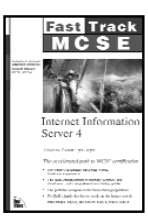
MCSE Fast Track: TCP/IP
1-56205-937-8,
$19.99, 9/98

MCSE Fast Track: Windows NT Server 4
1-56205-935-1,
$19.99, 9/98

MCSE Fast Track: Windows NT Server 4 Enterprise
1-56205-940-8,
$19.99, 9/98

MCSE Fast Track: Windows NT Workstation 4
1-56205-938-6,
$19.99, 9/98

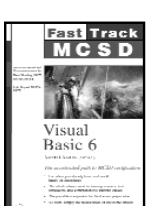
A+ Fast Track
0-7357-0028-1,
$29.99, 3/99



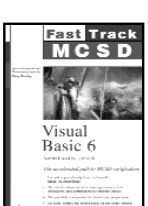
MCSE Fast Track: Internet Information Server 4
1-56205-936-X,
$19.99, 9/98

MCSE Fast Track: SQL Server 7 Administration
0-7357-0041-9,
$19.99, Q2/98

MCSE Fast Track: SQL Server 7 Database Design
0-7357-0040-0,
$19.99, Q2/98



MCSD Fast Track: Visual Basic 6, Exam 70-175
0-7357-0018-4,
$19.99, 12/98



MCSD Fast Track: Visual Basic 6, Exam 70-176
0-7357-0019-2,
$19.99, 12/98

MCSD Fast Track: Solution Architectures
0-7357-0029-X,
$19.99, Q2/99

TESTPREPS

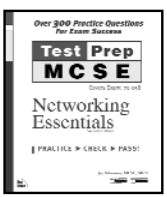
PRACTICE, CHECK, PASS!

Questions. Questions. And more questions. That's what you'll find in our New Riders *TestPreps*. They're great practice books when you reach the final stage of studying for the exam. We recommend them as supplements to our *Training Guides*.

What makes these study tools unique is that the questions are the primary focus of each book. All the text in these books support and explain the answers to the questions.

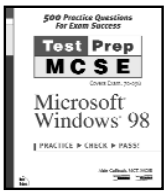
- ✓ **Scenario-based questions** challenge your experience.
- ✓ **Multiple-choice questions** prep you for the exam.
- ✓ **Fact-based questions** test your product knowledge.
- ✓ **Exam strategies** assist you in test preparation.
- ✓ **Complete yet concise explanations of answers** make for better retention.
- ✓ **Two practice exams** prepare you for the real thing.
- ✓ **Fast Facts** offer you everything you need to review in the testing center parking lot.

Practice, practice, practice—pass with New Riders* TestPreps*!



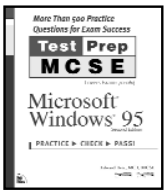
MCSE TestPrep: Networking Essentials, Second Edition

0-7357-0010-9, $19.99, 12/98



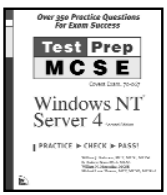
MCSE TestPrep: Windows 98

1-56205-922-X, $19.99, 11/98



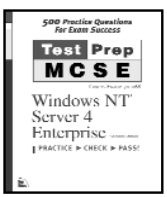
MCSE TestPrep: Windows 95, Second Edition

0-7357-0011-7, $29.99, 12/98



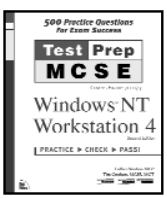
MCSE TestPrep: Windows NT Server 4, Second Edition

0-7357-0012-5, $19.99, 12/98



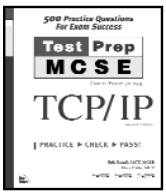
MCSE TestPrep: Windows NT Server 4 Enterprise, Second Edition

0-7357-0009-5, $19.99, 11/98



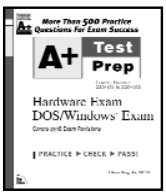
MCSE TestPrep: Windows NT Workstation 4, Second Edition

0-7357-0008-7, $19.99, 11/98



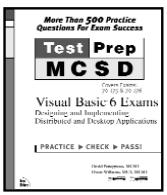
MCSE TestPrep: TCP/IP, Second Edition

0-7357-0025-7, $19.99, 12/98



A+ Certification TestPrep

1-56205-892-4, $19.99, 12/98



MCSD TestPrep: Visual Basic 6 Exams

0-7357-0032X, $29.99, 1/99

TEST PREPS

FIRST EDITIONS

MCSE TestPrep: SQL Server 6.5 Administration, 0-7897-1597-X

MCSE TestPrep: SQL Server 6.5 Design and Implementation, 1-56205-915-7

MCSE TestPrep: Windows 95 70-64 Exam, 0-7897-1609-7

MCSE TestPrep: Internet Explorer 4, 0-7897-1654-2

MCSE TestPrep: Exchange Server 5.5, 0-7897-1611-9

MCSE TestPrep: IIS 4.0, 0-7897-1610-0

How to Contact Us

IF YOU NEED THE LATEST UPDATES ON A TITLE THAT YOU'VE PURCHASED:

1) Visit our Web site at www.newriders.com .

2) Click on the Product Support link, and enter your book's ISBN number, which is located on the back cover in the bottom right-hand corner.

3) There you'll find available updates for your title.

IF YOU ARE HAVING TECHNICAL PROBLEMS WITH THE BOOK OR THE CD THAT IS INCLUDED:

1) Check the book's information page on our Web site according to the instructions listed above, or

2) Email us at support@mcp.com , or

3) Fax us at (317) 817-7488 attn: Tech Support.

IF YOU HAVE COMMENTS ABOUT ANY OF OUR CERTIFICATION PRODUCTS THAT ARE NON-SUPPORT RELATED:

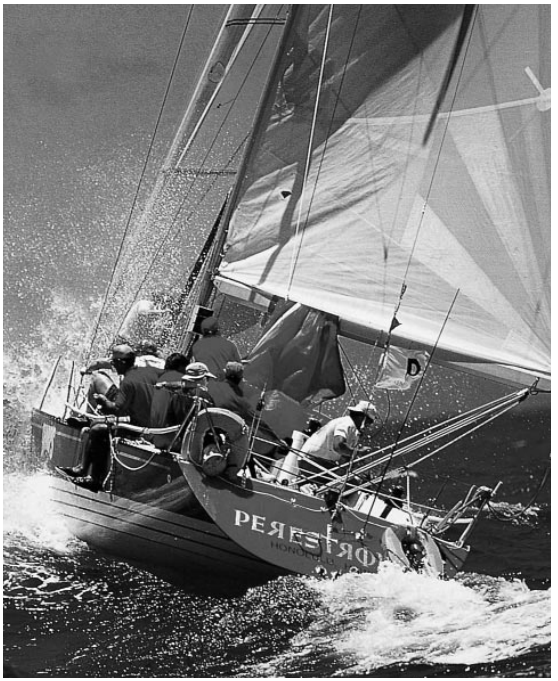
1) Email us at certification@mcp.com , or

2) Write to us at New Riders, 201 W. 103rd St., Indianapolis, IN 46290-1097, or

3) Fax us at (317) 581-4663.

