{ewc MVIMG, MVIMAGE,!welcome.shg}

This course includes lab exercises that give you experience using the skills presented in the chapters. Most of the labs contain code to get you started and code for the lab solution. You can access the labs from within each chapter, or by clicking **Labs** on the **Contents** menu.

Lab Setup

If you selected the Complete installation option during Setup, the lab files are copied to the Labs folder in the location where you installed the application. If you selected the Typical installation option, the lab files will not be copied to your local computer. To install the lab files on your local computer, re-install the application, and select the Complete installation option during Setup.

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

The Visual Basic development environment contains all the resources you need to build powerful Windows-based programs quickly and efficiently. This chapter introduces you to the features and capabilities of the Visual Basic 6.0 Learning Edition Development System, helps you get started with Visual Basic, and describes the various Visual Basic tools and windows that are available to you.

In this chapter, you'll learn how to:

® Explore and configure the Visual Basic development environment.

® Build your first program.

® Create an executable (.EXE) file.

If you haven’t written a program before, you might wonder just what a program is and how to create one in Visual Basic. This section provides an overview of the Visual Basic programming process.

A program is a set of instructions that collectively cause a computer to perform a useful task, such as processing electronic artwork or managing files on a network. A program can be quite small — something designed to calculate a home mortgage — or it can be a large application, such as Microsoft Excel.

A Visual Basic program is a Windows-based application that you create in the Visual Basic development environment. This section includes the following topics:

The first step in programming is determining exactly what you want your program to accomplish. This sounds simple (isn’t it obvious?), but without a mission statement, even the best programmer can have trouble building an application he or she is happy with.

Planning a program is a little like planning a barbecue. For a barbecue to go off smoothly, you need to prepare for it ahead of time. You need to organize the menu, invite your friends, buy the food, and (most likely) clean your house. But a barbecue can be entertaining if friends just happen to drop by and bring stuff. Programs, though, usually don’t turn out the best if they’re built with the stone-soup approach.

This section includes the following topics:

When you plan Visual Basic programming projects, you might find it useful to ask yourself the following questions about your program:

® What is the goal (mission) of the program I am writing?

® Who will use the program?

® What will the program look like when it starts?

® What information will the user enter in the program?

® How will the program process the input?

® What information (output) will the program produce?

{ewc mvimg, mvimage,!tip.bmp}

When you finish this preliminary work, you'll be ready to start building the program with Visual Basic.

Long before you sit down in front of your computer, you should spend some time thinking about the programming problem you are trying to solve. Up-front planning will save you development time down the road, and you'll probably be much happier with the result. One part of the planning process might be creating an ordered list of programming steps, called an algorithm.

To view an illustration that shows you how planning a barbecue can be transformed into an algorithm, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Building a Windows-based application with Visual Basic involves three programming steps: creating the user interface, setting the properties, and writing the code. And, of course, your project must be saved.

These steps are described in the following topics:

® Creating the User Interface

® Setting the Properties

® Writing Program Code

® Saving a Project

After you have established a clear goal for your program, it's important to think about how it will look and how it will process information. The complete set of forms and controls used in a program is called the program user interface. The user interface includes all the menus, dialog boxes, buttons, objects, and pictures that users see when they operate the program. In the Visual Basic development environment, you can create all the components of a Windows-based application quickly and efficiently.

A Successful Windows Application

To view an animation that demonstrates the components of a successful Windows-based application, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

You can read about the official interface design guidelines for Windows in *The Windows Interface Guidelines for Software Design* (Microsoft Press, 1995).

Properties are programmable characteristics associated with forms and their controls. You can set these properties either as you design your program (at design time) or while you run it (at run time). You change properties at design time by selecting an object, clicking the Properties window, and changing one or more of the property settings. To set properties at run time, you use Visual Basic program code.

To view an animation that uses a bicycle analogy to describe property settings in a program, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

You finish building your program by typing program code for one or more user interface elements. Writing program code gives you more control over how your program works than you can get by just setting properties of user interface elements at design time. By using program code, you completely express your thoughts about how your application:

- Processes data
- Tests for conditions
- Changes the order in which Visual Basic carries out instructions.

The Visual Basic Programming Language

The Visual Basic programming language contains several hundred statements, functions, and special characters. However, most of your programming tasks will be handled by a few dozen, easy-to-remember keywords.

In this course, you'll spend a lot of time exploring the subtleties of writing useful program code that you can adapt to a variety of situations. For now, though, just keep these points in mind:

- Program code follows a particular form (syntax) required by the Visual Basic compiler.
- You enter and edit program code in the Code window, a special text editor designed to track and correct (debug) program statement errors.

Visual Basic for Applications

The Visual Basic programming language is not unique to the Visual Basic software you're using. Microsoft Visual Basic for Applications is included in these products:

- Microsoft Excel
- Microsoft Word
- Microsoft Access
- Microsoft PowerPoint
- Microsoft Project

If you've used Visual Basic for Applications with the most recent version of one of these applications, or if you've had some experience with Basic in the past, you'll recognize most of the keywords, statements, and functions in Visual Basic 6.0.

After you complete a program or find a good stopping point, you should save the project to disk with the **Save Project As** command on the **File** menu.

To view an illustration of the **Save File As** dialog box, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Saving to Disk

Saving a Visual Basic project to disk is a little more complicated than saving a Word or Excel document. In addition to the project (.vbp) file, Visual Basic also creates a separate file for each form (.frm file) and (if necessary) for each standard code module (.bas file) in your project. You must save each of these files, one by one.

When you choose the **Save Project As** command, Visual Basic prompts you for each file name, one at a time. The process concludes when the project assembly instructions (which serve as the program's packing list) are saved in a project file. The file names included in this packing list are the components listed in the Project window.

Note You can recognize project files by their .vbp file name extension.

Reusing Component Files

You can use component files associated with a project in future programming projects by using the **Project** menu **Add File** command. This command merges forms and standard code modules into the current programming project and makes them appear in the current program Project window.

After you create a working version of your program, you need to test it carefully to verify that it works correctly. If you wish to distribute your program, you also need to compile it into an executable program (a stand-alone Windows-based program) and give it to your users. If you decide to revise the program later, you repeat the process, beginning with planning and an analysis of any new goals. These steps complete the software development life cycle.

To view an illustration that summarizes the software development life cycle, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

The following topics are included in this section:

® Testing and Debugging

® Compiling

® Distributing

Testing a program involves checking it against a variety of real-life operating conditions to determine whether it works correctly. A problem that stops the program from running or from producing the expected results is called a software defect (bug).

If you’ve used software for any length of time, you’ve probably run into your share of computer glitches caused by faulty software. Now that *you* are the programmer, it’s your job to stamp out these problems before they reach the end user. Fortunately for all of us, Visual Basic provides some excellent debugging tools to catch bugs. You’ll learn how to find and correct bugs in Chapter 5: Controlling Flow and Debugging.

When you’ve finished creating and testing your program, you can compile it into an executable (.exe) file that will run under Windows or Windows NT. Creating an executable file with Visual Basic is as simple as clicking the **Make Project1.exe** command on the **File** menu.

If you plan to run your new Visual Basic application only on your own system, creating the executable file will be your final step. When you want to run your new program, simply double-click the program’s .exe file in Windows Explorer. Or, use the file to create a shortcut icon that you can place on your Windows desktop.

You might want to share a Visual Basic executable file with friends or colleagues or sell your program. If so, you need to put the program and a few necessary support files on a disk (or the network), where your users or customers can get at them. All Visual Basic programs require one or more dynamic-link library (DLL) files to run. To get your users up and running, you need to copy the necessary files from your hard disk to a distribution disk. The exact DLL files you need depends on these factors:

® The operating system that your users run.

® The Visual Basic features that your program uses.

The *Visual Basic Programmer's Guide* (in MSDN Library Help) discusses the files you need to create your distribution disks and how to use the Package and Deployment Wizard. (Search for the index item *distributing, Visual Basic's Package and Deployment Wizard*.) For now, though, be assured that when you create Visual Basic programs, you and your users will be able to enjoy the benefits of their new software immediately.

This section helps you get Visual Basic loaded and running, and shows you how to control the development environment elements.

The following topics are included in this section:

Before you can work with a Visual Basic program, you need to load the program into memory, just as you would load a word processing document in a word processor for editing.

u **To load a Visual Basic program into memory and run it**

1. On the **File** menu, click **Open Project**.

   The **Open Project** dialog box appears. With this dialog box, you can open any existing Visual Basic program on your hard disk, attached network drive, CD-ROM, or floppy disk.

2. If necessary, use the **Look In** drop-down list box and the **Up One Level** button to locate the program you want to load. Then, double-click the program name.

   The project file loads the Visual Basic user interface form, properties, and program code. (Visual Basic project files are distinguished by the .VBP file name extension.)

3. If the program user interface does not appear, open the Forms folder in the Project window, select the first form, and then click **View Object** in the Project window.

   This is an optional but useful step, especially if you want to look at the program user interface in the Form window before you run it.

4. On the Visual Basic Standard toolbar, click **Start** to run the program.

   The toolbox and several of the other windows disappear, and the Visual Basic program starts to run.

5. On the toolbar, click **End** when you want to exit the program.

   To view a demonstration showing what happens when you run a Visual Basic program, click this icon.
   {ewc mvimg, mvimage,!democlip.bmp}

With several windows and programming tools available at the same time, the Visual Basic programming environment can become a pretty crowded place. With Visual Basic 6.0, you have complete control over the shape and size of each element in the development environment. You achieve this control by moving, docking, and resizing each of the programming tools and windows.

Docking Windows

In Visual Basic 6.0, you can align and attach (dock) windows to make all the elements of the programming system visible and accessible. If you align one window along the edge of another, the windows will attach to each other. The advantage of dockable windows is that they always remain visible — they won't become hidden behind tools or other windows.

Moving Windows and Tools

To move a window, the toolbox, or the toolbar, simply click the title bar and drag the window to a new location.

Resizing Windows

If you want to see more of a docked window, simply drag one of the borders to resize the window. If you get tired of docking and want the windows and tools to overlap each other, follow this procedure:

u **To undock windows and tools**

1. On the **Tools** menu, click **Options**.
2. Click the **Docking** tab.
3. Clear the check box of each tool you want to stand on its own.

To view a demonstration that shows you how to move, dock, and resize windows in the Visual Basic development environment, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

The first step in using Visual Basic is launching it and opening existing files or creating new ones.

u **To start Visual Basic 6.0**

1. In Windows, click **Start**, point to Programs, and point to the Microsoft Visual Basic 6.0 folder.

   The icons in the folder appear in a list.
2. Click the **Microsoft Visual Basic 6.0** program icon.

   The **New Project** dialog box appears. This dialog box prompts you for the type of programming project you want to create.

   To view an illustration of a new project dialog box, click this icon.
   {ewc mvimg, mvimage,!illust.bmp}
3. To accept the default new project, click **OK**.

   In the Visual Basic development environment, a new project (a standard, 32-bit Visual Basic application) and the related windows and tools open.

   To view an illustration of the Visual Basic development environment, click this icon.
   {ewc mvimg, mvimage,!illust.bmp}

The Visual Basic development environment contains these programming tools and windows, with which you construct your Visual Basic programs:

® Menu bar

® Toolbars

® Visual Basic toolbox

® Form window

® Properties window

® Project Explorer

® Immediate window

® Form Layout window

The exact size and shape of the windows depends on how your system has been configured. In Visual Basic 6.0, you can align and attach (dock) the windows to make all the elements of the programming system visible and accessible. You'll learn how to customize your development environment in Moving, Docking, and Resizing Windows.

This section contains information about all the tools and windows available to help you construct Visual Basic programs. These resources are described in the following topics:

When you start Visual Basic, a default form (Form1) with a standard grid (a window consisting of regularly spaced dots) appears in a pane called the Form window. You can use the Form window grid to create the user interface and to line up interface elements.

To view an illustration of the Form window, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Building Interface Elements

To build the interface elements, you click an interface control in the Visual Basic toolbox, and then you draw the user interface element on your form by using the mouse. This process is usually a simple matter of clicking to position one corner of the element and then dragging to create a rectangle the size you want. After you create the element — say, a text box — you can refine it by setting properties for the element. In a text box, for example, you can set properties to make the text boldface, italic, or underlined.

Adjusting Form Size

You can adjust the size of the form by using the mouse — the form can take up part or all of the screen.

{ewc mvimg, mvimage,!tip.bmp}

Controlling Form Placement

To control the placement of the form when you run the program, adjust the placement of the form in the Form Layout window.

You use special tools, called controls, to add elements of a program user interface to a form. You can find these resources in the toolbox, which is typically located along the left side of the screen. (If the toolbox is not open, display it by using the **Toolbox** command on the **View** menu.) By using toolbox controls, you can add these elements to the user interface:

® Artwork

® Labels and text boxes

® Buttons

® List boxes

® Scroll bars

® File system controls

® Timers

® Geometric shapes

® Data and OLE controls.

To view an illustration of the standard contents of the toolbox, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Visible and Invisible Controls

When a Visual Basic program runs, most toolbox controls operate like the standard objects in any Windows-based application — and they will be visible to the user. However, the toolbox also contains controls that can be used to perform special, behind-the-scenes operations in a Visual Basic program. The powerful objects you create with these controls do useful work but can be made invisible to the user when the program runs. These objects can be used for:

® Manipulating database information.

® Working with Windows-based applications.

® Tracking the passage of time in your programs.

With the Properties window, you change the characteristics (property settings) of the user interface elements on a form. A property setting is a characteristic of a user interface object. For example, you can change the text displayed by a text box control to a different font, point size, or alignment. (With Visual Basic, you can display text in any font installed on your system, just as you can in Microsoft Excel or Microsoft Word.)

Displaying the Properties Window

To display the Properties window, click the **Properties Window** button on the toolbar. If the window is currently docked, you can enlarge it by double-clicking the title bar. To redock the Properties window, double-click its title bar again.

To view an illustration of the Properties window, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Properties Window Elements

The Properties window contains the following elements:

® A drop-down list box at the top of the window, from which you select the object whose properties you want to view or set.

® Two tabs, which list the properties either alphabetically or by category.

® A description pane that shows the name of the selected property and a short description of it.

Changing Property Settings

You can change property settings by using the Properties window while you design the user interface or by using program code to make changes while the program runs.

A Visual Basic program consists of several files that are linked together to make the program run. The Visual Basic 6.0 development environment includes a Project window to help you switch back and forth between these components as you work on a project.

To view an illustration of the Project window, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Project Window Components

The Project window lists all the files used in the programming process and provides access to them with two special buttons: **View Code** and **View Object**.

Displaying the Project Window

To display the Project window, click the **Project Explorer** button on the Visual Basic toolbar. If the window is currently docked, you can enlarge it by double-clicking the title bar. To redock the Project window, double-click its title bar again.

Adding and Removing Files

The project file maintains a list of all the supporting files in a Visual Basic programming project. You can recognize project files by their .vbp file name extension.

You can add individual files to and remove them from a project by using commands on the Project menu. The changes that you make will be reflected in the Project window.

Note   In Visual Basic versions 1 through 3, project files had the .mak file name extension. In Visual Basic versions 4, 5, and 6.0, project files have the .vbp file name extension.

Adding Projects

If you load additional projects into Visual Basic with the **File** menu **Add Project** command, outlining symbols appear in the Project window to help you organize and switch between projects.

Visual Basic includes an online reference system called the MSDN Library that you can use to learn more about the Visual Basic development environment and programming language. You can get Help through the MSDN Library in several ways:

To get Help information	Do this
By topic or activity	On the **Visual Basic Help** menu, click **Contents** to open the MSDN Library.
While working in the Code window	Select the keyword or program statement you're interested in and press F1.
While working in a dialog box	Click the **Help** button in the dialog box.
By searching for a specific keyword	On the **Help** menu, click **Search** and type the term you're looking for in the MSDN Library Search tab.
By connecting to a Web page with information about Visual Basic or programming	On the **Help** menu, point to the **Microsoft on the Web** submenu, and then click the topic or location you're interested in.
About contacting Microsoft for product support	On the **Help** menu, click **Technical Support**.

The location and purpose of the Visual Basic 6.0 programming tools are described in the following table:

Tool	**Purpose**
Menu bar	Located at the top of the screen, the menu bar provides access to the commands that control the Visual Basic programming environment. Menus and commands work according to standard conventions used in all Windows-based programs. You can use these menus and commands by using keyboard commands or the mouse.
Toolbars	Located below the menu bar, toolbars are collections of buttons that serve as shortcuts for executing commands and controlling the Visual Basic development environment. You can open special-purpose toolbars by using the **View** menu **Toolbars** command.
Windows taskbar	This taskbar is located along the bottom of the screen. You can use the taskbar to switch between Visual Basic forms as your program runs and to activate other Windows-based programs.

You can create much of your program by using controls and setting properties. However, most Visual Basic programs require additional program code that acts as the brains behind the user interface that you create. This computing logic is created using program statements — keywords, identifiers, and arguments — that clearly spell out what the program should do each step of the way.

You enter program statements in the Code window, a special text editing window designed specifically for Visual Basic program code. You can display the Code window in either of two ways:

® By clicking **View Code** in the Project window.

® By clicking the **View** menu **Code** command.

The Immediate window is a programming tool that helps you debug (correct errors in) a program while you are in break mode. When you type a question mark and statement in the Immediate window and then press ENTER, Visual Basic executes the statement and displays the output in the Immediate window. With this feature, you can test the value of specific key variables as your program runs.

The Form Layout window is a visual design tool. With it, you can control the placement of the forms in the Windows environment when they are executed. When you have more than one form in your program, the Form Layout window is especially useful — you can arrange the forms onscreen exactly the way you want.

To position a form in the Form Layout window, simply drag the miniature form to the desired location in the window.

To view an illustration of the Form Layout window, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

In previous sections, you became acquainted with the Visual Basic development environment and the Visual Basic programming process. Now, it’s time to learn how to build a simple program. The Windows-based application that we’ll create is Lucky Seven, a game program that simulates a lucky-number slot machine.

This section includes the following topics:

Before you start building the Lucky Seven program, think in general terms about what the program should be like and the steps required to construct it. Lucky Seven will be a game program that simulates the random spins of a Las-Vegas-style slot machine. Think carefully about how slot machines are played.

To view an animation illustrating the planning process, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

In Visual Basic, you create the program user interface by placing toolbox controls on a form. As you create each new control, it appears on the form as an object surrounded by selection handles. (When you place a control on a form, the control becomes an object.) You can use the mouse to resize the object with the selection handles or to move it to a new location on the form.

Each project maintains its own unique toolbox. You add and remove ActiveX controls from the toolbox by using the **Project** menu **Components** command. You cannot remove the 20 standard controls from the toolbox.

To view a demonstration showing you how to create the seven objects on the Lucky Seven interface, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

As you learned earlier in the chapter, properties are programmable characteristics associated with the objects on your form. You can set object properties when you are designing your program (at design time) by selecting an object, clicking the Properties window, and changing one or more of the property settings. You can also set object properties while your program runs (at run time) by modifying object properties with Visual Basic program code.

The Lucky Seven program contains both types of property settings.

To view a demonstration showing you how to set Lucky Seven properties at design time, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

The following demonstration shows how you enter program code in the Code window to complete the Lucky Seven program. Since the **Spin** and **End** buttons drive the program, you create blocks of code (event procedures) for each of those buttons.

To view a demonstration showing you how to write Lucky Seven program code, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

The Lucky Seven program contains a form file, Lucky.frm and a project file, Lucky.vbp. To create these files, go to the **File** menu commands. Then, click **Save Form As** and **Save Project As**. After you save a project, you can close Visual Basic without losing your work.

To run a Visual Basic program in the Visual Basic development environment, you can use any of these methods:

® On the **Run** menu, click **Start**.

® Click **Start** on the toolbar.

® Press F5.

When a Visual Basic program runs, the toolbox, Properties window, and Form Layout window disappear, and several of the Visual Basic menu and toolbar commands appear dimmed. The objects in your program user interface become active, and you have an opportunity to experiment with the functionality of your program.

To view a demonstration of the Lucky Seven program, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

After you determine that the Lucky Seven program runs correctly, you may wish to create an executable file. With this stand-alone program, you (or another user) can run your program without starting the Visual Basic programming system. A Visual Basic 6.0 executable file is a 32-bit application specifically designed to run on the Windows or Windows NT operating systems.

If you want to distribute your executable (.exe) files, you need to create distribution disks that include a setup program and the necessary support files. This is easily accomplished by running the Package and Deployment Wizard, located in the Microsoft Visual Basic 6.0 Tools folder. (For more information, consult the online help included with the Package and Deployment Wizard.)

u **To create a standard, 32-bit executable file**

1. On the **File** menu, click **Make Project1.exe**.
2. To adjust the default settings, click **Options** in the **Make Project** dialog box that opens and make your choices.

   Use this optional step to change the program icon, version number, or advanced compiler options.
3. Specify a folder and file name for your executable file with the **Save In** drop-down box and **File Name** text box and click **OK** to compile your executable file.

If Visual Basic displays an error while your program runs, you may have a typing mistake in your program code. Or, a logic error may have prevented your program from performing correctly. These errors require debugging — searching for and correcting program errors.

After creating the Lucky Seven program, it's a good idea to watch the behavior of the program carefully and verify that it meets your programming goals. Here's a list of testing questions that demonstrates this principle:

® Does clicking the **Spin** button make three new numbers appear in the Spin windows?

® Are the numbers really random (non-repeating), and do their values range from zero to nine?

® When one or more sevens appear, does the coin stack appear?

® When a seven does not appear, does the coin stack disappear?

® Does the **End** button work?

In this lab, you will create a simple tool that displays the time whenever the user clicks a command button called Time. Your program will also have a **Quit** button, so that users can gracefully exit the application when they are finished. As you create this basic application, you will review the fundamental programming skills you have learned in Chapter 1.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **30 minutes**

Objectives

After completing this lab, you will be able to:

® Create a user interface for a program with toolbox controls.

® Set object properties with the Properties window.

® Write program code in the Code window.

® Run and compile a program.

Lab Setup

To complete the exercises in this lab, you must have Visual Basic 6.0 installed on your system. You can verify this by clicking the **Start** button, pointing to the **Programs** folder, pointing to the **Microsoft Visual Basic 6.0** folder, and clicking the **Visual Basic 6.0** program icon. When you click the program icon, Visual Basic should start and display the **New Project** dialog box.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

These exercises provide practice working with the concepts and techniques covered in this chapter.

® Exercise 1: Creating a User Interface

In this exercise, you create the Time program user interface by adding one label and two command buttons to a form.

® Exercise 2: Setting User Interface Properties

In this exercise, you use the Properties window to change the **Caption**, **Font**, and **Alignment** properties of the objects on your form.

® Exercise 3: Writing the Code

In this exercise, you use the Code window to write event procedures for the **Time** and **Quit** command buttons.

® Exercise 4: Compiling and Running the Program

In this exercise, you run the Time program to verify that it works correctly. Then, you compile the program as an executable (.exe) file.

In this exercise, you create the Time program user interface by adding one label and two command buttons to a form.

u To open a Visual Basic application

1. Click **Start**, point to **Programs**, point to **Microsoft Visual Basic 6.0**, and then click the **Visual Basic 6.0** program icon.
2. In the **New Project** dialog box, click **OK** to create a standard Visual Basic program.
3. Identify the elements of the Visual Basic development environment.

u To create objects on the form

1. In the toolbox, click **Label**, and then position the mouse pointer in the middle of the Form window.

   You can determine the name of a toolbox control by holding the mouse pointer over it and waiting for the ToolTip to appear.
2. On the form, create a large label by holding down the left mouse button, dragging down and to the right, and then releasing the mouse button.
3. To create a command button on the form, double-click **CommandButton** in the toolbox, and then drag it below the left side of the label.
4. To create a second command button, double-click **CommandButton**, and then drag it below the right side of the Label.
5. If necessary, resize the objects with the sizing pointer.
6. From the **File** menu, click **Save Project As**.
7. Save your form and project to disk under the name **Time**. Visual Basic will prompt you for two file names — one for your form file (Time.frm), and one for your project file (Time.vbp).

In this exercise, you use the Properties window to change the **Caption**, **Font**, and **Alignment** properties of the objects on your form.

u To set object properties

1. In the Properties window, click the drop-down arrow, click **Label1**, and then double-click the title bar to restore it to full size.

   If the Properties window is not visible, display it by going to the toolbar and clicking the Properties window or press F4.
2. At the top of the alphabetic list of properties, click **Alignment**. Then, click the drop-down arrow and change the property setting to **Center**.
3. Scroll down the properties list until the **Caption** property is visible. Then, change the caption value to **00:00:00 AM**.
4. Click the **Font** property, and then click the dialog box button in the **Value** box.
5. In the **Font** dialog box, change the font to **24-point bold**, and then click **OK**.

   If Label1 is not big enough to display 00:00:00 AM in the 24-point font, resize the object so that all of the text is visible.
6. Click **Command1**, and then change its **Caption** property to **Time**.
7. Click **Command2**, and then change its **Caption** property to **Quit**.
8. Close the Properties window or dock it to reduce its size. You can also double-click the title bar to return it to its original location.

In this exercise, you use the Code window to write event procedures for the **Time** and **Quit** command buttons.

u **To write the code**

1. Double-click the **Command1** object to open the Command1_Click event procedure in the Code window.

   If the Code window is too small to work in, you can resize it with the sizing pointer.
2. Type the following program statement below the **Private Sub** statement:
   ```
   Label1.Caption = Time
   ```
3. At the top-left of the Code window, click the **Object** drop-down list box (which currently contains the object name Command1), and then click the **Command2** object.
4. Type the following program statement below the Private Sub statement:
   ```
   End
   ```
5. Close the Code window.
6. On the toolbar, click **Save Project** to save your program to disk.

In this exercise, you run the Time program to verify that it works correctly. Then, you compile the program as an executable (.exe) file. In this simple program, the time is updated only when the user clicks **Time**. (You'll learn how to use the **Timer** control to create a digital clock that updates continuously without user interaction in Chapter 6: Using Loops and Timers.

u To run the program in the Visual Basic development environment

1. Point to the Visual Basic toolbar, and then click **Start**.
2. Click **Time** and verify that the current system time appears in Label1. You should be able to see the entire system time, down to the second.
3. Wait a few moments, and then click **Time** again. A new system time should appear.
4. When you are satisfied that the program runs correctly, click **Quit** to exit the program.

u To compile the program

1. From the **File** menu, click **Make Time.exe**. (Visual Basic automatically adds the file name to the File menu.)
2. In the **Make Project** dialog box, specify the location for your .exe file.
3. To build a standard, 32-bit Windows-based application, click **OK**.

If you plan to use your new executable file often, you can create a shortcut to it and place it on the Windows desktop.

u To create a shortcut icon

1. Click the Windows desktop with the right mouse button.
2. Point to **New**, and then click **Shortcut**.
3. Use the **Create Shortcut** dialog box to specify the location of the .exe file.

1. When you create a Visual Basic program, what's the first thing you should do?

{ewc mvimg, mvimage, !answer.bmp} A. Set the properties of the objects in the program user interface.

{ewc mvimg, mvimage, !answer.bmp} B. Write the program code for one or more event procedures.

{ewc mvimg, mvimage, !a C. Determine the requirements of the program and plan the user interface.

nswer.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. Compile the program with the **File** menu **Make** command.

2. In the Visual Basic development environment, which of the following is *not* a programming tool?

{ewc mvimg,mvimage,!answer.bmp} A. Toolbox.

{ewc mvimg,mvimage,!answ B. Properties window.

er.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. Immediate window.

{ewc mvimg, mvimage,!answer.bmp} D. Control Panel.

3. What is the purpose of the Form window?

{ewc mvimg, mvimage,!answer.bm A. To hold the program code in your program event procedures.

p}

{ewc mvimg, mvimage, !answer.bmp} B. To provide a place for you to design the program user interface. It contains objects and other window elements.

{ewc mvimg, mvimage, !answer.bmp} C. To provide a place to create and edit formulas and other mathematical equations.

{ewc mvimg, mvimage, !an D. To provide a place to configure the dockable windows and tools in the Visual Basic development environment.

swer.bmp}

4. What can you do in the Properties window?

{ewc mvimg, mvimage,!answer.bmp} A. Change the property settings of objects while your program runs.

{ewc mvimg, mvimage,!answer.bmp} B. Change the object property settings while you create your program.

{ewc mvimg, mvimage,!answer C. Control which files are saved with your Visual Basic project.

.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. Write program code for the objects in your program.

5. Which of the following is *not* a Project window feature?

{ewc mvimg,mvimage,!answer.bmp} A. A drop-down list box, with which you set the properties of the highlighted file.

{ewc mvimg,mvimage,!answer.bm B. Buttons that let you switch between code view and object view of project components.

p}

{ewc mvimg, mvimage,!answer.bmp} C. Outlining symbols that let you view the relationships between components in one or more projects.

{ewc mvimg, mvimage,!answer.bmp} D. Labels that show you the file names of each component in the project.

6. Which toolbox control is typically used to display text on a form?

{ewc mvimg, mvimage,!answer.bmp} A. CommandButton.

{e B. PictureBox.

w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

{e C. Label

w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

{e D. Timer.

w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w

er.bmp}

7. What is program code?

{ewc mvimg, mvimage,!answer.bmp} A. A collection of special keywords that programmers use to encrypt their programs.

{ewc mvimg, mvimage,!answer.bmp} B. The list of file names that is included in your project and stored in a file with a .vbp file name extension.

{ewc mvimg, mvimage,!answer.bm C. A summary description of your program, written in Norwegian.

p}

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

D. Written instructions for the compiler that are typed in the Code window and that use a language called Visual Basic.

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter introduces the standard Visual Basic toolbox controls and teaches you how to use them to build useful interface features. In this chapter, you will learn how to:

® Name Visual Basic objects.

® Use basic controls to display text and process input.

® Use file system controls to browse the files and folders on your computer.

® Use data input controls to display lists and check boxes.

® Use Data and OLE controls to work with Microsoft Office applications.

® Install ActiveX controls.

As you learned in Toolbox Controls, controls are the design tools you use to build the user interface of a Visual Basic program. After you place controls on a form, they become objects that you can customize with property settings and program code.

In this section, you learn about the most basic user interface controls in the Visual Basic toolbox. This section includes the following topics:

Label, the simplest control in the Visual Basic toolbox, displays formatted text on a user interface form. Typical uses for the **Label** control include:

® Help text

® Program splash screen headings

® Formatted output, such as names, times, and dates

® Descriptive labels for other objects, including text boxes and list boxes.

Creating Labels on a Form

This illustration shows a form with some labels created with the **Label** control. To view the illustration, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Creating Labels in Code

You can also set label properties with program code, as shown in this code window. To view the code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

To review the process of creating objects on a form, see Form Window.

To review how to set object properties, see Setting the Properties.

To see an illustration that describes each of the standard toolbox controls, see Toolbox Controls.

To view a demonstration of the **Label** control in action, see Demonstration: Basic Controls in Action.

The **TextBox** control is one of the most versatile tools in the Visual Basic toolbox. This control performs two functions:

® Displaying output (such as operating instructions or the contents of a file) on a form.

® Receiving text (such as names and phone numbers) as user input.

How a text box works depends on how you set its properties and how you reference the text box in your program code. The following popup illustration shows a text box that:

® Displays data entry instructions for the user.

® Receives input when the user types in a delivery note and clicks the **Print Order** button.

To view the illustration, click this icon
{ewc mvimg, mvimage,!illust.bmp}

To view sample code showing how to control a text box, click this icon.
{ewc mvimg, mvimage,!code.bmp}

To view the **TextBox** control in action, see Demonstration: Basic Controls in Action.

As you saw in Chapter 1, you use the **CommandButton** control to create buttons with a variety of uses on a form. A command button is the most basic way to get user input while a program is running. By clicking a command button, the user requests that a specific action be taken in the program. Or, in Visual Basic terms, clicking a command button creates an event, which must be processed in your program.

Here are some command buttons that you'd typically find in a program:

OK	Accepts a list of options and indicates that the user is ready to proceed.
Cancel	Discards a list of options.
Quit	Exits an open dialog box or program.

In each case, you should use command buttons carefully so that they work as expected when they are clicked.

Changing Command Button Properties

You can change command button properties (like those of all objects) in two ways:

® By adjusting property settings in the Properties window.

® By changing properties with program code.

To view an illustration of a form with two command buttons, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

To view a demonstration of the **CommandButton** control in action, see Demonstration: Basic Controls in Action.

To view a demonstration showing the **Label**, **TextBox**, and **CommandButton** controls in action, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

The Visual Basic toolbox provides these very useful controls, with which you can gain access to the file system. This table lists the names and purpose of these controls:

Control	Purpose
DriveListBox	Helps you browse the valid drives on a system.
DirListBox	Helps you navigate the folders on a particular drive.
FileListBox	Helps you select a specific file in a folder.

In this section, you learn how to use these file system controls to create a simple browser utility that locates and displays files stored on your computer.

This section includes the following topics:

® DriveListBox

® DirListBox

® FileListBox

® Demonstration: Combining File System Controls

The **DriveListBox** control creates a drop-down box object that displays:

® The current drive and volume label on your form.

® A new drive when users search for a file.

You can use this control on its own to select a drive to store files. Or, you can use it with **DirListBox** and **FileListBox** to create a file browser.

When you create a **DriveListBox** object on a form, Visual Basic includes the current drive and volume label in the object. Displaying this information automatically has two advantages:

® The user can see the current drive.

® You can size the object correctly so that the user can see the drive letter and label.

To view an illustration of a new drive list box on a form, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

To view a demonstration showing you how to combine the **DriveListBox** control with other file system controls, see Demonstration: Combining File System Controls.

DirListBox displays a scrollable list box, with which users set the current folder on a drive. In this way, its purpose complements that of the **DriveListBox** control. You can use the **DirListBox** control to:

® Display the current folder.

® Help users explore the file system when they save files or search for documents.

When you create a directory list box on your form, Visual Basic automatically displays the folders on the current drive as they appear when the program runs. You can use this information to size the directory list box object correctly. (Be sure to allow room for at least four or five folders.)

To view an illustration of a directory list box on a form, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

To view a demonstration showing you how to combine the **DirListBox** control with other file system controls, see Demonstration: Combining File System Controls.

With **FileListBox**, users select a specific file in the file system. **FileListBox** displays the files in the current folder, so you can size the object correctly while you design the program.

To view an illustration of a form that combines a file list box with drive and directory list boxes, click this icon. {ewc mvimg, mvimage,!illust.bmp}

To view a demonstration showing you how to combine the **FileListBox** control with other file system controls, see Demonstration: Combining File System Controls.