IF YOU ARE OUTSIDE THE UNITED STATES AND NEED TO FIND A DISTRIBUTOR IN YOUR AREA:

Please contact our international department at international@mcp.com .



IF YOU WISH TO PREVIEW ANY OF OUR CERTIFICATION BOOKS FOR CLASSROOM USE:

Email us at pr@mcp.com. Your message should include your name, title, training company or school, department, address, phone number, office days/hours, text in use, and enrollment. Send these details along with your request for desk/examination copies and/or additional information.

WE WANT TO KNOW WHAT YOU THINK

To better serve you, we would like your opinion on the content and quality of this book. Please complete this card and mail it to us or fax it to 317-581-4663.

Name _____

Address _____

City _____ State _____ Zip _____

Phone _____ Email Address _____

Occupation _____

Which certification exams have you already passed? _____

Which certification exams do you plan to take? _____

What influenced your purchase of this book?

- ❑ Recommendation
- ❑ Table of Contents
- ❑ Magazine Review
- ❑ Reputation of New Riders
- ❑ Cover Design
- ❑ Index
- ❑ Advertisement
- ❑ Author Name

How would you rate the contents of this book?

- ❑ Excellent
- ❑ Good
- ❑ Below Average
- ❑ Very Good
- ❑ Fair
- ❑ Poor

What other types of certification products will you buy/have you bought to help you prepare for the exam?

- ❑ Quick reference books
- ❑ Study guides
- ❑ Testing software
- ❑ Other

What do you like most about this book? Check all that apply.

- ❑ Content
- ❑ Accuracy
- ❑ Listings
- ❑ Index
- ❑ Price
- ❑ Writing Style
- ❑ Examples
- ❑ Design
- ❑ Page Count
- ❑ Illustrations

What do you like least about this book? Check all that apply.

- ❑ Content
- ❑ Accuracy
- ❑ Listings
- ❑ Index
- ❑ Price
- ❑ Writing Style
- ❑ Examples
- ❑ Design
- ❑ Page Count
- ❑ Illustrations

What would be a useful follow-up book to this one for you? _____

Where did you purchase this book? _____

Can you name a similar book that you like better than this one, or one that is as good? Why? _____

How many New Riders books do you own? _____

What are your favorite certification or general computer book titles? _____

What other titles would you like to see us develop? _____

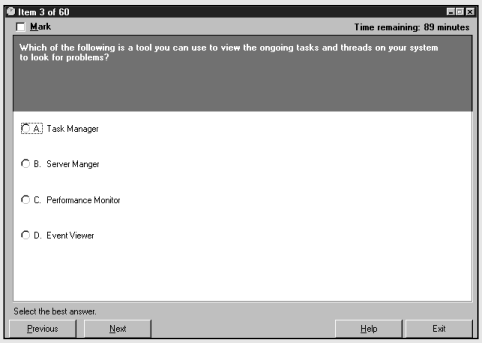
Any comments for us? _____

By opening this package, you are agreeing to be bound by the following agreement:

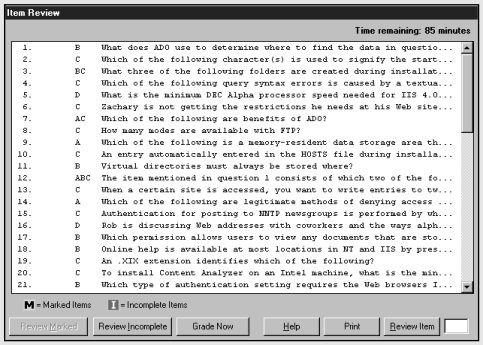
Some of the software included with this product may be copyrighted, in which case all rights are reserved by th
respective copyright holder. You are licensed to use software copyrighted by the publisher and its licensors on a
gle computer. You may copy and/or modify the software as needed to facilitate your use of it on a single comp
Making copies of the software for any other purpose is a violation of the United States copyright laws.

This software is sold as is without warranty of any kind, either expressed or implied, including but not limited
the implied warranties of merchantability and fitness for a particular purpose. Neither the publisher nor its dea
or distributors assumes any liability for any alleged or actual damages arising from the use of this program. (So
states do not allow for the exclusion of implied warranties, so the exclusion may not apply to you.)

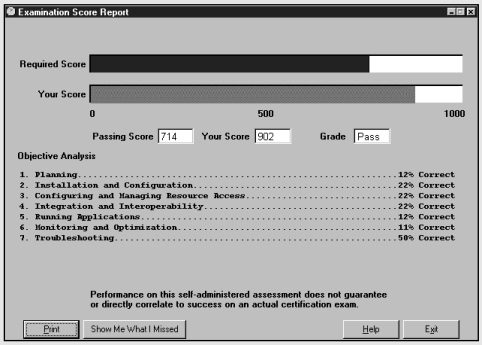
New Riders Top Score Test Simulation Software Suite



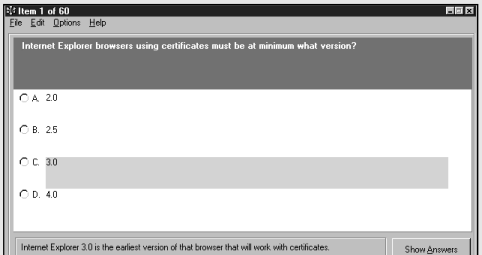
Practice Exams simulate the actual Microsoft exams. Option buttons and check boxes indicate whether there is one or more than one correct answer. All test questions are presented randomly to create a unique exam each time you practice—the ideal way to prepare.



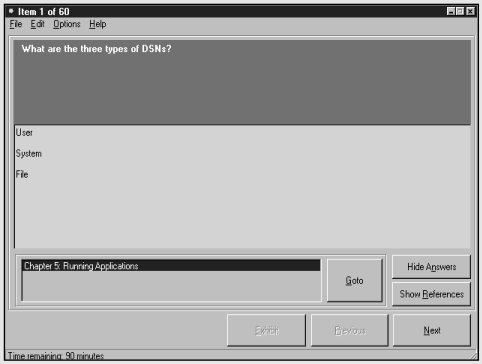
The Item Review shows you the answers you've already selected and the questions you need to revisit before grading the exam.



The Score Report displays your score for each objective category, helping you to define which objectives you need to study more. It also shows you what score you need to pass and your total score.

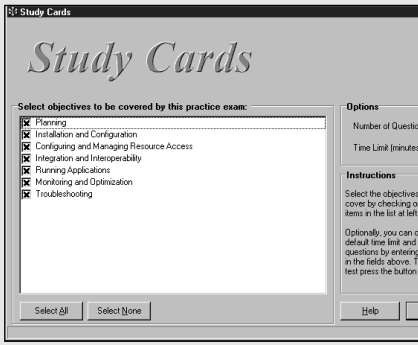


Study Cards allow you to test yourself



Flash Cards don't give you the answers—you must answer each question in your own words.

Customizable—Take an entire exam or select only the objectives and number of questions you want to study.