If you plan to combine drive, directory, and file list boxes, you need to use a few lines of program code to exchange information between the objects while your program runs. Although you haven’t seen much program code yet in this course, the syntax of these statements is quite straightforward.

This demonstration shows you how to combine file system controls to create a browser utility that displays artwork files stored on your system. You'll also see how to use the **Drive**, **Path**, and **FileName** properties of the file system controls to communicate between objects while a program runs.

To view the demonstration, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Visual Basic provides several other controls that you can use to collect user input in a program. In this section, you learn to use four versatile toolbox controls.

This section includes the following topics:

{ewc mvimg, mvimage,!tip.bmp}

To receive user input in a Visual Basic program, you can use controls that:

® Present a range of acceptable choices.

® Receive input in an acceptable format.

The **OptionButton** control satisfies these requirements. It displays a list of choices on a form and requires the user to pick one (but only one) item from the list.

Creating Option Buttons

Like the other items in the Visual Basic toolbox, you create an option button on a form by clicking **OptionButton** in the toolbox and then drawing on the form.

Grouping Option Buttons

If you want to create a group of option buttons that work together, you must first create a frame for the buttons. (To do this, use **Frame**, a Visual Basic toolbox control.) Next, place your option buttons inside the frame so that they can be processed as a group in your program code and moved as a group along with the frame.

To view an illustration that shows a group of option buttons surrounded by a frame, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

To view a demonstration featuring these controls, see Demonstration: Input Controls in Action.

The **CheckBox** control is similar to the **OptionButton** control, except that **CheckBox** displays a list of choices and gives the user the option to pick multiple items (or none at all) from the list.

You create a check box on a form much as you would make an option button. Start by clicking the **CheckBox** control in the toolbox, and then draw on the form. If you want to create a group of check boxes that work together, create a frame for the buttons with Frame, a toolbox control. Then, you can place your check boxes inside the frame so that they can be processed as a group in your program code or moved as a group along with the frame.

To view an illustration showing a group of check boxes surrounded by a frame, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

To view a demonstration featuring these controls, see Demonstration: Input Controls in Action.

When you want a user to choose a single response from a long list of choices, you use the **ListBox** control. (Scroll bars appear if the list is longer than the list box.) Unlike option buttons, however, list boxes don't require a default selection.

To view an illustration showing a list box with room for six or seven visible items, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Creating List Boxes

Building a list box is simple, since it doesn’t require creating a frame or a separate object for each list item. You just click **ListBox** in the Visual Basic toolbox and draw a suitably sized list box on your form.

Setting List Box Properties

In a Visual Basic program, you can define list box characteristics in the Properties window or by using program code, so that users can add, remove, or sort list box items while the program runs.

You add choices to a list box with the **AddItem** method, which is typically placed in the Form_Load event procedure. (**AddItem** is a method normally used with list-oriented objects).

To view sample code showing an example of the **AddItem** method, click this icon.
{ewc mvimg, mvimage,!code.bmp}

Adding Default Program Statements

Each time your Visual Basic program displays a form, the Form_Load event procedure (a block of code) runs. Although Form_Load doesn’t contain default program statements, you can add statements to it. Just double-click the form to open the Form_Load event procedure in the code window. You’ll learn more about the ins and outs of program code in Chapters 4: Visual Basic Variables and Operators, Chapter 5: Controlling Flow and Debugging, and Chapter 6: Using Loops and Timers.

To view a demonstration featuring these controls, see Demonstration: Input Controls in Action.

Objects created with the **ComboBox** control are similar to those created with the **ListBox** control. Like a list box, items can be added to, removed from, or sorted in a combo (combination) box while your program runs. However, there are differences. Combo boxes take up less space on the form, show a default setting, and in their default Style property setting are only one line high. But if the user clicks the combo box while your program runs, the combo box expands to reveal a full list of choices. (If the list is too long to be displayed in the combo box, scroll bars appear).

To view an illustration showing a combo box on a form, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Adding Combo Box Items

Typically, choices are added to a combo box with the **AddItem** method (a method normally used with list-oriented objects). You use the **AddItem** method by writing code in the Form_Load event procedure.

To view sample code showing an example of the Form_Load event procedure, click this icon.
{ewc mvimg, mvimage,!code.bmp}

To view a demonstration featuring these controls, see Demonstration: Input Controls in Action.

This demonstration shows input controls working in a program that simulates an electronic ordering system. Notice how the option button, check box, list box, and combo box elements process input differently and how they are manipulated with program code.

To view the demonstration, click this icon.
{ewc mvimg, mvimage, !democlip.bmp}

One of the most exciting features of Visual Basic is its ability to work easily with other applications on your system.

This section includes the following topics:

® OLE Controls

® Data Controls

Note   You will learn more about OLE and Data controls in Chapter 10: Working with Text Files and Databases and Chapter 11: Connecting to Microsoft Office.

With the **OLE** control (also called the **OLE Container** control), you can add objects that start Windows-based applications from your Visual Basic programs. You can also add application objects that have been registered in your system through the Windows system registry, a special configuration file. (When applications and other tools are installed on your computer, they're listed in the system registry.)

Using OLE Controls

When you place an **OLE** control on your form, the **Insert Object** dialog box appears and lists the application objects that you can use in your program.

To view an illustration showing what happens when you create an **OLE** object, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Create New and Create From File

After you select an application object, you can control how it starts by selecting the **Create New** or **Create From File** option button. **Create New** starts the selected application with a new document in your program. **Create From File** loads an existing document into the application when Visual Basic starts the application.

Display As Icon

Display As Icon is another useful option. If you select this check box option, Visual Basic creates an application icon on your form. This icon starts the selected application when the user double-clicks it on the form. If **Display As Icon** is left blank, the application opens in a window on the form.

Customizing Application Objects

When you click **OK** in the **Insert Object** dialog box, the application object you selected starts. You can customize the document that users will see when they launch the Windows-based application in your Visual Basic program. When you are finished customizing the document, click the **Close and Exit** command on the application's **File** menu. Close the application itself, then resize the **OLE** object as necessary.

Using the OLE Control

To view a demonstration that shows you how to use the **OLE** control to create an application object that starts the Paint utility from a Visual Basic program, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

With the **Data** control, you can manipulate database fields and records directly on your form and create a custom database viewer (a front-end or client application). Many corporate network users will find this feature useful because with it, they can extract just the information they want from a database and combine it with other useful information. With the **Data** control, you can display just the information you want.

What is a Database?

A database is an organized collection of information stored in a file. Computer databases are created with special programs, such as Microsoft Access, dBASE, or Paradox, which have the necessary tools to organize, manipulate, and locate structured information. If you have a database in your organization that you would like to customize or extract data from, the **Data** control is the tool for you.

Using the Data Control

Before you use the **Data** control, make sure that you have an existing database in one of the database formats that Visual Basic supports. Think about the types of information you want to display, because you can use additional toolbox controls (bound controls) to hold the output. Also, make sure that you are familiar with the structure of your database tables, fields, and records.

To view an illustration of a sample database structure, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

When you’re ready to display information from a database, follow these steps to create a data control:

u To create data and bound display objects

1. In the Visual Basic toolbox, click the **Data** control and create a rectangular data object on your form.

   The data object looks like a navigation control, so be sure to leave enough room for clean-looking arrow buttons and a database file name.
2. To create display windows appropriate for your data, use one or more toolbox controls. For example, create a text box to display text or a picture box to display artwork.

u To set properties

1. On the Visual Basic toolbar, click **Properties** to display the Properties window.
2. On your form, click **Data1**.
3. In the Properties window, set the **Connect** property to the database format that your program application uses. (Access is the default.)
4. Set the **DatabaseName** property to the name of the database you plan to use. (If necessary, browse for the file on your system.)
5. Set the **RecordSource** property to the name of the database table you want to use in your database. (If you specified the database name correctly in Step 4, you’ll see a list of table options when you click the **RecordSource** property.)
6. Set the **Caption** property to the name of your database. (This name will appear inside your data object when the program runs.)

u To bind display objects to a database

1. On the form, click the text box, picture box, or other bound object that will display the data.
2. Set the object **DataSource** property to **Data1** (the source of the data).
3. Set the object **DataField** property to the name of the field you want to display in the database.
4. For each bound object on your form, repeat Steps 2 and 3.
5. If you want to tell the user about the database information that the bound objects display, create one or more labels on your form.

To view an illustration of a form that contains a data object and two bound text box objects, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

You can extend Visual Basic's functionality by installing either the ActiveX controls that come with Visual Basic or ActiveX controls created by third-party tool vendors. (ActiveX controls are .ocx files, formerly known as custom controls.)

Adding and Deleting ActiveX Controls

For each project, you add ActiveX controls to the Visual Basic toolbox by using the **Project** menu **Components** command. To add a new control to the toolbox, successively click the **Project** menu, **Components**, the **Controls** tab, and the control you want to add. To remove a control, simply remove the check mark next to the control name.

Note that Visual Basic maintains a separate toolbox for each project you create, so you can mix and match controls as your programming needs dictate. Removing unused ActiveX controls from the toolbox helps you create programs that require less memory and disk space. (You cannot remove any of the 20 standard controls from the toolbox.)

To view an illustration showing the **Controls** tab with the Microsoft **CommonDialog** control selected, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

For more information about the **CommonDialog** control, see Using the CommonDialog Control.

1. In Visual Basic terminology, what is a *method*?

{ewc mvimg, mvimage,!answer.bmp} A. A special container used to hold data temporarily in a program.

{ewc mvimg, mvimage,!answer.bmp} B. A tool you use to create objects on a Visual Basic form.

{ewc mvimg, mvimage,!answer.bmp} C. A value or characteristic held by a Visual Basic object, such as **Caption** or **ForeColor**.

{ewc D. A special keyword in program code that performs an action or a service for a particular object.

mvimg, mvimage,!answer.bmp}

2. A Visual Basic programmer includes the abbreviation cbo in an object name. What toolbox control was probably used to create the object?

{ewc mvimg, mvimage,!answer.bmp} A. **CheckBox**.

{ewc mvimg, mvimage,!answer.bmp} B. **ComboBox**.

{ewc mvi C. **CommandButton**.

mg,mvimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. **CommonDialog**.

3. Which toolbox control would you use to set the current folder on a disk drive?

{ewc mvimg,mvimage,!answer.bmp} A. **DriveListBox**.

{ewc mvimg,m B. **DirListBox**.

vimage,!answer.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. **FileListBox**.

{ewc mvimg, mvimage,!answer.bmp} D. **OLE**.

4. Which toolbox control would you use to display a list of input choices for a user if you wanted to

allow them to pick more than one option?

{ewc mvimg, mvimage,!answer.bmp} A. **TextBox**.

{ewc mvimg, mvimage,!answer.bmp} B. **Label**.

{ewc mvimg, mvimage,!answer.bmp} C. **OptionButton**.

{ewc m D. **CheckBox**.

vimg,mvimage,!answer.bmp}

5. What type of object is listed in the OLE control Insert Object dialog box?

{ewc mvimg,mvimage,!answer.bmp} A. ActiveX controls that can be inserted into the toolbox.

{ewc mvimg,mvimage,!answer.bmp} B. Macros that can be run in Microsoft Office applications.

{ewc mvimg, C. Existing documents from the Microsoft Word recently-used file list.

mvimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. Application objects and special system components recorded in the Windows system registry.

6. What is a bound control?

{ewc mvimg,mvimage,!answer.bmp} A. A toolbox control that has been linked to a data object with the **DataSource** property.

{ewc mvimg,mvim B. An ActiveX control that displays several standard dialog box types.

age,!answer.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. The name of the **Data** control when its **Connect** property has been set to a valid database format.

{ewc mvimg, mvimage,!answer.bmp} D. A toolbox control that has been aligned to the left edge of a form.

7. How do you add ActiveX controls to the Visual Basic toolbox?

{ewc A. Use the **File** menu **New Project** command.

mvimg,mvimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} B. Use the **Project** menu **Components** command.

{ewc mvimg,mvimage,!answer.bmp} C. Use the **OLE** control **Insert Object** dialog box.

{ewc mvimg,m D. Install new software that modifies the Windows system registry.

vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

The program you create in this lab will be a simple tool that displays the Name record from Shops.mdb, a Microsoft Access database. Your program should include these user interface elements:

® A data object.

® A text box.

® A descriptive label.

® A Quit button.

As you create this simple application, you'll review the information in this chapter related to using controls, setting properties, and opening databases.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **30 minutes**

Objectives

After completing this lab, you will be able to:

® Create a data object and a text box on a form.

® Set properties to link the data object to a database.

® Set properties to make the text box a bound display object.

® Browse a field of the Shops.mdb database data at run time.

Lab Setup

For this program to run correctly, you need to locate the Shops.mdb database. If you installed the *Learn Microsoft Visual Basic 6.0 Now* CD-ROM with the default settings, a clean copy will be located in the \LVB6\Ch02 folder on your hard drive. You'll need to specify this file location when you set the **DatabaseName** property in Step 4 of Exercise 2.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Creating the User Interface

In this exercise, you create the Data Object program user interface.

® Exercise 2: Setting Data Properties

In this exercise, you use the Properties window to change several important data object properties.

® Exercise 3: Setting Bound Object Properties

In this exercise, you use the Properties window to set a database-bound text box's properties.

® Exercise 4: Browsing the Database

In this exercise, you run the Data Object program to verify that names in the Shops.mdb database are displayed correctly.

In this exercise, you create the user interface for the Data Object program. This interface consists of the following objects on a blank form:

® A data object

® A text box

® A label

® A command button

u To create a new Visual Basic project

1. From the Windows taskbar, click **Start**.
2. Point to **Programs**, and then point to **Microsoft Visual Basic 6.0**.
3. Click the **Visual Basic 6.0** program icon.
4. In the Visual Basic **New Project** dialog box, click **OK**.

u To create the Data1 object

1. In the toolbox, click the **Data** control.
2. Create a small rectangular data object in the middle of the form.

u To create the text box

1. In the toolbox, click the **TextBox** control.
2. Create a rectangular text box.
3. Position the text box above the data object.

u To create the label

1. Click the **Label** control.
2. Create a short label and position it to the left of the text box.
3. Set the **Caption** property of the label to **Name**.

u To create the command button

1. Click the **CommandButton** control.
2. Create a command button and position it beneath the data object.
3. Change the **Caption** property of the command button to **Quit**.

u To write the event procedure

1. To open the Code window, double-click the command button.
2. In the Command1_Click event procedure, type **End**.
3. Close the Code window.
4. From the **View** menu, click **View Object**.

u To save the project

1. On the **File** menu, click **Save Project As**.
2. Save your form and project to disk under the name **DataObj**.

   Visual Basic will prompt you for two file names — one for your form file (DataObj.frm) and one for your project file (DataObj.vbp).

In this exercise, you use the Properties window to change several important data object properties.

u **To set the Data1 object's properties**

1. On the form, click **Data1**, and then double-click the Properties window title bar to restore it to full size.
2. If you can't see the Properties window, click **Properties window** on the toolbar to make the window visible.
3. Click the **Connect** property and verify that it is set to Access (the default).
4. Set the **DatabaseName** property to *<drive>***\LVB6\Ch02\Shops.mdb**.

   Shops.mdb is a simple customer database created in Microsoft Access. Later in this exercise, you'll browse shops.mdb. For more information about the location of the Shops.mdb database, see Lab Setup.
5. Set the **RecordSource** property to **Customers**.
6. Set the **Caption** property to **Shops.mdb**.

In this exercise, you use the Properties window to set a database-bound text box’s properties.

u **To set the text box object's properties**

1. On the form, click the text box.
2. Set the **DataSource** property to **Data1** (the source of the data).
3. Set the **DataField** property to **OrganizationName** (the field you want to display).
4. On the toolbar, click **Save Project** to save your program to disk.

In this exercise, you run the Data Object program to verify that names in the Shops.mdb database are displayed correctly.

u To examine database field entries

1. On the Visual Basic toolbar, click **Start** to run Data Object.

   Visual Basic loads the Shops.mdb database and places the first OrganizationName field in the text box.
2. Now, examine other field entries by clicking the data object's buttons. In Data1, click the inner-right button.

   The second name in the database appears in the text window.
3. Click the outer-right button.

   Visual Basic displays the last name in the database.
4. Click the outer-left button.

   Visual Basic displays the first name in the database again.

u To modify database names (optional)

If you like, experiment with Data1 by modifying names in the database.

1. To replace a name, highlight it in the text box, press DELETE, and then type a new name.

   When you move to a new record by clicking one of Data1's buttons, your change is saved in the database. You can suppress this powerful capability by changing the Data1 **ReadOnly** property to **True**.
2. When you are satisfied that the program runs correctly, click **Quit** to exit the program.

Like any specialized activity, Visual Basic programming comes with its own evolving set of technical terms. Many of these terms have their roots in object-oriented programming, a methodology closely associated with C++ programming. As you work through the chapters in this course, you'll want to be familiar with these essential Visual Basic concepts.

Control

A control is a tool you use to create objects on a Visual Basic form. You select controls from the toolbox and use the mouse to draw objects on a form. You use most controls to create user interface elements, such as command buttons, image boxes, and list boxes.