To use New Riders Top Score software to its fullest potential:

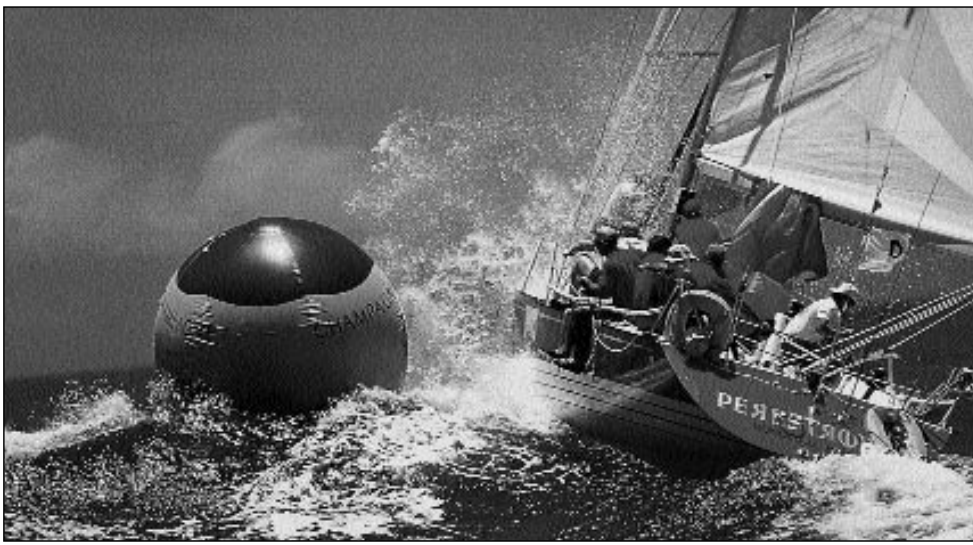
- Take a Practice Exam to become familiar with the exam format and to identify your strong and weak points. Print out the Score Report as a reference for the topics you need to study more.
- Use Study Cards for in-depth practice on specific objectives. Select an answer, check yourself immediately, and find out why the correct answer is right. You can also link back to the appropriate text within the Training Guide for further explanation.
- When you feel like you're ready for the actual exam, try the Flash Cards program first. You can't view the possible answers, so you need to know everything by memory. If you need a refresher, click on the Feedback button to see the correct answer or link directly to the relevant chapter within the Training Guide.
- Finally, before you take the exam, check yourself again with a Top Score Practice Exam. You'll be able to see what areas you may need to review again before the test.

Minimum System Requirements:
486/66 or higher, Pentium recommended
Windows 95 or Windows NT 4 (or later)
16MB RAM

Now Include

Practice exam-style interactive testing with the new Windows NT Workstation 4 Simulator! Step through basic tasks in Windows NT with the new simulator created for New Riders. Practice with beginner-, intermediate- and advanced-level tasks. Test yourself without jeopardizing a live system.

TRAINING GUIDE



MCSD

VISUAL BASIC 6 Exams

Exams: 70-175 and 70-176

Howard Hawhee, Senior Author
Corby Jordan
Richard Hundhausen
Felipe Martins
Thomas Moore

New Riders

MCSD Training Guide: Visual Basic 6 Exams

Copyright® 1999 by New Riders Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-7357-0002-8

Library of Congress Catalog Card Number: 98-83085

Printed in the United States of America

First Printing: March, 1999

03 02 01 00 99 7 6 5 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. New Riders cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Microsoft is a registered trademark of Microsoft Corporation in the United States and other countries. New Riders is an independent entity from Microsoft Corporation, and not affiliated with Microsoft Corporation in any manner. This publication may be used in assisting students to prepare for a Microsoft Certified Professional Exam. Neither Microsoft Corporation, its designated review company, nor New Riders warrants that use of this publication will ensure passing the relevant Exam.

EXECUTIVE EDITOR
Mary Foote

ACQUISITIONS EDITOR
Nancy Maragioglio

DEVELOPMENT EDITORS
Chris Zahn
Chris Morris

MANAGING EDITOR
Sarah Kearns

PROJECT EDITOR
Clint McCarty

COPY EDITORS
Keith Cline
Linda Neher

INDEXER
Chris Wilcox

TECHNICAL EDITORS
Mandeep Binning
Duncan Mackenzie
David Shapton

SOFTWARE DEVELOPMENT SPECIALIST
Jack Belbot

PROOFREADERS
John Rahm
Sheri Replin

PRODUCTION
Cheryl Lynch
Jeanette McKay

Contents at a Glance

Introduction *01*

Part I Visual Basic 6 Exam Concepts

1 Developing the Conceptual and Logical Design and Deriving the Physical Design 15
2 Establishing the Development Environment 43
3 Implementing Navigational Design 77
4 Creating Data Input Forms and Dialog Boxes 125
5 Writing Code that Validates User Input 209
6 Writing Code that Processes Data Entered on a Form 237
7 Implementing Online User Assistance in a Distributed Application 265
8 Creating Data Services: Part I 303
9 Creating Data Services: Part II 375
10 Instantiating and Invoking a COM Component 445
11 Implementing Error-Handling Features in an Application 477
12 Creating a COM Component that Implements Business Rules or Logic 509
13 Creating ActiveX Controls 609
14 Creating an Active Document 683
15 Understanding the MTS Development Environment 735
16 Developing MTS Applications 765
17 Internet Programming with IIS/Webclass and DHTML Applications 797
18 Using VB's Debug/Watch Facilities 843
19 Implementing Project Groups to Support the Development and Debugging Process 897
20 Compiling a VB Application 917
21 Using the Package and Deployment Wizard to Create a Setup Program 963

PART II Final Review

Fast Facts 1,007
Study and Exam Prep Tips 1,039
Practice Exams 1,049

Part III Appendixes

A Glossary 1,087
B Overview of the Certification Process 1,099
C What's On the CD-ROM 1,107
D Using the Top Score Software 1,109
E Visual Basic Basics 1,117
F Suggested Readings and Resources 1,167
Index *1,169*

Table of Contents

PART I: Visual Basic 6 Exam Concepts

1 Developing the Conceptual and Logical Design and Deriving the Physical Design 15

Introduction 18

Overview of Microsoft Application Development Concepts 18

The VB Enterprise Development Model 20
- The Conceptual Design 20
- Deriving the Logical Design From the Conceptual Design 21
- Deriving the Physical Design From the Logical Design 22
- Assessing the Logical Design's Impact on the Physical Design 23

Designing VB Data-Access Components for a Multitier Application 29

Designing Properties, Methods, and Events of Components 30
- Designing Properties of Components 30
- Designing Methods of Components 31
- Designing Events of Components 31

Implementing Load Balancing 32
- Review Questions 37
- Exam Questions 37
- Answers to Review Questions 39
- Answers to Exam Questions 40

2 Establishing the Development Environment 43

Introduction 45

Implementing Source-Code Control with Visual SourceSafe 45
- The Nature of a Visual SourceSafe Project 46
- The Visual SourceSafe Database 46
- Visual SourceSafe Administrator 47