Object

An object is a type of user interface element you create on a Visual Basic form by using a toolbox control. (In fact, in Visual Basic, the form itself is also an object.) You can move, resize, and customize objects by setting object properties. Objects also have what is known as inherent functionality — they know how to operate and can respond to certain situations on their own. (A list box "knows" how to scroll, for example.) You can customize Visual Basic objects by using event procedures that are fine-tuned for different conditions in a program.

Property

A property is a value or characteristic held by a Visual Basic object, such as **Caption** or **ForeColor**. Properties can be set at design time by using the Properties window or at run time by using statements in the program code. In code, the format for setting a property is:

```
Object.Property = Value
```

where

Object is the name of the object you're customizing.

Property is the characteristic you want to change.

Value is the new property setting.

For example,

```
Command1.Caption = "Hello"
```

could be used in the program code to set the **Caption** property of the **Command1** object to "Hello".

Event Procedure

An event procedure is a block of code that runs when a program object is manipulated. For example, clicking the first command button in a program executes the Command1_Click event procedure. Event procedures typically evaluate and set properties and use other program statements to perform the work of the program.

Program Statement

A program statement is a combination of keywords, identifiers, and arguments in the code that does the work of the program. Visual Basic program statements create storage space for data, open files, perform calculations, and do several other important tasks.

Method

A method is a special keyword that performs an action or a service for a particular program object. In code, the format for using a method is

```
Object.Method Value
```

where

Object is the name of the object you are working with.

Method is the action you want the object to perform.

Value is an optional argument to be used by the method.

For example, this statement uses the **AddItem** method to put the word *Check* in the **List1** list box:

```
List1.AddItem "Check"
```

Variable

A variable or identifier is a special container that holds data temporarily in a program. You create variables to store calculation results, create file names, process input, and so on. Variables can store numbers, names, property values, and references to objects.

You may be a Visual Basic beginner now, but that won’t be true for long. As your programs increase in size and sophistication, the number of objects you use on your forms will multiply quickly. There is an easy way to avoid mistaking one object for another in the Properties window or in your program code: assigning a unique name to each object soon after you create it. It’s simple — when you set your other object properties, just click the (**Name**) property, and then give the object a unique name.

This section includes the following topics:

® Visual Basic Object Names

® Object Naming Conventions

Programmers typically create names for their objects that clearly identify the purpose of the object and the toolbox control that created it. For example, you might give the name **lblInstructions** to a label that displays user operating instructions. (In this case, lbl stands for the **Label** control, and Instructions describes the label's purpose).

{ewc mvimg, mvimage,!tip.bmp}

This table lists the naming conventions for objects created by the 20 standard Visual Basic toolbox controls. (**Form** and **menu** objects, which you use often but which are not in the toolbox, are given the prefixes *frm* and *mnu* respectively.) Whenever you use more than five or six objects on a form, use this table as a naming guide:

Object	**Prefix**	**Example**
combo box	cbo	cboEnglish
check box	chk	chkReadOnly
command button	cmd	Open
data	dat	datBiblio
directory list box	dir	dirSource
drive list box	drv	drvTarget
file list box	fil	filSource
frame	fra	fraLanguage
horizontal scroll bar	hsb	hsbVolume
image	img	imgEmptyBarrel
label	lbl	lblInstructions
line	lin	linUnderline
list box	lst	lstPeripherals
OLE	ole	oleObject1
option button	opt	optFrench
picture box	pic	picSmokeCloud
shape	shp	shpWireScreen
text box	txt	txtGetName
timer	tmr	tmrRunAnimation
vertical scroll bar	vsb	vsbTemperature

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter focuses on processing input from another source in the user interface: menu commands and dialog boxes. In this chapter, you will learn how to:

® Add menus to your programs by using the Menu Editor.

® Process menu choices by using program code.

® Use the **CommonDialog** ActiveX control to display standard dialog boxes.

Menus, which are located on the menu bar of a form, contain a list of related commands. When you click a menu title in a Windows-based program, a list of menu commands should always appear in a well-organized list.

Most menu commands run immediately after they are clicked. For example, when the user clicks the **Edit** menu **Copy** command, Windows immediately copies information to the Clipboard. However, if ellipsis points (…) follow the menu command, Visual Basic displays a dialog box that requests more information before the command is carried out.

This section includes the following topics:

The Menu Editor is a Visual Basic dialog box that manages menus in your programs. With the Menu Editor, you can:

® Add new menus

® Modify and reorder existing menus

® Delete old menus

® Add special effects to your menus, such as access keys, check marks, and keyboard shortcuts.

To view an illustration of the Menu Editor, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

Creating Menu Command Lists

To build lists of menu commands, you first need to create the menus and then add them to the program menu bar.

u To create a list of menu commands on a form

1. Click the form itself (not an object on the form).
2. On the Visual Basic toolbar, click the Menu Editor icon, or select **Menu Editor** from the **Tools** menu.
3. In the **Caption** text box, type the menu caption (the name that will appear on the menu bar), and then press TAB.
4. In the **Name** text box, type the menu name (the name the menu has in the program code).

   By convention, programmers use the *mnu* object name prefix to identify both menus and menu commands.
5. To add the menu to your program menu bar, click **Next**.

   The Menu Editor clears the dialog box for the next menu item. As you build your menus, the structure of the menus and commands appear at the bottom of the dialog box.
6. In the **Caption** text box, type the caption of your first menu command.
7. Press tab, and then type the object name of the command in the **Name** text box.
8. With this first command highlighted in the menu list box, click the right arrow button in the Menu Editor.

   In the **Menu** list box, the command moves one indent (four spaces) to the right. Click the right arrow button in the Menu Editor dialog box to move items to the right, and click the left arrow button to move items to the left.
9. Click **Next**, and then continue to add commands to your menu.

   The position of list box items determines what they are:

List box item	Position
Menu title	Flush left
Menu command	One indent
Submenu title	Two indents
Submenu command	Three indents

u To add more menus

1. When you're ready to add another menu, click the left arrow button to make the menu flush left in the **Menu** list box.
2. To add another menu and menu commands, repeat Steps 3 through 9 in the preceding procedure.
3. When you're finished entering menus and commands, click **OK** to close the Menu Editor. (Don't accidentally click **Cancel** or all your menu work will be lost.)

   The Menu Editor closes, and your form appears in the programming environment with the menus you created.

Adding Event Procedures

After you add menus to your form, you can use event procedures to process the menu commands. Clicking a menu command on the form in the programming environment displays the event procedure that runs when the menu command is chosen. You'll learn how to create event procedures that process menu selections in Processing Menu Choices.

To view a demonstration showing you how to create menus, see Demonstration: Creating Menus.

Visual Basic makes it easy to provide access key and shortcut key support for menus and menu commands.

Access and Shortcut Keys

The access key for a command is the letter the user can press to execute the command when the menu is open. The shortcut key is the key combination the user can press to run the command without opening the menu. Here's a quick look at how to add access and shortcut keys to existing menu items:

To	Do this
Add an access key to a menu item	Start the Menu Editor.
	Prefix the access key letter in the menu item caption with an ampersand (&).
Add a shortcut key to a menu command	Start the Menu Editor.
	Highlight the command in the menu list box.
	Pick a key combination from the Shortcut drop-down list box.

Creating Access and Shortcut Keys

You can create access keys and shortcut keys either when you first create your menu commands or at a later time.

The following illustration shows the menu commands associated with two menus, **File** and **Clock**. Each menu item has an access key ampersand character, and the **Time** and **Date** commands are assigned shortcut keys. To view the illustration, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

To view a demonstration showing you how to create menus, see Demonstration: Creating Menus.

After you place menu items on the menu bar, they become objects in the program. To make the menu objects do meaningful work, you need to write event procedures for them. Typically, menu event procedures:

® Contain program statements that display or process information on a form.

® Modify one or more object properties.

For example, the event procedure for a command named **Time** might use the **Time** keyword to display the current system time in a text box.

Processing the selected command might require additional information (you might need to open a file on disk, for example). If so, you can display a dialog box to receive user input by using a common dialog box. You'll learn this technique in the next section.

{ewc mvimg, mvimage,!tip.bmp}

To view a demonstration showing you how to create menus, see Demonstration: Creating Menus.

A dialog box is simply a form in a program that contains input controls designed to receive information. To make your programming faster, Visual Basic includes an ActiveX control, named **CommonDialog**.

With this control, you can easily display six standard dialog boxes in your programs. These dialog boxes handle routine tasks such as opening files, saving files, and picking fonts. If the dialog box you want to use is not included in this ready-made collection of objects, you can create a new one by adding a second form to your program. (For more information about adding forms, see Chapter 7: Working with Forms, Printers, and Error Handlers.)

This section includes the following topics:

Before you can use the **CommonDialog** control, you need to verify that it is in your toolbox. If you don't see the CommonDialog icon, follow this procedure to add it to the toolbox.

u To add the CommonDialog control to the toolbox

1. From the **Project** menu, click **Components**.
2. Click the **Controls** tab.
3. Ensure that the **Selected Items Only** box is not checked.
4. Place a check mark next to **Microsoft Common Dialog Control**, and then click **OK**.

Creating a Dialog Box

Follow this procedure to create a dialog box with the **CommonDialog** control.

u To create a common dialog object on your form

1. In the toolbox, double-click the **CommonDialog** control.
2. When the common dialog object appears on your form, drag it to an out-of-the-way location.

Note You cannot resize a common dialog object, and it disappears when your program runs. The common dialog object itself displays nothing — its only purpose is to display a standard dialog box on the screen when you use a method in program code to request it.

To view an illustration of a common dialog object on a form, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

This table lists the name and purpose of the six standard dialog boxes that the common dialog object provides and the methods you use to display them:

Dialog Box	Purpose	Method
Open	Gets the drive, folder name, and file name for an existing file that is being opened.	**ShowOpen**
Save As	Gets the drive, folder name, and file name for a file that is being saved.	**ShowSave**
Print	Provides user-defined printing options.	**ShowPrinter**
Font	Provides user-defined font type and style options.	**ShowFont**
Help	Provides online user information.	**ShowHelp**
Color	Provides user-defined color selection from a palette.	**ShowColor**

To display a standard dialog box in a program, you need to call the common dialog object. You do this by using the appropriate object method in an event procedure. If necessary, you also use program code to set one or more common dialog object properties before the call. (For example, if you are using the **Open** dialog box, you might want to control what type of files are displayed in the list box.) Finally, your event procedure needs to process the choices made by the user when they complete the standard dialog box.

This section presents two simple event procedures, one that manages an **Open** dialog box and one that uses information received from a **Color** dialog box.

The following topics are included in this section:

® Creating an Open Dialog Box

® Creating a Color Dialog Box

The following code window shows an event procedure named mnuOpenItem_Click. You can use this event procedure to display an **Open** dialog box when the user clicks the **Open** command on the **File** menu. The event procedure assumes that you have already created a **File** menu containing **Open** and **Close** commands and that you want to open Windows metafiles (.wmf). To view the code window, click this icon:
{ewc mvimg, mvimage,!code.bmp}

The event procedure uses these properties and methods:

Object	Property/Method	Purpose
Common Dialog	**ShowOpen**	Displays the dialog box.
	Filter	Defines the file type in the dialog box.
Menu	**Enables**	Enables the **Close** menu command, which users can use to unload the picture.
Image	**Picture**	Opens the selected file.

To view an illustration showing an **Open** dialog box created by the preceding event procedure, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

Note You'll use this event procedure as the foundation of your solution to the Review Lab 3 exercises. To see the event procedure in action, view the demonstration in Review Lab 3.

If you need to update the color of a user interface element while your program runs, you can prompt the user to pick a new color with the **Color** dialog box displayed by using the **Common Dialog** object. The color selections provided by the **Color** dialog box are controlled by the **Flags** property, and the **Color** dialog box is displayed with the **ShowColor** method.

This code window shows an event procedure that you can use to change the color of a label while your program runs. The value used for the **Flags** property — which in this case prompts Visual Basic to display a standard palette of color selections — is a hexadecimal (base 16) number. (To see a list of other potential values for the **Flags** property, search for *CommonDialog constants* in the Visual Basic online Help.) The event procedure assumes that you have already created a menu command named **TextColor** with the Menu Editor. To view the event procedure code window, click this icon:
{ewc mvimg, mvimage,!code.bmp}

Demonstration: Using the Color Dialog Box

The following demonstration shows how to use the **Color** dialog box to customize the time and date program created earlier in this chapter. As you run the demonstration, pay close attention to:

® How the common dialog box is located.

® Where the program code that displays the **Color** dialog box is placed.

To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

1. What is the purpose of the Name text box in the Menu Editor?

{ewc mvimg, mvimage,!answer.bmp} A. Assigning an object name to a menu or command.

{ewc mvimg, mvimage,!answer.bmp} B. Creating a menu or command caption on the menu bar.

{ewc mvimg, mvimage,!answer.bmp} C. Assigning a shortcut key to a menu command.

{ewc D. Assigning keyboard access keys.

m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

2. Access keys perform what function?

{ew
c
mvi
mg,
mvi
ma
ge,!
ans
wer
.bm
p}

A. Run a menu command when you press a function key.

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

B. Provide access to database information in a program.

{e
w
c
m
vi
m

C. Run commands in an open menu when you press the underlined letter in the command name.

g,mvimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. Close menus.

3. **Which of the following is not a standard dialog box that you can display with the CommonDialog control?**

{ewc mvimg,mvimage,!answer.bmp} A. **Open**.

{ewc mvimg,m B. **Print**.

vimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} C. **Font**.

{ewc mvimg,mvimage,!answer.bmp} D. **Find**.

4. What property do you set with program code if you want to disable a menu command?

{ewc mvimg, mvimage,!answer.bmp}	A.	**Enabled**.
{ewc mvimg, mvimage,!answer.bmp}	B.	**Disabled**.
{ewc mvimg, mvimage,!answer.bmp}	C.	**ShowOpen**.
{ewc mvim	D.	**False**.

g,mvimage,!answer.bmp}

5. **What is the purpose of the following program statement?**

```
CommonDialog1.Filter = "Metafiles (*.WMF)|*.WMF"
```

{ewc mvimg,mvimage,!answer.bmp}

A. Displaying a Windows metafile in an image box object.

{ewc mvimg,mvimage,!answsw

B. Creating a common dialog object named **CommonDialog1**.

er.bmp}

{ewc mvimg, mvimage, !answer.bmp} C. Using the **Filter** property to set the default file types displayed in a dialog box created by a common dialog object.

{ewc mvimg, mvimage, !answer.bmp} D. Filtering all Windows metafiles in the dialog box with the .wmf conversion routine.

The program you create in this lab will be a menu-driven metafile browser capable of locating and displaying any Windows metafile on your system. (Metafiles are resizable graphic images that take up very little disk space and have a .wmf file name extension.) Your program should include these features:

® A **File** menu with **Open**, **Close**, and **Exit** commands.

® A common dialog object for opening the metafile.

® An image object for displaying the picture.

As you create this basic application, you will review the information in this chapter related to creating menus, opening standard dialog boxes, and using program code.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **40 minutes**

Objectives

After completing this lab, you will be able to:

® Create a menu on a form.

® Create access keys and shortcut keys for menu commands.

® Use a common dialog object to display an **Open** dialog box.

® Use program code to control **Open**, **Close**, and **Exit** menu commands.

Lab Setup

The metafile browser program displays metafiles, so when you run the program you need to have a few metafiles to look at. Look for a few basic images in the \LVB6\Ch03 folder on your hard drive.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Creating a Menu

In this exercise, you use the Menu Editor to create a **File** menu with **Open**, **Close**, and **Exit** commands for your program. You also assign access keys and shortcut keys to the commands, so you can run them from the keyboard.

® Exercise 2: Using the CommonDialog Control

In this exercise, you create common dialog and image objects on your form.

® Exercise 3: Writing Code to Process Menu Commands

In this exercise, you write event procedures for the **Open**, **Close**, and **Exit** menu commands in your program.

® Exercise 4: Test the Program

In this exercise, you run the metafile browser program to test the menus, access keys, and shortcut keys.

In this exercise, you use the Menu Editor to create a **File** menu with **Open**, **Close**, and **Exit** commands for your program. You also assign access keys and shortcut keys to the commands, so you can run them from the keyboard.

u To create a File menu

1. Start Visual Basic and open a new, standard Visual Basic application.
2. On the toolbar, click **Menu Editor** to open the Menu Editor dialog box.
3. In the **Caption** text box, type **&File**.
4. In the **Name** text box, type **mnuFile**, and then click **Next**.

   By placing the & character before the letter F, you specify F as the menu access key.

u To assign access and shortcut keys

1. In the **Caption** text box, type **&Open…**.
2. In the **Name** text box, type **mnuOpenItem**.
3. To indent the selected (highlighted) command, click the right arrow button.
4. In the Shortcut drop-down list, click CTRL+O for a shortcut key, and then click **Next**.
5. In the **Caption** text box, type **&Close**.
6. In the **Name** text box, type **mnuCloseItem**.
7. In the Shortcut drop-down list, click CTRL+C for a shortcut key, and then click **Next**.
8. In the **Caption** text box, type **E&xit**.
9. In the **Name** text box, type **mnuExitItem**.
10. In the Shortcut drop-down list, click CTRL+X for a shortcut key, and then click **OK**.

u To save your project

1. From the **File** menu, click **Save Project As**.
2. Save your form and project to disk under the name Metafile. Visual Basic will prompt you for two file names — one for your form file (Metafile.frm), and one for your project file (Metafile.vbp).