Visual SourceSafe Explorer 51
Installing and Configuring VB for Developing Desktop and Distributed Applications 65
Exercises 68
Review Questions 72
Exam Questions 72
Answers to Review Questions 74
Answers to Exam Questions 74

3 Implementing Navigational Design 77

Introduction 80
Understanding Menu Basics 81
Knowing Menu Terminology 81
Using the Menu Editor 82
Attaching Code to a Menu Item's Click Event Procedure 83
Dynamically Modifying the Appearance of a Menu 84
Adding a Pop-Up Menu to an Application 85
Defining the Pop-Up Menu 85
Determining the Mouse Button 86
Displaying the Pop-Up Menu 88
Controls With Pop-Up Menus 88
Creating an Application That Adds and Deletes Menus at Runtime 89
Creating Runtime Menu Items 89
Code for Runtime Menu Items 90
Removing Runtime Menu Items 91
Adding Controls to Forms 91
Setting Properties for CommandButtons, TextBoxes, and Labels 92
Referring to a Property Within Code 94
Important Common Properties of CommandButtons, TextBoxes, and Labels 95
Important Properties of the CommandButton Control 99
Important Properties of the TextBox Control 100
Important Properties of the Label Control 102

Assigning Code to a Control to Respond to an Event 103
Changing a Control Name After You Assign Code to the Event Procedure 104
The Click Event 105
The DblClick Event 106
MouseUp and MouseDown 106
Mouse Events Compared With Click and DblClick 108
MouseMove 109
The Change Event 109
Other Events Commonly Used for Input Validation 110
Exercises 111
Review Questions 119
Exam Questions 119
Answers to Review Questions 121
Answers to Exam Questions 123

4 Creating Data Input Forms and Dialog Boxes 125

Introduction 127
Adding an ActiveX Control to the ToolBox 128
Using ActiveX Controls to Create Data Input Forms and Dialog Boxes 129
Using the ImageList Control 129
Using the TreeView Control 134
Using the ListView Control 139
Using the ToolBar Control 147
Using the StatusBar Control 153
Using the Controls Collection 157
Techniques for Adding and Deleting Controls Dynamically 160
More on Creating Data Input Forms and Dialog Boxes 172
Using the Forms Collection 179
Exercises 186
Review Questions 197
Exam Questions 198
Answers to Review Questions 203
Answers to Exam Questions 204

5 Writing Code that Validates User Input 209

Keystroke Events at Field and Form Level 212
The KeyPress Event 212
The KeyUp and KeyDown Events 214
KeyPress Versus KeyUp and KeyDown 216
Enabling Two-Tier Validation With the Form's KeyPreview Property 216
Field-Level Validation Techniques 217
The Validate Event and CausesValidation Property 218
The Validate Event 218
The CausesValidation Property 219
The Change Event and Click Events 220
An Obsolete Technique: Validation With GotFocus and LostFocus Events 221
Enabling Controls Based on Input 222
Miscellaneous Properties for Validation 224
MaxLength 224
Data-Bound Properties 224
Exercises 228
Review Questions 233
Exam Questions 233
Answers to Review Questions 235
Answers to Exam Questions 235

6 Writing Code that Processes Data Entered on a Form 237

Introduction 239
Relative Timing of Form Events 239
Initialize, Load, and Activate Events 240
The Initialize Event 240
The Load Event and the Activate Event 241
DeActivate, Unload, QueryUnload, and Terminate Events 243
The DeActivate Event 243
The QueryUnload Event 243
The Unload Event 245
The Terminate Event 246

Activate /DeActivate Versus GotFocus /LostFocus Events 246
Show/Hide Methods Versus Load/Unload Statements 247
Using the Unload and QueryUnload Events in an MDI Application 248
Form Methods and Their Effect on Form Events 249
Implicitly Loading a Form 249
Show and Hide 249
Manipulating a Form From Another Form's Load Event Procedure 250
Exercises 255
Review Questions 259
Exam Questions 259
Answers to Review Questions 261
Answers to Exam Questions 262

7 Implementing Online User Assistance in a Distributed Application 265

Two Types of Help Files 267
HTML Help Files 267
WinHelp Files 267
Referencing Help Through the HelpFile Property of an Application 268
Setting Help Files at Design Time 269
Setting Help Files at Runtime 270
Context-Sensitive Help for Forms and Controls 271
Context-Sensitive Help With the HelpContextID Property 271
Adding ToolTips to an Application 272
Providing WhatsThisHelp in an Application 273
Creating HTML Help 276
HTML Help Source File Structures 276
Creating and Compiling an HTML Help File Project With HTML Help Workshop 277
Exercises 294
Review Questions 299
Exam Questions 299
Answers to Review Questions 301
Answers to Exam Questions 301

8 Creating Data Services: Part I 303

Introduction 305

Overview of OLE DB and ADO 305
ADO and the ADO Object Model 306

Programming With Automated Data-Binding Tools 308
Managing ADO Objects With the Data Environment Designer 308

Accessing Data With ADO and the ADO Data Control 317
Using the ADO Data Control 318
Programming With ADO 329

Using the ADO Errors Collection 357
Exercises 359
Review Questions 368
Exam Questions 368
Answers to Review Questions 371
Answers to Exam Questions 371

9 Creating Data Services: Part II 375

Introduction 378

ADO Data-Access Models 378
Accessing Data With the Execute Direct Model 379
Accessing Data With the Prepare/Execute Model 380
Accessing Data With the Stored Procedures Model 382
How to Choose a Data-Access Model 383

Using Stored Procedures 384
Creating Stored Procedures 385
Using the Parameters Collection to Manipulate and Evaluate Parameters for Stored Procedures 388
Using Stored Procedures to Execute Statements on a Database 390
Using Stored Procedures to Return Records to an Application 396

Using Cursors 400
Using Cursor Locations 400
Using Cursor Types 402

Managing Database Transactions 404

Writing SQL Statements 408
Writing SQL Statements that Retrieve and Modify Data 409
Writing SQL Statements that Use Joins to Combine Data from Multiple Tables 411
Using Locking Strategies to Ensure Data Integrity 413
Choosing Cursor Options 414
Exercises 417
Review Questions 436
Exam Questions 436
Answers to Review Questions 440
Answers to Exam Questions 441

10 Instantiating and Invoking a COM Component 445

Introduction 448
COM, Automation, and ActiveX 449
Creating a Visual Basic Client Application that Uses a COM Component 451
Setting a Reference to a COM Component 452
Using the Object Browser to Find Out About a COM Component's Object Model 453
Using the New Keyword to Declare and Instantiate a Class Object from a COM Component 455
Late and Early Binding of Object Variables 456
Using the CreateObject and GetObject Functions to Instantiate Objects 458
Using a Component Server's Object Model 460
Manipulating the Component's Methods and Properties 461
Releasing an Instance of an Object 463
Detecting Whether a Variable Is Instantiated 463
Handling Events from a COM Component 463
Exercises 468
Review Questions 472
Exam Questions 472
Answers to Review Questions 474
Answers to Exam Questions 474

11 Implementing Error-Handling Features in an Application 477

Introduction 479

Setting Error-Handling Options 479
- Setting Break on All Errors 480
- Setting Break in Class Modules 480
- Setting Break on Unhandled Errors 481