In this exercise, you create common dialog and image objects on your form.

u To create a common dialog object

1. Verify that the **CommonDialog** control is in your project toolbox. If it isn't, add it now by using the **Project** menu **Components** command.
2. To add a common dialog object to your form, double-click the **CommonDialog** control in the toolbox, and then drag the object to the lower right-hand side of the form.

u To create the image object

1. Click the **Image** control and create a large image object in the middle of your form.

   When you run your program, the image object displays bitmaps, metafiles, and other graphic images on a form.
2. On the form, click **Image1**. To restore the Properties window to full size, double-click the Properties window title bar.

   If you cannot see the Properties window, click **Properties** on the toolbar to display it.
3. Click the **Stretch** property and set it to **True**.

   When you run your program, **Stretch** makes the metafile fill the entire image object.
4. On the toolbar, click **Save Project** to save these changes to your program.

In this exercise, you write event procedures for the **Open**, **Close**, and **Exit** menu commands in your program.

u **To write event procedures**

1. In the Project window, click **View Code** , click the Code window **Object** drop-down list box, and then click **mnuOpenItem**.
2. In the mnuOpenItem_Click event procedure, type the following code:
   ```
   CommonDialog1.Filter = "Metafiles (*.WMF)|*.WMF"
   CommonDialog1.ShowOpen
   Image1.Picture = LoadPicture(CommonDialog1.Filename)
   mnuCloseItem.Enabled = True
   ```
3. In the **Object** drop-down list box, click **mnuCloseItem**, and then type the following code:
   ```
   Image1.Picture = LoadPicture("")
   mnuCloseItem.Enabled = False
   ```
4. In the **Object** drop-down list box, click **mnuExitItem**, and then type **End** in the event procedure.
5. On the toolbar, click **Save Project** to save your changes.

In this exercise, you run the metafile browser program to test the menus, access keys, and shortcut keys.

u To run the program

1. On the Visual Basic toolbar, click **Start**.

   Visual Basic loads the program and the form with its **File** menu.
2. From the **File** menu, click **Open**.
3. When the **Open** dialog box appears, load a metafile from the \LVB6\Ch03 folder.

   The metafile should appear correctly sized in your image object.
4. From the **File** menu, click **Close**.

   Your program should clear the metafile and turn off the **Close** command.
5. Try using the access keys and the shortcut keys to run the **File** menu commands. When you're finished, click the **File** menu **Exit** command.

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This is the first of three chapters that focus specifically on writing program code to manage the events in a Visual Basic application. This chapter describes Visual Basic variables and operators and tells you how to use specific data types to improve program efficiency. In this chapter, you will learn how to:

® Use variables to store information in your program.

® Work with specific data types to streamline your calculations.

A variable is a temporary data storage location in your program. Your code can use one or many variables, which can contain words, numbers, dates, properties, or object references. Variables are useful because they let you assign a short, easy-to-remember name to a piece of data you plan to work with. Variables can hold:

® User information entered at run time.

® The result of a specific calculation.

® A piece of data you want to display on your form.

In short, variables are simple tools you can use to track almost any type of information whose value can change.

This section includes the following topics:

Declaring a variable is the process of creating a variable in your Visual Basic program code. You can do this explicitly with the **Dim** keyword or implicitly by typing a new variable name in a program statement.

To view an expert point-of-view that describes some of the behind-the-scenes details of creating program variables, click this icon:
{ewc mvimg, mvimage,!exppov.bmp}

This section includes the following topics:

® Explicit Declarations

® Implicit Declarations

In Visual Basic, one of the ways to create a variable is to declare it explicitly. Typically, you declare a variable explicitly before you use the variable (usually at the beginning of an event procedure). You declare the variable by typing a **Dim** (dimension) statement and the variable name. For example, the following statement creates space for a program variable named LastName:

```
Dim LastName
```

Typing the statement reserves room for the variable in memory when the program runs, and lets Visual Basic know what type of data it should expect to see later.

Specifying the Variable Data Type

If you like, you can specify the variable data type after you type the variable name. (You'll learn about several fundamental data types in Working with Specific Data Types.) With Visual Basic, you can identify the data type in advance, so you can control how much memory your program uses. For example, if the variable holds an integer (a small number without any decimal places), you can declare the variable as an integer and save some memory.

Default Data Type

By default, however, Visual Basic reserves space for a variable data type called a **Variant** (a variable that can hold data of any size or format). This general-purpose data type is extremely flexible — in fact, it might be the only variable you use in your programs.

Assignment Operator

After you declare a variable, you are free to assign information to it in your code by using the assignment operator (=). This statement assigns the name "Jefferson" to the LastName variable:

```
LastName = "Jefferson"
```

After this assignment, you can substitute the LastName variable for the name "Jefferson" in your code. For example, this assignment statement would display *Jefferson* in the first label (Label1) on your form:

```
Label1.Caption = LastName
```

You can also declare a variable without the **Dim** statement; this process is called implicit declaration. To declare a variable implicitly, you simply use the variable on its own and skip the **Dim** statement altogether. Here's an example:

```
LastName = "Charles V"
```

Advantages and Disadvantages

In this course, you'll see variables declared with both explicit and implicit techniques. Implicit declaration has the advantage of speed because you don't spend time typing the **Dim** statement. However "the management" often discourages it for several reasons. First, implicit declaration doesn't force you to organize and list your variables in advance. Also, creating variables in this way prevents Visual Basic from displaying an error message if you mistype the variable name later.

{ewc mvimg, mvimage,!tip.bmp}

Changing Variable Values

Variables can maintain the same value throughout a program; or, more likely, they can change values several times, depending on your needs.

To view sample code that shows a variant variable named LastName can contain both text and a number, and how the variable can be assigned to object properties, click this icon:
{ewc mvimg, mvimage,!code.bmp}

In this section, you learn how to use these three special Visual Basic keywords, called functions, in a program.

® The **MsgBox** function, which displays output in a popup dialog box.
® The **InputBox** function, which prompts the user to supply input in a dialog box.
® Mathematical functions, with which Visual Basic calculates new values.

This section includes the following topics:

MsgBox is a useful dialog box function that displays output. Like **InputBox**, **MsgBox** takes one or more arguments as input, and you can assign the value returned (the results of the function call) to a variable. This is the syntax for **MsgBox**:

```
ButtonClicked = MsgBox(Message, NumberOfButtons, Title)
```

where

® *Message* is the text to be displayed on the screen.

® *NumberOfButtons* is a button style number (0 through 5).

® *Title* is the text displayed in the message box title bar.

® *ButtonClicked* is assigned the result returned by the function, which indicates which button in the dialog box the user clicked.

Note If you want to display only a message with **MsgBox**, the assignment operator (=), the *ButtonClicked* variable, the *NumberOfButtons* argument, and the *Title* argument are optional items. For more information about these optional arguments (including the different buttons you can use in a message box), search for *MsgBox function* in the Visual Basic online Help.

To view an illustration showing a dialog box, which created by using **MsgBox** in a program, click this icon: {ewc mvimg, mvimage,!illust.bmp}

The **MsgBox** dialog box illustration used this program code:

```
Dim FullName
FullName = "Microsoft Visual Basic"
MsgBox FullName, , "Product"
```

Note that when you omit the *ButtonClicked* argument, you leave an empty space between the two commas.

To view a demonstration showing the **MsgBox** and **InputBox** Functions, see Demonstration: Using InputBox and MsgBox.

Now and then, you might want to do a little number crunching in your programs. You may need to convert a value to a different type, calculate a complex mathematical expression, or introduce randomness to your programs.

Visual Basic functions can help you work with numbers in your formulas. As with any function, mathematical functions appear in a program statement and return a value to the program. In the following table, the argument *n* represents the number, variable, or expression you want the function to evaluate.

Function	Purpose
Abs(*n*)	Returns the absolute value of *n*.
Atn(*n*)	Returns the arctangent, in radians, of *n*.
Cos(*n*)	Returns the cosine of the angle *n*. The angle *n* is expressed in radians.
Exp(*n*)	Returns the constant *e* raised to the power *n*.
Int(*n*)	Returns the integer portion of *n*.
Rnd(*n*)	Generates a random number greater than or equal to 0 and less than 1.
Sgn(*n*)	Returns -1 if *n* is less than zero, 0 if *n* is zero, and +1 if *n* is greater than zero.
Sin(*n*)	Returns the sine of the angle *n*. The angle *n* is expressed in radians.
Sqr(*n*)	Returns the square root of *n*.
Str(*n*)	Converts a number to a numeric string value.
Tan(*n*)	Returns the tangent of the angle *n*. The angle *n* is expressed in radians.
Val(*n*)	Converts a numeric string value to a number.

For example, the following formula uses the **Sqr** (square root) function to calculate the hypotenuse of a triangle by using the variables a and b:

```
Hypotenuse = Sqr(a ^ 2 + b ^ 2)
```

A Visual Basic function is a statement that performs meaningful work (such as prompting the user for information) and returns a value to the program. The value that a Visual Basic function returns can be assigned to a variable, a property, another statement, or another function. A Visual Basic function is different from a Function procedure that you write yourself. (See Function Procedures for more information.)

Visual Basic functions often include one or more arguments that define their activities. For example, the **InputBox** function uses the Prompt variable to display a dialog box with instructions for the user. When a function uses one or more arguments, they are separated by commas, and the whole group of arguments is enclosed in parentheses. The following statement shows a function call that has two arguments:

```
FullName = InputBox$(Prompt, Title)
```

The **Prompt** argument takes a string that displays a prompt for the user in the **Input** box, and the **Title** argument takes a string that displays a title in the input box title bar.

One excellent use for a variable is to hold user input information. Often, you can use an object such as a file list box or a text box to retrieve this information. At times, though, you'll want to deal directly with the user and save the input in a variable rather than in a property. One way to do this is to use the **InputBox** function to display a dialog box on the screen and then store in a variable the text that the user types.

InputBox syntax looks like this:

```
VariableName = InputBox$(Prompt)
```

where

VariableName is a variable used to hold the input.

Prompt is a prompt that appears in the dialog box.

The dialog box created with an **InputBox** function typically contains these features:

® A prompt for directing the user.

® A text box for receiving typed input.

® Two command buttons, **OK** and **Cancel**.

To view an illustration of an input box, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

Here is the program code that created the **InputBox** shown above:

```
Dim Prompt, FullName
Prompt = "Please enter your name"

FullName = InputBox$(Prompt)
Label1.Caption = FullName
```

Note Two variables (Prompt and FullName) are declared explicitly with the **Dim** statement, and the FullName variable receives input from the **InputBox** function.

To view a demonstration showing the **InputBox** and **MsgBox** Functions, see Demonstration: Using InputBox and MsgBox.

To view a demonstration that shows how you can use the **InputBox** and **MsgBox** functions together with variables to process input and output in a program, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

A program statement is a line of code in a Visual Basic program. Your program statements can contain any combination of Visual Basic keywords, identifiers, properties, functions, operators, and symbols that Visual Basic recognizes as a valid instruction. A complete program statement can be a simple keyword:

```
Beep
```

This program statement sounds a note from your computer’s speaker. Or, a statement can be a combination of elements, such as the statement showing in this illustration, which assigns the current system time to a label caption.

{ewc MVIMG, MVIMAGE,!B04G005.BMP}

Statement Syntax

The anatomy of a program statement is syntax, the rules of construction that you must use when you build a program statement. Visual Basic shares many of its syntax rules with earlier versions of the Basic programming language and with other language compilers.

The trick to writing good program statements is learning the syntax of the most useful language elements. Then, it's a matter of using those elements correctly to process the data in your program. Fortunately, Visual Basic does a lot of the toughest work for you. The time you spend writing program code will be relatively short, and the results can be used again in future programs.

In the following topics, you’ll learn the most important Visual Basic keywords and program statements. You’ll find that they complement the programming skills you’ve already learned rather nicely—and they can help you to write powerful programs in the future.

The **Variant** data type works well in most situations. There are times, though, when working with specific data types can improve your Visual Basic programs. Specifying variable size can help you increase program performance. Creating a user-defined data type can help you hold information that is formatted in an unusual format. And if the variable never changes its value, declaring it as a constant can help you increase program efficiency. In this section, you round out your experience with variables by exploring other aspects of data types.

This section includes the following topics:

In most cases, the **Variant** data type will be the only data type you need. When you use **Variant** variables, you can store all Visual Basic predefined data types and switch formats automatically. **Variants** are also easy to use, and you don't have to give much thought to the eventual size of the variable when you declare it.

If you want to create especially fast and concise code, however, you may want to use more specific data types. For example, a variable might always contain small integers (numbers without a decimal point). When this occurs, declaring the variable as an **Integer** rather than as a **Variant** can save memory when the program runs. An **Integer** variable will speed up arithmetic operations too, so you can gain a small speed advantage when you use it.

{ewc mvimg, mvimage,!tip.bmp}

Declaring Data Types

Using specific data types is really not that much different than using Variant variables. You can specify some fundamental data types by adding a type-declaration character to the variable name. (Several data type declarations appear in the table below.) For example, you can declare a variable as an Integer type by adding a % character to the end of its name. So, in Visual Basic, the following two declaration statements are equivalent:

```
Dim I As Integer
Dim I%
```

The following table lists the fundamental data types in Visual Basic. Use this table as a reference list if you decide to fine-tune the size and type of your variables.

Data Type	Size	Range
Byte	1 byte	0 through 255
Integer	2 bytes	–32,768 through 32,767
Long	4 bytes	–2,147,483,648 through 2,147,483,647
Single	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double	8 bytes	-1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency	8 bytes	–922337203685477.5808 through 922337203685477.5807
String (variable length)	10 bytes + 1 byte per character	0 to approximately 2 billion
String (fixed length)		1 through 65,400 characters
Boolean	2 bytes	True or False
Date	8 bytes	January 1, 100

		through December 31, 9999
Variant	16 bytes (with numbers); 22 bytes + 1 byte per character (with strings)	All data type ranges

The following program code shows how you might declare and use the fundamental data types in an event procedure. The program code uses a few unfamiliar statements (such as **Select Case**) which is introduced fully in the next chapter. For now, just notice how the variables are declared and assigned values in the program code. To view the program code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

Visual Basic also lets you create your own data types. This feature is most useful when you work with mixed data types that naturally fit together.

Creating User-Defined Data Types

To create a user-defined data type, you:

® Use the **Type** statement in the Declarations section of a form or standard module.

® Declare variables associated with the new data type (variables belonging to a type are not declared with the **Dim** statement.)

This sample code illustrates the **Type** statement. The declaration uses **Type** to create **Employee**, a user-defined data type. (**Employee** stores a worker's name, date of birth, and hire date.) To view the sample code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

Note If you want to place the **Type** statement in the Declarations section of a form module, you must precede the **Type** statement with the **Private** keyword.

Placing User-Defined Data in Program Code

After you create a new data type, you can use it in the program code. The following Dim statement uses the user-defined **Employee** data type. The first statement creates a **variable** named ProductManager and states that it's the **Employee** data type. The second statement assigns the name “Erick Cody” to the Name component of the variable:

```
Dim ProductManager As Employee
ProductManager.Name = "Erick Cody"
```

This looks a little like setting a property, doesn’t it? Visual Basic uses the same notation for the relationship between objects and properties as it uses for the relationship between user-defined data types and their component variables.

If a variable in your program contains a value that never changes, you should consider storing the value as a constant instead of as a variable. A constant is a meaningful name that takes the place of a number or text string that doesn't change (such as π, a fixed mathematical quantity.)

Advantages of using constants include:

® Making program code more readable

® Saving memory

® Making program-wide changes easier to accomplish.

How Constants Work

Constants operate a lot like variables, but you can't modify their values at run time. You declare (define) constants with the **Const** keyword, as shown in the following example:

```
Const Pi = 3.14159265
```

This statement creates a constant called Pi that you can use in place of the value of π in the program code. To create a constant that's available to procedures throughout a program, use the **Public** keyword to create the constant in a standard module. For example:

```
Public Const Pi = 3.14159265
```

Note To learn more about using standard modules, see Chapter 9: Working with Modules and Procedures or search for *Module* in the Visual Basic online Help.

Displaying Constants

The following program statement shows how you can display the value contained in the Pi constant with an object named Label1:

```
Label1.Caption = Pi
```

Constants are very useful in program code, especially when your program includes mathematical formulas, such as Area = $2\pi r^2$. The next section describes how you can use operators and variables to write mathematical formulas.

Visual Basic contains several language elements designed for use in formulas (statements that use a combination of numbers, variables, operators, and keywords to create a new value). These language elements are mathematical operators, the symbols used to tie together the parts of a formula. With a few exceptions, these mathematical symbols are the ones you use in everyday life — their operations are quite intuitive.