Using the Err Object 481
- Properties of the Err Object 482
- Methods of the Err Object 485
- Using the vbObjectError Constant 486

Handling Errors in Code 487
- Using the On Error Statement 487
- Inline Error Handling 489
- Error-Handling Routines 490
- Trappable Errors 492

Using the Error-Handling Hierarchy 494

Common Error-Handling Routines 496

Using the Error Function 499

Using the Error Statement 499

Inline Error Handling 500
- Exercises 503
- Review Questions 505
- Exam Questions 505
- Answers to Review Questions 507
- Answers to Exam Questions 507

12 Creating a COM Component that Implements Business Rules or Logic 509

Introduction 513

Overview of COM Component Programming 513
- The COM Specification and the ActiveX Standard 514
- Comparing In-Process and Out-of-Process Server Components 515

Steps in Creating a COM Component 517

Implementing Business Rules With COM Components 518
Implementing an Object Model With a COM Component 519
Implementing COM Components Through Class Modules 520
The Uses of Class Modules 520
Starting a Class Module in a Standard EXE Project 521
The Class Module Name Property 521
Implementing Custom Methods in Class Modules 523
Implementing Custom Properties in Class Modules 524
Implementing Custom Events in Class Modules 528
Built-In Events of Class Modules 530
Using Public , Private , and Friend 532
Storing Multiple Instances of an Object in a Collection 533
Declaring and Using a Class Module Object in Your Application 538
Managing Threads in a COM Component 541
Managing Threads in ActiveX Controls and In-Process Components 542
Managing Threading in Out-of-Process Components 542
The Instancing Property of COM Component Classes 544
Using Private Instancing for Service Classes 545
Using PublicNotCreatable Instancing for Dependent Classes 545
Instancing Property Settings for Externally Creatable Classes 546
Deciding Between SingleUse and MultiUse Server Classes 549
Handling Errors in the Server and the Client 549
Passing a Result Code to the Client 550
Raising an Error to Pass Back to the Client 551
Managing Components With Visual Component Manager 552
Storing VCM Information in Repository Databases 553
Making VCM Available in the VB IDE 553
Publishing Components With VCM 555
Finding and Reusing Components With VCM 559
Using Interfaces to Implement Polymorphism 561
Steps to Implement an Interface Class 563
Providing Asynchronous Callbacks 572
Providing an Interface for the Callback Object 574
Implementing the Callback Object in the Client 575
Manipulating the Callback Object in the Server 577

Registering and Unregistering a COM Component 579
Registering/Unregistering an Out-of-Process Component 579
Registering/Unregistering an In-Process Component 580
Sending Messages to the User from a COM Component 581
Managing Forms in an Out-Of-Process Server Component 581
Managing Forms in an In-Process Server Component 582
Choosing the Right COM Component Type 583
Implementing Scalability Through Instancing and Threading Models 584
Under-the-Hood Information About COM Components 585
Exercises 588
Review Questions 595
Exam Questions 596
Answers to Review Questions 603
Answers to Exam Questions 604

13 Creating ActiveX Controls 609

Introduction 612
Overview of ActiveX Control Concepts 612
ActiveX Controls as ActiveX Components 612
Creating ActiveX Controls from Constituent Controls 613
Creating User-Drawn ActiveX Controls 614
The Lifetime of an ActiveX Control 614
Control Authors and Developers 615
Special Considerations for ActiveX Control Development 615
Steps to Creating an ActiveX Control that Expose Properties 616
The UserControl Object 616
Implementing User-Drawn Graphic Features 623
Implementing Custom Methods 624
Implementing Custom Events 625
Implementing Custom Properties 627
Implementing Property Persistence 631
Implementing Constituent Controls 639
Creating Data-Aware ActiveX Controls 645
Enabling the Data-Binding Capabilities of an ActiveX Control 645
Creating an ActiveX Control that Is a Data Source 647

Create and Enable Property Pages for ActiveX Controls 652
Creating the PropertyPage Object's Visual Interface 653
Determining Which Controls Are Selected for Editing With the SelectedControls Collection 654
Using the SelectionChanged Event to Detect When the Developer Begins to Edit Properties 655
Flagging Property Changes With the Changed Property 656
Saving Property Changes With the ApplyChanges Event 657
Connecting a Custom Control to a Property Page 658
Connecting a Single Complex Property to a Property Page 658
Detecting Which Complex Property Is Being Edited With the EditProperty Event 660
Connecting a Property to a Standard VB Property Page 661
Testing and Debugging Your ActiveX Control 661
Testing Your ActiveX Control With Existing Container Applications 662
Testing and Debugging Your ActiveX Control in a Test Project 663
What to Look for When Testing Your ActiveX Control 666
Exercises 670
Review Questions 676
Exam Questions 676
Answers to Review Questions 679
Answers to Exam Questions 680

14 Creating an Active Document 683

Introduction 686
Overview and Definition of Active Documents 687
Steps to Implementing an Active Document 688
Setting Up the UserDocument 689
Converting an Existing Project to an Active Document 690
Creating an Active Document Project 690
Choosing Between an Active Document EXE and an Active Document DLL 691
Running Your Active Document in a Container Application 692
Detecting the Type of Container With the TypeName Function and UserDocument.Parent 693

Managing the Events in Your Active Document's Lifetime 694
Initialize Event 694
InitProperties Event 695
EnterFocus Event 695
Show Event 695
The ReadProperties Event and ReadProperty Method 696
The WriteProperties Event and the WriteProperty Method 697
ExitFocus Event 698
Hide Event 698
Terminate Event 698
Managing Active Document Scrolling 698
The Scrollbars Property and MinHeight and MinWidth Properties 699
The HScrollSmallChange and VScrollSmallChange Properties 700
The Scroll Event Procedure and the ContinuousScroll Property 700
Managing The Active Document's ViewPort 701
The ViewPort Coordinate Properties 701
SetViewPort Method 704
Defining Your Active Document's Custom Members 704
Methods 705
Properties 705
Data and Property Persistence in Active Documents 706
Saving Information in the .vbd File 706
Data Preservation Events and the Properties Bag 707
Asynchronous Download of Information 708
Starting the Download With the AsyncRead Method 709
Stopping the Download With the CancelAsyncRead Method 710
Reacting to the Download Completion With the AsyncReadComplete Event 711
Defining Your Active Document's Menus 712
Design Considerations for Active Document Menus 712
Negotiating With the Container's Menus 713
Merging Your Help Menu With the Container's Help Menu 714
Limitations of Modeless Forms in an Active Document Project 715

Navigating Between Documents in the Container Application 716
Using the Hyperlink Object With Internet-Aware Containers 716
Navigating the Container App's Object Model 718
Writing an Application to Handle Different Containers' Navigation Styles 718
Creating an ActiveX Project With Multiple UserDocument Objects 719
Testing Your Active Document in the VB Design Environment 722
Compiling and Distributing Your Active Document 724
Using Your Active Document on a Web Page 725
Exercises 727
Review Questions 730
Exam Questions 730
Answers to Review Questions 732
Answers to Exam Questions 733