This section includes the following topics:

The operators for addition, subtraction, multiplication and division are familiar, and their use is straightforward — you can use them in any formula in which numbers or numeric variables appear. The following table shows the operators you can use for basic arithmetic:

Operator	**Mathematical operation**
+	Addition
-	Subtraction
*****	Multiplication
/	Division (floating-point)

For example, the following program statements use the addition and multiplication operators to calculate the total cost of a $250 bicycle including 8.1% sales tax:

```
Dim BicycleCost, TotalPrice
Const SalesTaxRate = 0.081
BicycleCost = 250
TotalPrice = BicycleCost * SalesTaxRate + BicycleCost
```

In addition to the four basic arithmetic operators, Visual Basic includes four advanced operators, which are useful in special-purpose mathematical formulas and text processing applications:

Operator	Mathematical operation
\	Integer (whole number) division
Mod	Remainder division
^	Exponentiation (raising to a power)
&	String concatenation (combination)

For example, the following program statement uses the **Time** and **Date** functions and the string concatenation operator (**&**) to build a sentence from four string values:

```
Label1.Caption = "The current time is " & Time & " on " & Date
```

When you execute the program statement, the **Label1** object displays output in the following format:

```
The current time is 5:14:16 PM on 12/5/97
```

Demonstration: Advanced Arithmetic Operators

This demonstration shows the four advanced arithmetic operators in action. The program is a simple Visual Basic utility that prompts the user for two variables and calculates the result by using an advanced operator. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

In the last few sections, you experimented with seven mathematical operators and one string operator. In Visual Basic, you can mix as many mathematical operators in a formula as you like—as long as an operator separates each numeric variable and expression. For example, this is an acceptable Visual Basic formula:

```
Total = 10 + 15 * 2 / 4 ^ 2
```

The formula processes several values and assigns the result to a variable named Total. But you're probably wondering how Visual Basic evaluates this expression. After all, there's no way to tell which mathematical operators Visual Basic uses first to solve the formula. In this example, the order of evaluation matters a great deal.

Visual Basic solves this dilemma by using a specific order of precedence for mathematical operations. When Visual Basic evaluates an expression that contains more than one operator, the order of precedence supplies rules that determine which operators Visual Basic evaluates first.

The rules can be summarized by these principles:

® Use the operators in the order of precedence.

® For operators on the same precedence level, evaluate from left to right as they appear in an expression.

The following table lists mathematical operators in their order of precedence:

Order of precedence	**Symbols**	**Operator**
First	( )	The values between parentheses
2	^	Exponentiation (raising a number to a power)
3	-	Negation (creating a negative number)
4	* /	Multiplication and division
5	\	Integer division
6	Mod	Remainder division
7	+ -	Addition and subtraction
Last	&	String concatenation

Here's the formula you saw earlier:

```
Total = 10 + 15 * 2 / 4 ^ 2
```

Given the order of precedence, the expression would be evaluated by Visual Basic in the following steps. (Boldface type highlights what's happening in each step.)

Total = 10 + 15 * 2 / **4 ^ 2**	Exponentiation
Total = 10 + **15 * 2** / 16	Multiplication
Total = 10 + **30 / 16**	Division
Total = **10 + 1.875**	Addition
Total = **11.875**	Result

You can use one or more pairs of parentheses in a formula to clarify the order of precedence. For example, here's how Visual Basic would calculate this simple formula:

```
Number = (8 - 5 * 3) ^ 2
Number = (-7) ^ 2
Number = 49
```

Visual Basic determines the expression between the parentheses (-7) before doing the exponentiation — even though exponentiation has a higher order of precedence than subtraction and multiplication do.

Nested Parentheses

You can further refine the calculation by placing nested parentheses in the formula. Here's the same formula with a new twist:

```
Number = ((8 - 5) * 3) ^ 2
```

Embedding (nesting) the second set of parentheses in the formula directs Visual Basic to calculate the formula this way:

```
Number = ((8 - 5) * 3) ^ 2
Number = (3 * 3) ^ 2
Number = 9 ^ 2
Number = 81
```

As you can see, the results produced by the two formulas are different: 49 in the first formula and 81 in the second. So, adding parentheses can change the result of a mathematical operation as well as make it easier to read.

1. What is the purpose of the Dim keyword?

{ewc mvimg, mvimage,!answer.bmp} A. To declare explicitly a variable used in your program.

{ewc mvimg, mvimage,!answer.bmp} B. To declare implicitly a variable used in your program.

{ewc mvimg, mvimage,!answer.bmp} C. To disable a menu or menu command.

{ewc D. To control the order of precedence in a formula.

mvimg, mvimage,!answer.bmp}

2. If you declare a variable without formally identifying the type of data that you place in it, what type of variable do you create?

{ewc mvimg, mvimage,!answer.bmp} A. String.

{ewc mvimg, mvimage,!answer.bmp} B. Integer.

{ewc mvi C. Single-precision floating point.

mg,mvimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. Variant.

3. If you create an InputBox with the following program statement, what appears in the InputBox title bar?

```
FullName = InputBox$("Please enter your name")
```

{ewc mvimg,mvimage,!an A. InputBox$.

swer.bmp}

{ewc mvimg, mvimage, !answer.bmp} B. Please enter your name.

{ewc mvimg, mvimage, !answer.bmp} C. The full name of the user.

{ewc mvimg, mvima D. The application name of the project.

g
e,
!
a
n
s
w
er
.b
m
p}

4. **If you execute the following program code in Visual Basic, what result does the Label1 object display?**

```
Dim FavoriteFood, FavoriteMeal
FavoriteFood = "Salmon"
FavoriteMeal = FavoriteFood & "Pasta"
Label1.Caption = FavoriteMeal
```

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

A. FavoriteMeal

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er

B. Salmon & Pasta

.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. SalmonPasta

{ewc mvimg, mvimage,!answer.bmp} D. Salmon Pasta

5. Which of the following Visual Basic operators divides one number by another and discards the remainder?

{ewc mvimg, mvimage,!answer.bm A. /

p}

{ewc mvimg,mvimage,!answer.bmp} B. ****

{ewc mvimg,mvimage,!answer.bmp} C. **Mod**

{ewc mvimg,mvimage,!an D. **^**

s
w
er
.b
m
p}

6. What is the purpose of the Val mathematical function?

{ewc mvimg,mvimage,!answer.bmp} A. To return a value that is the square root of the number specified.

{ewc mvimg,mvimage,!answer.bmp} B. To convert a string value to a number.

{ewc mvimg,mvimage,!answer C. To generate a random value.

.bmp}

{ewc mvimg, mvimage,!answer.bmp} D. To convert a numeric value to a string.

7. When Visual Basic evaluates the following formula, what is the result? (Try to guess without using Visual Basic, then test your answer.)

```
4 + 3 ^ 2 * (5 MOD 3) - 6
```

{ewc mvimg, mvimage,!answer.bmp} A. 12.

{ewc mvimg, m B. 16.

vimage,!answer.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. 20.

{ewc mvimg, mvimage,!answer.bmp} D. 92.

Your assignment in this lab is to create a classic magic number program that calculates a surprise “mystery value” from your age and the year of your birth. Here is the procedure you follow by hand to create the magic formula: Write down the year in which you were born, double it, add five to the result, multiply it by fifty, add your current age, subtract 250, and then divide the entire sum by 100. In this lab, you’ll have the program compute the number. Can you guess the result?

To determine this value, your program should include these objects:

® A label describing the purpose of the application.

® A text box object to display the final magic number.

® Two command buttons named **Calculate** and **Quit**.

When the user clicks the **Calculate** button, your program should prompt them for their age and the year of their birth with two InputBoxes, then compute the magic number and display it.

As you create this basic application, you will review the information in this chapter related to creating variables, processing input, using arithmetic operators, and controlling the order of calculation.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **30 minutes**

Objectives

After completing this lab, you will be able to:

® Declare variables in a program.

® Use the **InputBox** function to receive input.

® Use arithmetic operators to build a formula.

® Display output with the **MsgBox** function.

Lab Setup

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Designing the Form

In this exercise, you open a new project and build a form that consists of:

— A label.

— A text box.

— Two command buttons.

You’ll also set a few important object properties.

® Exercise 2: Writing the Program Code

In this exercise, you write an event procedure that:

— Declares variables.

— Receives input.

— Calculates a formula.

— Displays output.

® Exercise 3: Find the Magic Number

In this exercise, you run the program and find out what all the fuss is about.

In this exercise, you open a new project and build a form that consists of:

® A label

® A text box

® Two command buttons

u To create a label

1. Start Visual Basic and open a new, standard Visual Basic program.
2. In the Visual Basic toolbox, click **Label**. Then, create a label object in the middle of your form.
3. Set the **Caption** property to **Magic Number Program**.
4. Use the **Font** property to format the text in 14-point.

u To create a text box

1. In the Visual Basic toolbox, click **TextBox**. Then, create a text box object below the label.
2. Set the **Text** property to empty (remove all of the text from the text box).

u To create two command buttons

1. In the Visual Basic toolbox, click **CommandButton** and create two command buttons. Then, position them below the text box.
2. Use the **Caption** property to name these buttons **Calculate** and **Quit**.

u To save your form and project

1. From the **File** menu, click **Save Project As**.
2. Save your form and project to disk under the name **Magic**.

   Visual Basic will prompt you for two file names — one for your form file (Magic.frm), and one for your project file (Magic.vbp).

In this exercise, you write an event procedure that:

® Declares variables

® Receives input

® Calculates a formula

® Displays output

u To write the code

1. In the form, double-click **Calculate** and type the following code into the Command1_Click event procedure.

   Note In the MagicNum formula, be sure to include each of the parentheses — these control the order of evaluation and are critical for the program to work correctly.

```
Dim Prompt, Year, Age, MagicNum, Output

Prompt = "Please enter the year you were born."
Year = InputBox$(Prompt)
Prompt = "Please enter your current age."
Age = InputBox$(Prompt)

MagicNum = (((Year * 2 + 5) * 50 + Age) - 250) / 100
Output = "The magic number is " & MagicNum
MsgBox Output, , "Calculation Results"
Text1.Text = MagicNum
```

2. In the Code window's **Object** drop-down list box, click **Command2** and type **End** in the Command2_Click event procedure.
3. To save your program to disk, click **Save Project**.

In this exercise, you run the program and find out what all the fuss is about.

u To run the program

1. On the Visual Basic toolbar, click **Start** to run the program.
2. Click **Calculate**, type your birth year in the first input box, and press ENTER. (For example, type **1963**.)
3. In the second input box, type your current age and press ENTER. (For example, type **33**.)

   Your program should do the calculation and display the magic number with a message box.

   If you used the input described above, you'll see **1963.33**. Curiously, this decimal number is a combination of the original values you entered. (What will those mathematicians think of next….)
4. Press ENTER and watch as your program uses the MagicNum variable to display the magic number in a text box on your form.
5. To try another number click **Calculate** or to exit the program, click **Quit**.

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter describes writing conditional expressions that control the flow of program code, and tracking down software defects with Visual Basic’s debugging tools. In this chapter, you will learn how to:

® Understand and use the principles of event-driven programming.

® Use conditional expressions, decision structures, and mathematical operators to control the order that your program executes commands.

® Find and correct errors in your programs.

So far, the programs you have written displayed menus, objects, and dialog boxes on the screen and encouraged users to manipulate screen elements in whatever order they saw fit. These programs put the user in charge, waited patiently for a response, and then processed the input predictably.

In programming circles, this methodology is known as event-driven programming. You build a program by creating a group of intelligent objects (objects that know how to respond when the user interacts with them), and then you process the input by using event procedures associated with the objects.

Older, character-based versions of Basic such as QuickBasic or BASICA lacked visual development tools and executed code in sequence, from beginning to end. Event-driven programs start with graphical objects and then place the user in charge. This different approach requires a different development strategy. In the event-driven model, the programmer's job is to implement tasks that the user wants to accomplish.

This animation provides more information about event-driven programming and how you can create robust and responsive programs in Visual Basic. To view the animation, click this icon:
{ewc mvimg, mvimage,!anim.bmp}

An event-driven program must be ready to respond to almost any operating condition. It’s probably a safe bet, though, that only a few of the routines in a particular program are actually executed. For example, a word processing application may always be prepared to generate a sophisticated mailing list. However, it probably doesn’t actually do so very often. The program code that accomplishes mail merge tasks sits ready, but unused, until the moment is right.

In Visual Basic programs, you can direct the course (flow) of your program by creating efficient interface objects and event procedures. However, you can also control which statements and in which order statements inside the event procedures run. In Visual Basic terminology, this process is called creating conditional expressions.

Note Expressions that can be evaluated as **True** or **False** are also known as Boolean expressions, in which the **True** or **False** result can be assigned to a Boolean variable or property. You can assign Boolean values to certain object properties, Variant variables, or Boolean variables that have been created by using the **Dim** statement and the **As Boolean** keywords.

This section includes the following topics:

When you use conditional expressions in a special block of statements known as a decision structure, you can control the order in which statements are executed. With an If...Then decision structure, your program can evaluate a condition in the program and take a course of action based on the result.

This section includes the following topics:

In Visual Basic, if you include more than one selection criterion in your decision structure, you can test more than one conditional expression in your If...Then and ElseIf decision structures. Visual Basic links the extra conditions by using one or more of the operators shown in this table:

Logical Operator	Meaning
And	If both conditional expressions are True, then the result is True.
Or	If either conditional expression is True, then the result is True.
Not	If the conditional expression is False, then the result is True. If the conditional expression is True, then the result is False.
Xor	If one and only one of the conditional expressions is True, then the result is True. If both are True or both are False, then the result is False.

Note When expressions contain mixed operator types, your program evaluates operators in this order:

® Mathematical operators.

® Comparison operators.

® Logical operators.

Logical Operators at Work

This table lists some examples of logical operators at work. In the expressions, it is assumed that the variable Vehicle contains a value of “Bike” and that the variable Price contains a value of 200.

Logical expression	Result
Vehicle = “Bike” And Price < 300	**True** (both expressions are **True**)
Vehicle = “Car” Or Price < 500	**True** (second condition is **True**)
Not Price < 100	**True** (condition is **False**)
Vehicle = “Bike” Xor Price < 300	**False** (both conditions are **True**)

Demonstration: User Logon Utility

This demonstration shows how you can use the If…Then decision structure and logical operators to validate users as they log onto a program. (You could use similar logic to write a network application, but use this simple utility for tutorial purposes only.) To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

In Visual Basic, your If...Then decision structures can work with single or multiple conditional statements.

Single-condition Structures

In its simplest form, an If...Then decision structure is written on a single line:

```
If condition Then statement
```

where:

condition is a conditional expression.

statement is a valid Visual Basic program statement.

For example, this is an If...Then decision structure that uses a simple conditional expression:

```
If Score >= 20 Then Label1.Caption = "You win!"
```

The decision structure uses the conditional expression, Score >= 20, to determine whether the program should set the **Label1** caption to “You win!” If the Score variable contains a value greater than or equal to 20, Visual Basic sets the **Caption** property. Otherwise, it skips the assignment statement and executes the next line in the event procedure. This sort of comparison always results in a result of **True** or **False**. A conditional expression never results in “Maybe.”

Multiple-condition Structures

Another Visual Basic If...Then decision structure supports several conditional expressions. This decision structure is a block of statements that can be several lines long — it contains these important keywords: **ElseIf**, **Else**, and **End If**.

```
If condition1 Then
    statements executed if condition1 is True
ElseIf condition2 Then
    statements executed if condition1 is False AND condition2 is True
[Additional ElseIf clauses and statements can be placed here]
Else
    statements executed if none of the conditions is True
End If
```

In the decision structure, Visual Basic evaluates condition1 first. If this conditional expression is **True**, the block of statements below it is executed one statement at a time. (You can include one or more program statements.) If the first condition is not **True**, Visual Basic evaluates the second conditional expression (condition2). If the second condition is **True**, the second block of statements is executed. (You can add additional **ElseIf** conditions and statements if you have more conditions to evaluate.) Finally, if none of the conditional expressions is **True**, Visual Basic evaluates the statements below the **Else** keyword. The whole structure is closed at the bottom with the **End If** keywords.

This code sample shows how you could use a multiline If...Then structure to determine the amount of tax due on a hypothetical progressive tax return. To view the sample code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

In this decision structure, Visual Basic performs these steps:

® Tests the variable AdjustedIncome at the first income level.

® Continues to test subsequent income levels until one of the conditional expressions evaluates to **True**.

® Determines an income tax for the taxpayer.

This simple decision structure is quite useful. It could be used to compute the tax owed by any taxpayer in a progressive tax system, such as the one in the United States. (We’re assuming that the tax rates are complete and up to date and that the value in the AdjustedIncome variable is correct.) If the tax rates change, it’s a simple matter to update the conditional expressions.

Important The order of the conditional expressions in your If...Then and ElseIf clauses is critical. For example, reverse the order of the conditional expressions in the tax computation example. If you listed rates in the structure from highest to lowest, this is what happens:

® Taxpayers in the 15 percent, 28 percent, and 31 percent tax brackets would be placed in the 36 percent tax bracket. (That's because they all would have an income that is less than or equal to 250,000.)

® Visual Basic would stop at the first conditional expression that is **True**, even if the others are also **True**.

All of the conditional expressions in this example test the same variable, so they need to be listed in ascending order to get the taxpayers to fall out at the right spots. Moral: When you use more than one conditional expression, watch their order carefully.

In Visual Basic, Select Case decision structure is another way to control the execution of statements in your programs. A Select Case structure is similar to an If...Then structure. However, when the branching depends on one key variable (test case), a Select Case decision structure is more efficient. This efficiency can make your program code more readable and efficient.

This section includes the following topics:

Visual Basic lets you use comparison operators to include a range of test values in a Select Case structure. These are the Visual Basic comparison operators that you can use in your programs:

Comparison Operator	**Meaning**
=	Equal to
< >	Not equal to
>	Greater than
<	Less than
> =	Greater than or equal to
< =	Less than or equal to

The Is and To Keywords

To use the comparison operators, you need to include the **Is** keyword or the **To** keyword in the expression to identify the comparison you're making. The **Is** keyword instructs the compiler to compare the test variable to the expression listed after the **Is** keyword. The **To** keyword identifies a range of values.