15 Understanding the MTS Development Environment 735

Introduction 738
Basic MTS Concepts 738
Overview of MTS 738
MTS Packages and Their Relationship to COM Components 739
Setting Up MTS 741
Configuring a Server to Run MTS 741
Installing MTS 741
Setting Up Security on the System Package 744
Working With MTS Packages 746
The Package and Deployment Wizard 746
Creating a Package by Using the MTS Explorer 750
Assigning Names to Packages 752
Assign Security to Packages 753
Exporting and Importing Existing Packages 755
Exercises 759
Review Questions 761
Exam Questions 761
Answers to Review Questions 763
Answers to Exam Questions 763

16 Developing MTS Applications 765

Introduction 768

Calling MTS Components from Visual Basic Clients 768
- Creating Packages That Install or Update MTS Components on a Client 769
- Configuring a Client Computer to Use an MTS Component 771

Developing MTS Components With Visual Basic 772
- Understanding the MTS Runtime Environment 773
- Adding Components to an MTS Package 775
- Using Transactions 777

Understanding MTS Client Development 780

Understanding MTS Security 781
- Using Role-Based Security to Limit Use of an MTS Package to Specific Users 781
- Creating and Adding Users to Roles 782
- Assigning Roles to Components or Component Interfaces 784
- Setting Security Properties of Components 785
- Exercises 788
- Review Questions 792
- Exam Questions 793
- Answers to Review Questions 795
- Answers to Exam Questions 795

17 Internet Programming With IIS/WebClass and DHTML Applications 797

Introduction 799

WebClass Applications 799
- Creating a Simple ASP Page 800
- IIS (WebClass Designer) Applications in VB 803

DHTML Applications 818
- Creating a Web Page With the DHTML Page Designer 818
- Modifying a DHTML Web Page and Positioning Elements 819
- Exercises 827
- Review Questions 838
- Exam Questions 838

Answers to Review Questions 840
Answers to Exam Questions 840

18 Using VB's Debug/Watch Facilities 843

Introduction 846
Preventing Bugs 846
Using Watch Expressions and Contexts 848
Creating a Watch Expression 849
Types of Watch Expression 850
Watch Contexts 852
Using Break Mode 854
Entering Break Mode Manually 854
Stepping Through Your Code 855
Using the Watch Window 859
Entering Break Mode Dynamically 861
Using Quick Watch 863
Watching on Demand 864
Immediate Window and the Debug Object 864
Displaying the Debug Window 864
Displaying Messages Programmatically With the Debug Object 865
Using the Print Method 866
Formatting Debug.Print Messages 867
Displaying Data Values 869
Using the Debug.Assert Method 871
Interacting with the Immediate Window 873
Querying or Modifying Data Values 874
Testing and Executing VB Procedures 875
Using the Locals Window 877
Using the Immediate Window in Place of Breakpoints 880
Using the MouseDown and KeyDown Events 880
Using the GotFocus and LostFocus Events 880
Levels of Scope 881
Local Scope 882
Module Scope 883
Global Scope 884

Scope Considerations 885
Striving to Narrow the Scope 885
Performance Concerns 886
Exercises 888
Review Questions 891
Exam Questions 891
Answers to Review Questions 894
Answers to Exam Questions 894

19 Implementing Project Groups to Support the Development and Debugging Process 897

Introduction 899
Understanding Project Groups 899
Creating Project Groups 900
Building Multiple Projects 902
Using Project Groups to Debug an ActiveX DLL 903
Setting Up a Sample Group 903
Debugging Features in Project Groups 906
Using Project Groups to Debug an ActiveX Control 907
Exercises 910
Review Questions 914
Exam Questions 914
Answers to Review Questions 915
Answers to Exam Questions 916

20 Compiling a VB Application 917

Introduction 919
P-Code Versus Native Code 920
Native Code 920
P-Code 924
Understanding When and How to Optimize 925
Compiling to P-Code 926
Compiling to Native Code 928
Using Compile On Demand 941

Understanding Conditional Compilation 942
Preprocessor Directives 943
Types of Expressions 945
Compiler Constants 947
Applications and Styles 952
Exercises 957
Review Questions 959
Exam Questions 959
Answers to Review Questions 961
Answers to Exam Questions 961

21 Using the Package and Deployment Wizard to Create a Setup Program 963

Introduction 966

Using Package and Deployment Wizard to Create a Setup Program 966
Preparing to Run Package and Deployment Wizard 967
Starting Package and Deployment Wizard and Choosing the Type of Package 968
Choosing the Type of Setup Package 969
Creating a Standard Setup Package 970
Creating an Internet Setup Package 975
Creating a Dependency File 977

Standard Files Used in a Microsoft Setup 978
Setup File Information in SETUP.LST 979
Dependency Information in DEP Files 981
SETUP.EXE and Package and Deployment Wizard's Custom Setup 983

Customizing a Standard Setup 984
Customizing SETUP.LST and Your Application's DEP File 984
Customizing the Standard VB Setup Project 985
Implementing Application Removal 985

Registering a Component that Implements DCOM and Configuring DCOM 986

Deploying Your Application 987
Deploying to Floppy Disks 988
Deploying to a Network Directory or to CDs 990

Deploying to the Web 992
Deploying Updates to Your Application 994
Exercises 996
Review Questions 1,000
Exam Questions 1,000
Answers to Review Questions 1,002
Answers to Exam Questions 1,002

PART II: Final Review

Fast Facts **1,007**

Developing the Conceptual and Logical Design 1,007
Deriving the Physical Design 1,008
Establishing the Development Environment 1,009
Creating User Services 1,011
Creating and Managing COM Components 1,022
Creating Data Services 1,029
Testing the Solution 1,032
Deploying an Application 1,035
Maintaining and Supporting an Application 1,036

Study and Exam Prep Tips **1,039**

Study Tips 1,040
Study Strategies 1,040
Pre-Testing Yourself 1,041
Exam Prep Tips 1,041
The MCP Exam 1,041
Exam Format 1,042
New Question Types 1,044
Putting It All Together 1,045
Final Considerations 1,047

Practice Exams **1,049**

Exam 1: Developing Distributed Applications (70-175) 1,050
Answers to Exam Questions 1,062
Exam 2: Developing Desktop Applications (70-176) 1,068
Answers to Exam Questions 1,080

Part III Appendixes

A Glossary **1,087**

B Overview of the Certification Process **1,099**

Types of Certification 1,099
Certification Requirements 1,100
How to Become a Microsoft Certified Professional 1,100
How to Become a Microsoft Certified Professional+Internet 1,100
How to Become a Microsoft Certified Professional+Site Building 1,101
How to Become a Microsoft Certified Systems Engineer 1,101
How to Become a Microsoft Certified Systems Engineer+Internet 1,103
How to Become a Microsoft Certified Solution Developer 1,103
Becoming a Microsoft Certified Trainer 1,105

C What's On the CD-ROM **1,107**

Top Score 1,107
Exclusive Electronic Version of Text 1,107
Copyright Information and Disclaimer 1,107