Example: Using Is and To

This sample code shows how the decision structure uses **Is**, **To**, and several comparison operators to test the Age variable and to display one of five messages. To view the sample code, click this icon: {ewc mvimg, mvimage,!code.bmp}

If the value of the Age variable is less than 13, the program displays the message, "Enjoy your youth!" For the ages 13 through 19, the program displays the message, "Enjoy your teens!" and so on.

{ewc mvimg, mvimage,!tip.bmp}

In a Review Lab 5, the Select Case structure turns up again to process the choices in a list box.

A Select Case structure begins with the **Select Case** keywords and ends with the **End Select** keywords.

This sample code below contains the syntax for a Select Case decision structure.

```
Select Case variable
Case value1
    statements executed if value1 matches variable
Case value2
    statements executed if value2 matches variable
Case value3
    statements executed if value3 matches variable
.
.
.
End Select
```

u To create a Select Case decision structure

1. Replace *variable* with the variable, property, or other expression that is to be the structure's key value (test case).
2. Replace *value1*, *value2*, and *value3* with numbers, strings, or other values related to the test case.

   If one of the values matches the variable, the statements below its Case clause are executed and Visual Basic continues executing program code after the **End Select** statement.
3. Include any number of Case clauses in a Select Case. If you list multiple values after a case, separate them with commas.

This example shows how you could use a Select Case structure to print an appropriate message about a person's age in a program. If the Age variable matches one of the Case values, an appropriate message is displayed by using a label. To view the sample code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

Using a Case Else Clause

A Select Case structure also supports a Case Else clause that displays a message if none of the earlier cases matches. This program code, which works with the Age example, illustrates the Case Else clause. To view the code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

One of the most useful tools for processing event procedure information is a conditional expression. A conditional expression is part of a complete program statement that asks a true-or-false question about:

® A property

® A variable

® Another piece of data in the program code.

At the heart of every conditional expression is a comparison operator that creates a relationship between values. Here's a simple conditional expression:

```
Price < 100
```

If the Price variable contains a value that is less than 100, the expression evaluates to **True**. If Price contains a value that is greater than or equal to 100, it evaluates to **False**. You can use the comparison operators in this table to create conditional expressions:

Comparison Operator	Meaning
=	Equal to
< >	Not equal to
>	Greater than
<	Less than
> =	Greater than or equal to
< =	Less than or equal to

This table shows some conditional expressions that include some of the comparison operators. (You'll be working with these expressions later in this section.)

Conditional Expression	Result
10 < > 20	**True** (10 is not equal to 20)
Score < 20	**True** if Score is less than 20; otherwise, **False**
Score = Label1.Caption	**True** if the **Caption** property of the **Label1** object contains the same value as the Score variable; otherwise, **False**
Text1.Text = "Bill"	**True** if the word *Bill* is in the first text box; otherwise, **False**

So far, the errors you have encountered in your programs have probably been simple typing mistakes or syntax errors. But what if you discover a nastier problem in your program — one you can't find and correct by a simple review of the objects, properties, and statements in your application? The Visual Basic development environment contains several tools you can use to track down and fix errors (bugs) in your programs. These tools won't stop you from making mistakes, but they often can ease the pain when you encounter a mistake.

This section includes the following topics:

As you develop Visual Basic programs, three types of errors can produce unwanted results in your applications. These are described in the following topics:

® Logic errors

® Syntax errors

® Run-time errors

Note When you encounter error messages produced by syntax errors or run-time errors be sure to use the Visual Basic online Help resources. If a run-time error dialog box appears, click the **Help** button and learn more about the problem.

A logic error is a human error — a programming mistake that makes the program code produce the wrong results. Most debugging efforts focus on tracking down programmer logic errors.

Consider the following If...Then decision structure, which evaluates two conditional expressions and then displays one of two messages based on the result:

```
If Age > 13 And Age < 20 Then
  Text2.Text = "You're a teenager."
Else
  Text2.Text = "You're not a teenager."
End If
```

Can you spot the problem with this decision structure? A teenager is a person who is between 13 and 19 years old, inclusive, yet the structure fails to identify the person who is exactly 13. (For this age, the structure incorrectly displays the message, “You’re not a teenager.”)

This type of mistake is not a syntax error (the statements follow the rules of Visual Basic); it is a mental mistake, or logic error. The correct decision structure contains a greater than or equal to operator (>=) in the first comparison after the **If...Then** statement:

```
If Age >= 13 And Age < 20 Then
```

Believe it or not, this type of mistake is the most common problem in Visual Basic programs. It's a matter of code that works most of the time — but not all of the time — and it's the hardest problem to track down and fix.

A syntax error (compiler error) is a programming mistake that violates the rules of Visual Basic, such as a misspelled property or keyword. As you type program statements, Visual Basic points out several types of syntax errors — and you won’t be able to run your program until you fix each one.

A run-time error is any error — usually an outside event or an undiscovered syntax error — that forces a program to stop running. Two examples of conditions that can produce run-time errors are a misspelled file name in a **LoadPicture** function and an open floppy drive.

To fix a syntax error, edit the incorrect statement (identified by Visual Basic) in the Code window. To fix a logic or run-time error, use break mode or the **Stop** statement to isolate the mistake.

® Using Break Mode

® Using the Stop Statement

One way to identify an error is to execute your program code one line at a time and examine the content of one or more variables or properties as it changes. To do this, you can enter break mode while your program runs and view your code in the Code window.

Break mode gives you a close-up look at your program while the Visual Basic compiler runs it. It's like pulling up a chair behind the pilot and copilot and watching them fly the airplane. But in this case, you can touch the controls.

Visual Basic Resources for Debugging

While you debug your program, you may also want to open and use these Visual Basic resources:

Resource	Function
Debug toolbar	Provides tools devoted entirely to tracking down errors.
Watches window	Displays the contents of critical variables you're interested in viewing.
Immediate window	Provides a place to enter program statements and see their effect immediately.

This illustration shows the **Debug** toolbar, which you open by clicking **Toolbars** in the **View** menu, and then clicking **Debug**. To view the illustration, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

Demonstration: How to Use Break Mode

In this demonstration, you can see how using break mode can help you find and correct the logic error you discovered earlier in the **If...Then** structure. (The error is part of an actual program.)

To isolate the problem, the demonstration uses the **Step Into** button on the **Debug** toolbar, which executes program instructions one at a time. The demonstration also uses the **Quick Watch** button on the **Debug** toolbar to watch the content of the Age variable change. Pay close attention to this two-pronged debugging strategy. You can use it to correct many types of glitches in your own programs. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

In Visual Basic, you can place a **Stop** statement in your code to pause the program and display the Code window. All you have to know is exactly where in the program code you want to enter break mode and start debugging.

For example, you could click the **Break** button to pause the program. Or, you could enter break mode by inserting a **Stop** statement at the beginning of the Command1_Click event procedure.

To view an illustration showing how the **Stop** statement method works, click this icon:
{ewc mvimg, mvimage,!code.bmp}

When you run a program that includes a **Stop** statement, Visual Basic enters break mode as soon as it hits the Stop statement. While Visual Basic is in break mode, you can use the Code window, the **Step Into** button, and the **Quick Watch** button just as you would if you had entered break mode manually. Finally, when you finish debugging, just remove the **Stop** statement.

1. **When evaluated by Visual Basic, what is the result of the following conditional expression?**

```
30 > 20
```

{ewc mvimg, mvimage, !answer.bmp} A. 10.

{ewc mvimg, mvimage, !answer.bmp} B. 50.

{ewc mvimg, mvimage, C. **True**.

!answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} D. **False**.

2. In an If…Then decision structure, which of the following keywords is *not* valid?

{ewc mvimg, mvimage,!answer.bmp} A. **Else**.

{ewc mvimg, mvimage, !an B. **Else If**.

swer.bmp}

{ewc mvimg, mvimage, !answer.bmp} C. **End If**.

{ewc mvimg, mvimage, !answer.bmp} D. **Then**.

3. If Item1 contains the value 15 and Item2 contains the value 30, what value will Visual Basic place in the Label1.Caption object?

```
If Item1 > Item2 Or (Item1 + 10) > Item2 Then
    Label1.Caption = "Buy Item 1"
ElseIf Item2 < Item1 Then
    Label1.Caption = "Buy Item 2"
Else
    Label1.Caption = "Buy Both"
```

```
End If
```

{ewc mvimg, mvimage,!answer.bmp} A. Buy Item 1.

{ewc mvimg, mvimage,!answer.bmp} B. Buy Item 2.

{ewc mvimg, mvimage,!an C. Buy both.

swer.bmp}

{ewc mvimg, mvimage,!answer.bmp} D. <Empty>.

4. Opening the floppy drive while you are reading data from a diskette in a program is likely to create which type of programming error?

{ewc mvimg, mvimage,!answer.bmp} A. Syntax error.

{ewc mvimg, mvimage,!answ B. Run-time error.

er.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. Logic error.

{ewc mvimg, mvimage,!answer.bmp} D. Typing error.

5. Which of the following debugging buttons is *not* on the Debug toolbar?

{ewc mvimg, mvimage,!answer.bm A. **View Code**.

p}
{e B. **Step In**.
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}
{e C. **Quick Watch**.
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}
{e D. **Break**.
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n

s
w
er
.b
m
p}

6. If the Temp variable contains a value of 95, what output will the following If…Then decision structure produce?

```
If (Temp >= 70) Then
    Label1.Caption = "It's nice today!"
ElseIf (Temp >= 80) Then
    Label1.Caption = "A hot one!"
ElseIf (Temp >= 90) Then
    Label1.Caption = "A scorcher!"
End If
```

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

A. It's nice today!

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m

B. A hot one!

p}

{e C. A scorcher!
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

{e D. No output.
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

In this lab, you will create a program that displays a list of choices for the user in a list box and then processes them with a Select Case decision structure. The purpose of this event-driven tool is to display a list of countries and then display a welcome message in a foreign language when the user clicks the country of their choice. By default, the program only contains four countries, but you can customize it to include additional countries and information.

To operate effectively, your program should include these objects:

® Several labels to introduce the program.

® A list box to display the countries.

® Several labels to display the welcome message.

® A **Quit** button to terminate the program.

As you create this basic application, you review the information in this chapter related to managing input, controlling program flow, and processing information.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **30 minutes**

Objectives

After completing this lab, you will be able to:

® Use the **ListBox** control to obtain a choice from the user.

® Use the Select Case decision structure to process the selection.

Lab Setup

This program requires no supporting files. To complete the program, simply follow the instructions in the lab exercises.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Designing the Form

In this exercise, you open a new project and build a form containing these features:

— Four labels.

— A list box.

— A command button.

You'll also set a few important properties.

® Exercise 2: Using Select Case

In this exercise, you:

— Write an event procedure that uses the Select Case decision structure to process the items in the list box.

— Fill the list box by using the List1 object's **AddItem** method in the Form_Load event procedure.

® Exercise 3: Testing the Program

In this exercise, you run the program and verify that each message contains accurate information. If it doesn't, you use the debugging tools to find the problem.

In this exercise, you will open a new project and build a form containing four label objects, a list box object, and a command button. You'll also set a few important properties.

u To create objects on the form

1. Start Visual Basic and open a new, standard Visual Basic application.
2. In the toolbox, click **Label**, and then create a large box in the top middle of the form to display a title for the program.
3. In the toolbox, click **ListBox**, and then create a list box on the left side of the form.
4. Above the list box, create a small label. To display program output, create two small labels on the right side of the screen.
5. In the toolbox, click **CommandButton**, and then create a small command button in the bottom middle of the form.
6. Open the Properties window, and then set these object properties on the form:

Object	Property	Setting
Label1	**Caption**	"International Welcome Program"
	Font	Times New Roman, Bold, 14-point
Label2	**Caption**	"Choose a country"
Label3	**Caption**	(Empty)
Label4	**Caption**	(Empty)
	BorderStyle	1 - Fixed Single
	ForeColor	Medium red (&H00000080&)
Command1	**Caption**	"Quit"

Note The word *(Empty)* in the table means remove all text from the indicated property setting.

7. From the **File** menu, click **Save Project As**, and then save your form and project to disk under the name MyLab5. You will be prompted for two file names — one for your form file (MyLab5.frm) and one for your project file (MyLab5.vbp).

In this exercise, you:

® Write an event procedure that uses the Select Case decision structure to process the items in the list box.

® Fill the list box by using the List1 object's **AddItem** method in the Form_Load event procedure.

u To write the code

1. Double-click the form (not an object, but the form itself).

   The Form_Load event procedure appears in the Code window.
2. To initialize the list box, type the following program code:

   ```
   List1.AddItem "England"
   List1.AddItem "Germany"
   List1.AddItem "Spain"
   List1.AddItem "Italy"
   ```

   These lines of code use the **AddItem** method of the list box object to add entries to the list box on your form.
3. Open the **Object** drop-down list box, and then click the **List1** object in the list box. The List1_Click event procedure appears in the Code window.
4. To process the list box selection with Select Case, type these lines of program code:

   ```
   Label3.Caption = List1.Text
   Select Case List1.ListIndex
   Case 0
       Label4.Caption = "Hello, programmer"
   Case 1
       Label4.Caption = "Hallo, programmierer"
   Case 2
       Label4.Caption = "Hola, programador"
   Case 3
       Label4.Caption = "Ciao, programmatori"
   End Select
   ```

   The first statement in this block of code copies the name of the selected list box item to the caption of the third label on the form. The most important property used in the statement is **List1.Text**, which contains the exact text of the item selected in the list box. The remaining statements are part of the Select Case decision structure. The structure uses the **ListIndex** property of the list box object as a test case variable and compares it to several values. The **ListIndex** property always contains the number of the item selected in the list box; the item at the top is 0 (zero), the second item is 1, the third item is 2, and so on. Using **ListIndex**, the Select Case structure can quickly identify the user's choice and display the correct greeting on the form.
5. Open the **Object** drop-down list box.
6. In the list box, click the **Command1** object.
7. In the event procedure, type **End**, and then close the Code window.
8. To save your program to disk, click **Save Project**.

In this exercise, you run the program and verify that each message contains accurate information. If it doesn't, you use the debugging tools to find the problem.

u To test the program

1. To run the program, click **Start** on the toolbar.

   Visual Basic loads the program and displays your opening form.
2. Click each of the country names in the **Choose A Country** list box.

   The program should display a greeting for each of the countries listed, and the name of the country should appear above the greeting.
3. If you notice a bug in the program, or if you want to watch Visual Basic run the program code:
   - Click **Break** to enter break mode.
   - Run the program one statement at a time by displaying the **Debug** toolbar and clicking **Step Into** repeatedly.
4. When you're finished testing the program, click **Quit** to exit.

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter describes how to write repeating statements (loops) in program code and how to use the **Timer** control to create clocks and other time-related utilities. In this chapter, you will learn how to:

® Write the **For…Next Loop** statement to run a block of code a set number of times.

® Write the **Do… Loop** statement to run a block of code until a specific condition is met.

® Use the **Timer** control to create a digital clock and other special effects.

With a **For...Next** loop, you can execute a specific group of program statements in an event procedure a specific number of times. This can be useful when you want to perform several related calculations, work with elements on the screen, or process several pieces of user input.

This section includes the following topics:

A **For...Next** loop is really just a shorthand way of writing out a long list of program statements. Since each group of statements in the list would do essentially the same thing, Visual Basic lets you define a group of statements and request that it be executed as many times as you want.

For...Next Loop Syntax

The syntax for a **For...Next** loop looks like this:

```
For variable = start To end
    statements to be repeated
Next variable
```

In this syntax statement, **For**, **To**, and **Next** are required keywords, and the equal sign (**=**) is a required operator. First, you replace *variable* with the name of a numeric variable that keeps track of the current loop count. Next, you replace *start* and *end* with numeric values that represent the starting and stopping points for the loop. The line or lines between the **For** and **Next** statements are the instructions that repeat each time the loop executes.

Making a Beeper

For example, the following **For...Next** loop sounds four beeps in rapid succession from the computer's speaker:

```
For i = 1 To 4
    Beep
Next i
```

This loop is the functional equivalent of writing the **Beep** statement four times in a procedure. To the Visual Basic compiler, the loop looks like this:

```
Beep
Beep
Beep
Beep
```

The loop uses the variable *i*. By convention, *i* stands for the first integer counter in a **For...Next** loop. Each time Visual Basic executes the loop, the counter variable increases by one. (The first time through the loop, the variable contains a value of 1, the value of *start*. The last time through, the variable value is 4, the value of *end*.) As you'll see in the following examples, you can use this counter variable in your loops to great advantage.

A counter variable is just like any other variable. You can assign counter variables to properties, use them in calculations, or display them in a program. One of the handiest techniques for displaying a counter variable is to use the **Print** method. This method displays output on a form or prints output with an attached printer. The **Print** method has the following syntax:

```
Print expression
```

where *expression* is a variable, property, text value, or numeric value in the procedure.

Print Method Counter Variable

For example, you could use the **Print** method in a **For…Next** loop to display output on a form:

```
For i = 1 To 10
    Print "Line"; i
Next i
```

The **Print** method displays the word *Line*, followed by the loop counter, 10 times on the form. The **Print** method uses these symbols to separate elements in an expression list:

Symbol	Behavior
comma (**,**)	Places the elements one tab field apart.
semicolon (**;**)	Places elements side by side. (Visual Basic displays the counter variable next to the string with no additional spaces in between.)