D Using the Top Score Software **1,109**

Getting Started 1,109
Instructions on Using the Top Score Software 1,109
Using Top Score Practice Exams 1,110
Using Top Score Study Cards 1,112

Using Top Score Flash Cards 1,113
Using Top Score Simulator 1,115
Summary 1,116

E Visual Basic Basics 1,117

Programming With Variables in VB 1,117
Declaring and Defining the Scope of a Variable 1,117
Using the Appropriate Declaration Statement 1,121
Understanding Visual Basic's Standard Simple Data Types 1,125
Checking the Data Type of a Variable 1,131
Converting Between Data Types 1,132
Common String-Manipulation Functions 1,136
Using Arrays 1,139
Working with Collections 1,142
Programming With Sub and Function Procedures 1,143
Sub Procedures 1,143
Functions 1,145
Passing Arguments to General Procedures 1,146
Using Exit Sub or Exit Function to Abruptly Terminate a Procedure 1,153
Syntax for Calling Procedures 1,154
Procedure Scope 1,156
Control Names and Event Procedure Names 1,157
Programming With VB's Control Structures 1,158
Branching or Selection 1,158
Looping 1,161
Terminating a Loop Abruptly With Exit 1,164
The With...End With Construct 1,164
Obsolete Techniques 1,165

F Suggested Readings and Resources 1,167

Index 1,169

About the Authors

Howard Hawhee is a Microsoft Certified Professional (MCP). He has broad experience in the world of PC development and training. He has served numerous corporate clients as a developer, consultant, and instructor over the past fourteen years and has never stopped having fun using and explaining Visual Basic.

He currently works for Solomon Technology Center (formerly ClearView Software, now a business unit of Solomon Software, Inc.), an organization devoted to customizing and supporting the Solomon IV accounting software package with Visual Basic.

He can be reached through New Riders Publishing.

Corby Jordan has been in the technology industry for 7 years, working with such companies as Intel, Autodesk and Boeing. As a Microsoft Certified Trainer (MCT), he has trained hundreds of professionals seeking to become MCSEs and MCSDs. He is currently working as the IT Lead Project Manager at IRSC, with a focus on developing multi-tier applications. He lives in southern California with his wife Michele, and his three kids, Caitlin, Sophia and Symeon.

Richard Hundhausen, Microsoft Certified Trainer and Microsoft Certified Solution Developer (MCT, MCSD), is an independent trainer and consultant specializing in Microsoft Visual Studio and SQL Server development. His primary focus is delivering Microsoft certified courses aimed at preparing students to become Microsoft Certified Solution Developers. He has worked with computer systems for 15 years and currently lives in Stuttgart, Germany with his devoted wife Kristen and their two sled dogs, Shanee and Dawson. Send him email at atomicity@msn.com.

Felipe Martins is a Microsoft Certified Trainer and Microsoft Certified Solution Developer (MCT and MCSD) who has been working with VB since version 3.0 building software in the manufacturing, healthcare, and aerospace industries. He currently works for ImagiNET Resources Corp. as a Senior Solutions Developer. ImagiNET Resources Corp supplies clients with leading edge business solutions by employing highly skilled and experienced people who are experts at using Microsoft technologies. Besides training and software development, Felipe enjoys his family life, a good game of Quake, and a strong cup of coffee. Felipe Martins can be reached at fmartins@imaginets.com.

Thomas Moore is a Microsoft Certified Trainer, Microsoft Certified Solution Developer, and a Microsoft Certified Systems Engineer (MCT, MCSD, MCSE). He has been a computer programmer since 1981. He is knowledgeable in many programming languages including COBOL, Pascal, Visual Basic, C, Fortran, and RPG. He has development experience on many platforms from the PC to supercomputers. Thomas also has extensive database development experience including CICS, dBase, and Microsoft SQL Server. He has performed training for government agencies in Canada and US and has extensive experience as a LAN manager (7 years).

Thomas has conducted technical training for 15 years at both private and public colleges as well as in the profession context (ATEC). His computer curricula background includes lesson, exam, and case study development as well as creation of other classroom materials.

About the Technical Editors

Duncan Mackenzie is a Microsoft Certified Trainer and Microsoft Certified Solution Developer (MCT, MCSD). He has worked on a wide variety of projects, including ASP-based Web sites, E-Commerce implementations and many different types of Visual Basic applications. He is currently working as an independent consultant and can be reached through New Riders Publishing or his own Web site at http://www.dmconsulting.mb.ca .

Mandeep Binning is a Microsoft Certified Solution Developer (MCSD), who holds Associate of Science and Associate of Arts degrees. He began Basic programming and working with computers in 1982 and has been a professional programmer and solutions consultant since 1993. He enjoys his work doing custom programming for clients—particularly using Visual Basic, Microsoft Office, and SQL—and running his company, Flying Mouse Programming.

Some of his most recent projects have included:

- Creating the SPSS Report Parser and Unbundler for automating a data retrieval system
- Designing and developing reporting systems for SQL databases
- Connecting mainframe data sources to PC desktops
- Full sales and order processing system implemented with Access

In addition, he has just begun publishing the Flying Mouse Newsletter. Located in Vancouver, BC, Flying Mouse has clients throughout North America. You may contact Mandeep at mbinning@smartt.com .

David Shapton is a Microsoft Certified Trainer and Microsoft Certified Solution Developer (MCT, MCSD). He has over 14 years experience in systems development and training and is currently associated with LearnQuest/ExecuTrain in Ottawa as a technical trainer and consultant. David holds a bachelor's degree from McMaster University. Most of his professional life is now spent in the classroom teaching Microsoft developer courses, but he still enjoys getting involved in development and writing projects. He lives outside Ottawa with his wife, Shauna and their children, Krista, Olivia and Mitchell.

Dedication

To all those who've given their support during the writing of this book.

Acknowledgments

Thanks are due to the following people and organizations for their support during this project:

To Nancy Maragioglio and Chris Zahn at New Riders Publishing, who made this book possible.

To the very understanding management of Solomon Technology Center (Bobby Priestley, Dave Stritzinger, Jim Stritzinger), who kindly gave me enough slack to get this book written.

To the employees of ClearView Software, Dallas, for their understanding and suggestions as VB experts and veteran Certified Exam takers.

And finally, to the patient and knowledgeable technical editors on this book, Duncan Mackenzie and Mandeep Binning, two VB experts with a great deal of patience and ability to focus on detail.

Tell Us What You Think!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As the Executive Editor for the Certification team at New Riders Publishing, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

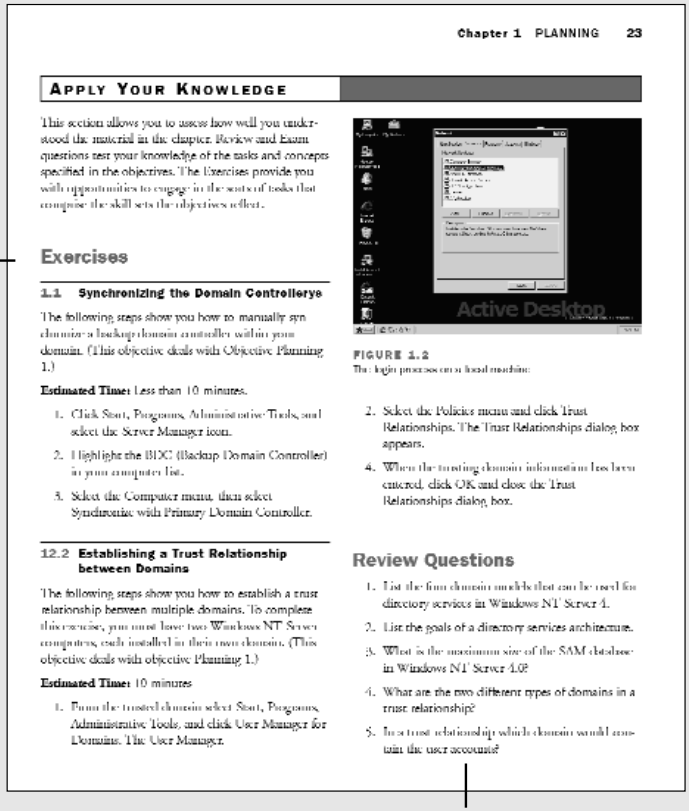
Fax: 317-581-4663

Email: certification@mcp.com

Mail: Mary Foote
Executive Editor
Certification
New Riders Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Apply Your Knowledge

Among the many new features New Riders has incorporated into its second edition Training Guides, you will find extensive review and self-evaluation options. The end of each chapter includes a section titled "Apply Your Knowledge," which contains exercises, open-ended review questions, and multiple-choice questions that mimic the style of questions you will see on the exam. See "How to Use This Book" for a visual demonstration and explanation of the other features designed to benefit the learning process and maximize your study time.



Chapter 1 PLANNING 23

APPLY YOUR KNOWLEDGE

This section allows you to assess how well you understood the material in the chapter. Review and Exam questions test your knowledge of the tasks and concepts specified in the objectives. The Exercises provide you with opportunities to engage in the sorts of tasks that comprise the skill sets the objectives reflect.

Exercises

1.1 Synchronizing the Domain Controllers

The following steps show you how to manually synchronize a backup domain controller within your domain. (This objective deals with Objective Planning 1.)

Estimated Time: Less than 10 minutes.

1. Click Start, Programs, Administrative Tools, and select the Server Manager icon.
2. Highlight the BDC (Backup Domain Controller) in your computer list.
3. Select the Computer menu, then select Synchronize with Primary Domain Controller.

12.2 Establishing a Trust Relationship between Domains

The following steps show you how to establish a trust relationship between multiple domains. To complete this exercise, you must have two Windows NT Server computers, each installed in their own domain. (This objective deals with objective Planning 1.)

Estimated Time: 10 minutes

1. From the trusted domain select Start, Programs, Administrative Tools, and click User Manager for Domains. The User Manager.

FIGURE 1.2
The login process on a local machine.

2. Select the Policies menu and click Trust Relationships. The Trust Relationships dialog box appears.
4. When the trusting domain information has been entered, click OK and close the Trust Relationships dialog box.

Review Questions

1. List the four domain models that can be used for directory services in Windows NT Server 4.
2. List the goals of a directory services architecture.
3. What is the maximum size of the SAM database in Windows NT Server 4.0?
4. What are the two different types of domains in a trust relationship?
5. In a trust relationship which domain would contain the user accounts?

Exercises: These activities provide an opportunity for you to master specific hands-on tasks. Our goal is to increase your proficiency with the product or technology. You must be able to conduct these tasks in order to pass the exam.

Review Questions: These open-ended short-answer questions allow you to quickly assess your comprehension of what you just read in the chapter. Instead of allowing you to choose from a list of options, these questions require you to state the correct answers in your own words. Although you will not experience these kinds of questions on the exam, these questions will indeed test the level of your comprehension of key concepts.

6. Can a local account be used in a trust relationship? Explain.
7. In a complete trust domain model that uses 4 different domains, what is the total number of trust relationships required to use a complete trust domain model?

Exam Questions

The following questions are similar to those you will face on the Microsoft exam. Answers to these questions can be found in section Answers and Explanations, later in the chapter. At the end of each of those answers, you will be informed of where (that is, in what section of the chapter) to find more information..

1. ABC Corporation has locations in Toronto, New York, and San Francisco. It wants to install Windows NT Server 4 to encompass all its locations in a single WAN environment. The head office is located in New York. What is the best domain model for ABCis directory services implementation?
   A. Single-domain model
   B. Single-master domain model
   C. Multiple-master domain model
   D. Complete-trust domain model
2. JPS Printing has a single location with 1,000 users spread across the LAN. It has special printers and applications installed on the servers in its environment. It needs to be able to centrally manage the user accounts and the resources. Which domain model would best fit its needs?
   A. Single-domain model
   B. Single-master domain model
   C. Multiple-master domain model
   D. Complete-trust domain model
5. What must be created to allow a user account from one domain to access resources in a different domain?
   A. Complete Trust Domain Model
   B. One Way Trust Relationship
   C. Two Way Trust Relationship
   D. Master-Domain Model

Answers to Review Questions

1. Single domain, master domain, multiple-master domain, complete-trust domain. See section, Windows NT Server 4 Domain Models, in this chapter for more information. (This question deals with objective Planning 1.)
2. One user, one account, centralized administration, universal resource access, synchronization. See section, Windows NT Server 4 Directory Services, in this chapter for more information. (This question deals with objective Planning 1.)
6. Local accounts cannot be given permissions across trusts. See section, Accounts in Trust Relationships, in this chapter for more information. (This question deals with Planning 1.)

Answers and Explanations: For each of the Review and Exam questions, you will find thorough explanations located at the end of the section.

Exam Questions: These questions reflect the kinds of multiple-choice questions that appear on the Microsoft exams. Use them to become familiar with the exam question formats and to help you determine what you know and what you need to review or study more.

Suggested Readings and Resources

The following are some recommended readings on the subject of installing and configuring NT Workstation:

1. Microsoft Official Curriculum course 770: *Installing and Configuring Microsoft Windows NT Workstation 4.0*
   - Module 1: Overview of Windows NT Workstation 4.0
   - Module 2: Installing Windows NT Workstation 4.0
2. Microsoft Official Curriculum course 922: *Supporting Microsoft Windows NT 4.0 Core Technologies*
   - Module 2: Installing Windows NT
   - Module 3: Configuring the Windows NT Environment
3. *Microsoft Windows NT Workstation Resource Kit Version 4.0* (Microsoft Press)
   - Chapter 2: Customizing Setup
   - Chapter 4: Planning for a Mixed Environment
4. Microsoft TechNet CD-ROM
   - *MS Windows NT Workstation Technical Notes*
     - MS Windows NT Workstation Deployment Guide – Automating Windows NT Setup
     - An Unattended Windows NT Workstation Deployment
5. Web Sites
   - `www.microsoft.com/train_cert`

Suggested Readings and Resources: The very last element in each chapter is a list of additional resources you can use if you wish to go above and beyond certification-level material or if you need to spend more time on a particular subject that you are having trouble understanding.