If you were to run this program, however, you would probably notice that there *is* a space between "Line" and the counter variable. That's because when the **Print** method prints numeric values, Visual Basic reserves a space for a minus sign, even if the minus sign isn't needed.

Note You can use any combination of semicolons and commas to separate expression list elements.

Using a Command Button

You can also use a command button to run the **Print** method loop. To view an illustration of what you would see on your form when Visual Basic repeatedly executes the **Print** method, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

{ewc mvimg, mvimage,!tip.bmp}

In Visual Basic, you can use loops to change properties and update key variables that:

® Open bitmaps with the **Picture** property.

® Keep a running total with a variable.

® Change the text size on your form by updating the form's **FontSize** property.

This demonstration shows how a **For…Next** loop can change the text size on a form by changing the form's **FontSize** property. (The **FontSize** property adjusts the point size of the text on a form. You can use it as an alternative to changing the point size with the **Font** property.) In the demonstration, you'll see the program run through the loop to create type that seems to grow as the program runs. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

Finally, you may wish to know how to leave a loop early if the need arises. In Visual Basic, you can use an **Exit For** statement to exit a **For...Next** loop before the loop has finished executing. With this capability, you can respond to specific events that occur before the loop runs the preset number of times.

For example, in this **For...Next** loop, the loop prompts the user for 10 names and prints them on the form, unless the user enters the word *Done*. (If the user does enter *Done*, the program jumps to the first statement that follows the **Next** statement):

```
For i = 1 To 10
    InpName = InputBox("Type a name or Done to quit.")
    If InpName = "Done" Then Exit For
    Print InpName
Next i
```

As this example shows, you can use If statements with **Exit For** statements. You'll find this combination useful for handling special cases that come up in a loop, and you'll probably use it often.

Using counter variables in a **For...Next** loop can be a powerful tool in your programs. With a little imagination, you can use counter variables to create several useful sequences of numbers in your loops.

Creating Loops with a Custom Counter

You can create a loop with a counter pattern other than 1, 2, 3, 4, and so on. First, you specify a different start value in the loop. Then, you use the **Step** keyword to increment the counter at different intervals. For example, this loop controls the print sequence of numbers on a form:

```
For i = 5 To 25 Step 5
    Print i
Next i
```

The program prints this sequence of numbers:

```
5
10
15
20
25
```

Specifying Decimal Values

You can also specify decimal values in a loop. For example, here's another **For...Next** loop that controls the print sequence of numbers on a form:

```
For i = 1 To 2.5 Step 0.5
    Print i
Next i
```

The program prints this sequence of numbers:

```
1
1.5
2
2.5
```

Nested For...Next Loops

You can also place one **For…Next** loop inside another to create really interesting effects. If you try this, be sure you use a different counter variable for each loop, such as **i** for the first, and **j** for the second.

Do loops are valuable because occasionally you can’t know in advance how many times a loop should repeat. As an alternative to a **For...Next** loop, you can write a **Do** loop that executes statements until a certain condition in the loop is true.

For example, you might want to let the user enter names in a database until the user types *Done* in an input box. In that case, you could use a **Do** loop to cycle indefinitely until the user enters the text string, *Done*.

This section includes the following topics:

A **Do** loop has several formats, which depend on where and how Visual Basic evaluates the loop condition.

A Standard Do Loop

The most common **Do** loop syntax looks like this:

```
Do While condition
    block of statements to be executed
Loop
```

For example, this **Do** loop consists of statements that process input until the user enters the word *Done*:

```
Do While InpName <> "Done"
    InpName = InputBox("Type a name or Done to quit.")
    If InpName <> "Done" Then Print InpName
Loop
```

The conditional statement in this loop is `InpName <> "Done"`. The Visual Basic compiler translates this statement to mean "loop as long as the InpName variable doesn't contain the word "`Done`."

Note This code brings up an interesting fact about **Do** loops: if the condition at the top of the loop is not **True** when the **Do** statement is first evaluated, Visual Basic never executes the **Do** loop. Suppose, that the InpName variable *did* contain the text string "Done" before the loop started (perhaps from an earlier assignment in the event procedure). Visual Basic would skip the loop altogether and continue with the line below the **Loop** keyword.

Also, note that this type of loop requires an extra **If...Then** structure to prevent the exit value from being displayed when the user types it.

Conditional Test at the Bottom of the Loop

If you want the loop to always run at least once in a program, put the conditional test at the bottom of the loop. For example, this loop is essentially the same as the **Do** loop shown above, but the loop condition is tested after a name is received from the **InputBox** function:

```
Do
    InpName = InputBox("Type a name or Done to quit.")
    If InpName <> "Done" Then Print InpName
Loop While InpName <> "Done"
```

The advantage of this syntax is that you can update the InpName variable before the conditional test in the loop. This syntax prevents a preexisting "Done" value causing the loop to be skipped. Testing the loop condition at the bottom ensures that your loop will be executed at least once, but often, you'll need to add a few extra statements to process the data.

Do loops are relentless, so it is very important that you design test conditions carefully. Each loop must have a **True** exit point. If a loop test never evaluates to **False**, the loop will execute endlessly, and your program will no longer respond to input.

Consider the following example:

```
Do
    Number = InputBox("Enter number to square, or -1 to quit.")
    Number = Number * Number
    Print Number
Loop While Number >= 0
```

In this loop, the user enters number after number, and the program squares each number and prints it on the form. Unfortunately, when the user has had enough, he or she can't quit because the advertised exit condition doesn't work.

When the user enters **-1**, the program squares it, and the Number variable is assigned the value 1. (You can fix this logic error by setting a different exit condition.)

It's a good thing to watch for endless **Do** loops. Fortunately, faulty exit conditions are pretty easy to spot if you test your program thoroughly.

{ewc mvimg, mvimage,!tip.bmp}

So far, the **Do** loops you have seen use the **While** keyword to execute statements as long as the loop condition remained **True**. In Visual Basic, you can also use the **Until** keyword in **Do** loops to cycle *until* a certain condition is true.

Testing a Condition

To test a condition, you can use the **Until** keyword at the top or bottom of a **Do** loop, just as you use the **While** keyword. For example, the following **Do** loop uses the **Until** keyword to loop repeatedly until the user enters the word *Done* in an input box:

```
Do
    InpName = InputBox("Type a name or Done to quit.")
    If InpName <> "Done" Then Print InpName
Loop Until InpName = "Done"
```

Test Condition Operators

As you can see, a loop that uses the **Until** keyword is very similar to a loop that uses the While keyword. In our example, the only difference is that the test condition usually contains the equal to operator (**=**) versus the not-equal-to operator **(<>**).

If using the **Until** keyword makes sense to you, feel free to use it with test conditions in your **Do** loops.

To execute a group of statements for a specified period of time, you can use the **Timer** control in the Visual Basic toolbox. The **Timer** control is an invisible stopwatch that gives your programs access to the system clock. You can use the **Timer** to:

® Count down from a preset time, like an egg timer.

® Delay a program.

® Repeat an action at prescribed intervals.

Objects that you create with the Visual Basic Timer:

® Are accurate to 1 millisecond (1/1000 of a second).

® Aren't visible at run time.

® Are associated with an event procedure that runs whenever the preset timer interval elapses.

Setting the Timer Interval

To set a timer interval, you start by using the timer **Interval** property. Then, you activate the timer by setting the timer **Enabled** property to **True**. When a timer is enabled, it runs constantly. The program executes the timer event procedure at the prescribed interval — until the user stops the program, the timer is disabled, or the **Interval** property is set to 0.

This section includes the following topics:

® Creating a Digital Clock

® Setting a Time Limit for Input

A digital clock is one of the most practical uses for a timer.

To view a demonstration that creates a simple digital clock that keeps track of the current time, down to the second, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

In the demonstration, setting the timer **Interval** property to **1000** directs Visual Basic to update the clock time every 1000 milliseconds (once per second).

Note The Windows operating system is a multi-tasking environment, so other programs will also require processing time. Visual Basic might not always get a chance to update the clock each second, but if it gets behind, it will always catch up. To keep track of the time at other intervals (such as once every tenth of a second), simply adjust the number in the **Interval** property.

There's another interesting way to use a timer: to wait for a set period of time and then to either enable or prohibit an action. This is a little like setting an egg timer in your program — you set the Interval property with the delay you want, and then you start the clock ticking by setting the **Enabled** property to **True**.

This procedure shows you how to set a time limit for input in a program. In the lab exercises at the end of this chapter, you'll adapt these steps to create a password protection program that times out if the user takes more than 15 seconds to enter a password.

u To set a user input time limit

1. Create a **Timer** object on your form.
2. Set the **Interval** property of the **Timer** object to the user input time limit.

   Be sure to specify the time limit in milliseconds. For example, set the **Interval** property to 30000 for a 30-second time limit.
3. In the **Timer** object’s Timer event procedure, place statements that print a "time expired" message and that stop the program. For example:

   ```
   MsgBox ("Sorry, your time is up.")
   End
   ```

u To create an event procedure to manage user input

1. At the place you want to begin the timed input interval, type the following program statement to start the clock:

   ```
   Timer1.Enabled = True
   ```

   You can associate the event procedure with a command button, text box, or any other object that receives input. If you want the clock to start when the form first appears, place the statement in the Form_Load event procedure.
2. To turn off the clock when the user completes the input satisfactorily, use an event procedure with the following statement:

   ```
   Timer1.Enabled = False
   ```

   Without this statement, the timer object event procedure automatically closes the program when the allotted time expires.

{ewc mvimg, mvimage,!tip.bmp}

1. **In the following For…Next loop, which is the counter variable?**

```
For i = 1 to 10
    Print "Line"; i
Next i
```

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

A. For.

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

B. Next.

{e
w
c
m
vi
m
g,
m
vi
m
a

C. 10.

g
e,
!
a
n
s
w
er
.b
m
p}

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p} D. i.

2. When used to separate elements in a Print statement, the comma symbol:

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p} A. Displays the elements side by side.

{e
w
c B. Displays the elements one tab field apart.

mvimg, mvimage, !answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} C. Places the cursor on the next line.

{ewc mvimg, mvimage, !answer.b D. Assigns a new value to a property.

m
p}

3. In the following For…Next loop, how many times will Visual Basic execute the Print statement?

```
For i = 2 to 8 Step 2
    If i = 6 Then Exit For
    Print i
Next i
```

{ewcmvimg,mvimage,!answer.bmp} A. 2.

{ewcmvimg,mvimage,!answer.bmp} B. 3.

{ewcmvimg, C. 4.

mvimage, !answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} D. 8.

4. When would you use a Do loop instead of a For…Next loop?

{ewc mvimg, mvimage, !answer.bm A. When you know in advance exactly how many times you want the loop to repeat.

p}

{ewc mvimg, mvimage,!answer.bmp} B. When you want the program to loop until a certain condition is met.

{ewc mvimg, mvimage,!answer.bmp} C. When you want to avoid an endless loop.

{ewc mvimg, mvimage,!an D. When you plan to use the **Step** keyword to increase the counter variable.

swer.bmp}

5. What word must the user type for the following Do loop to end?

```
Do
    AddName = InputBox("Type a name or Done to quit.")
    If AddName <> "Quit" Then Print AddName
Loop Until AddName = "Continue"
```

{ewc mvimg,mvimage,!answer.bmp} A. Done.

{ewc mvimg,mvimage,!answer.bmp} B. Quit.

{ewc C. Continue.

mvimg, mvimage, !answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} D. No solution—this is an endless loop.

6. Which property starts a timer object running in a program?

{ewc mvimg, mvimage, !ans A. **Time**.

wer.bmp}		
{ewc mvimg, mvimage, !answer.bmp}	B.	**Interval**.
{ewc mvimg, mvimage, !answer.bmp}	C.	**True**.
{ewc mvimg, mvimag	D.	**Enabled**.

e,!answer.bmp}

7. **If you wanted to give a user one minute to enter a password on a form, what value would you assign to the timer object Interval property?**

{ewc mvimg, mvimage,!answer.bmp} A. **True**.

{ewc mvimg, mvimage,!answer.bmp} B. 1.

{ewc C. 60.

mvimg,mvimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. 60,000.

In this lab, you create a program that gives the user 15 seconds to enter a password in a text box. If the user doesn't enter the correct password (**secret)** in the allotted time, the program should display a "Time Expired" message and then close.

To operate effectively, your program should include these objects:

® A text box to receive the password.

® A label to provide instructions and show the time limit.

® A command button that users can click to test their password attempts.

® A timer to handle the countdown.

As you create this basic application, you use the information in this chapter related to using the **Timer** and setting an input time limit.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **30 minutes**

Objectives

After completing this lab, you will be able to:

® Use the **Timer** control.

® Use an **If…Then** decision structure.

® Set a time limit for entering a password.

Lab Setup

This program requires no supporting files. To complete the program, simply follow the instructions in the lab exercises.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Create Objects

In this exercise, you open a new project and build a form that contains these objects:

— A text box.

— A label.

— A command button.

— A timer.

® Exercise 2: Set Timer Properties

In this exercise, you set the properties for the objects on your form. The key properties are **Interval** and **Enabled**, which prepare the timer for the countdown.

® Exercise 3: Process the Password

In this exercise, you write timer and command button event procedures. In the command button event procedure, you use an **If…Then** decision structure to process the user password.

In this exercise, you open a new project and build a form that contains these objects:

® A text box

® A label

® A command button

® A timer

u To design the form

1. Start Visual Basic and open a new, standard Visual Basic application.
2. Resize the form to a small rectangular window about the size of an input box.
3. In the Visual Basic toolbox, click the **TextBox** control, and then create a rectangular text box in the middle of the form.
4. To create a label, click the **Label** control, and create a long label object above the text box.
5. To create a command button, click the **CommandButton** control and create a button object below the text box.
6. To create a timer, click the **Timer** control and then create a timer object in the lower left corner of the form.

u To save your project

1. From the **File** menu, click **Save Project As**.
2. Save your form and project to disk under the name **Passwd**.

   Visual Basic will prompt you for two file names — one for your form file (Passwd.frm), and one for your project file (Passwd.vbp).

In this exercise, you set the properties for the objects on your form. The key properties are **Interval** and **Enabled**, which prepare the timer for the countdown.

Note When the user enters a password, the text box **PasswordChar** property displays asterisks (*) in the text box. You can use this special property any time that you want to give users confidentiality while they enter data in a text box.

---

u To set object properties

To set the object properties, use this table:

Object	Property	Setting
Text1	**Text**	(Empty)
	PasswordChar	*
Label1	**Caption**	"Enter your password within 15 seconds."
Command1	**Caption**	"Try Password"
Timer1	**Interval**	15000
	Enabled	True (default setting)
Form1	**Caption**	"Password"

In this exercise, you write timer and command button event procedures. In the command button event procedure, you use an If…Then decision structure to process the user password.

u To create the code

1. On the form, double-click the timer.
2. Type the following statements in the Timer1_Timer event procedure:

   ```
   MsgBox ("Sorry, your time is up.")
   End
   ```

   If the timer interval reaches 15 seconds and the user hasn't entered a valid password, Visual Basic executes this event procedure. The first statement displays a message that indicates that the time has expired, and the second statement stops the program.
3. In the Code window's **Object** drop-down list box, click the **Command1** object.
4. In the Command1_Click event procedure, type the following statements:

   ```
   If Text1.Text = "secret" Then
       Timer1.Enabled = False
       MsgBox ("Welcome to the system!")
       End
   Else
       MsgBox ("Sorry, friend, I don’t know you.")
   End If
   ```

   This program code tests whether the password entered in the text box is correct. If it is, the timer is disabled, a welcome message appears, and the program ends. (Of course, a more useful program would continue working rather than ending here.)

   If the password entered is not a match, the program notifies the user with a message box, and the user gets another chance to enter the password. But the user had better be quick — he or she has only 15 seconds!

u To run the program

1. On the toolbar, click **Start** to run the program.

   The program starts, and the 15-second clock starts ticking.
2. In the text box, type **open**, and then click **Try Password**.

   The message "Sorry, friend, I don't know you." appears.
3. Click **OK** and wait patiently until the sign-on period expires.
4. When the program displays the message, "Sorry, your time is up", click **OK** again to exit the program.

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter describes how adding forms, printer support, and error-processing to your user interface will enhance your programs. These techniques will help you build applications that are more sophisticated and robust. In this chapter, you will learn how to:

® Add extra forms to the user interface.

® Use the **Printer** object to send output to a printer.

® Create error handlers to manage run-time errors.

So far, each program in this course has used only one form for input and output. In many cases, one form will be enough to communicate with the user. But if you want to add additional screens to a program, Visual Basic supports extra forms that give you lots of flexibility. You can make all of the forms in a program visible at the same time, or you can load and unload forms as the program requires.

Modal Forms

If you display more than one form at once, you can allow the user to switch between forms, or you can control the order in which the forms are used. A form that must be used when it appears on the screen is known as a modal form. This type of form remains on center stage until the user clicks **OK**, clicks **Cancel**, or is otherwise finished with it.

Nonmodal Forms

Forms that the user can switch away from are known as nonmodal (modeless) forms. Microsoft Windows-based applications use nonmodal forms to display information because they give the user more flexibility. That's why, when you create a new form, nonmodal is the default.

Independent Access

You can also set any property on a form — including the caption, size, border style, foreground and background colors, display font, and background picture — independently.

To view an animation that lists several practical uses for extra forms in your programs and describes the difference between modal and nonmodal forms, click this icon:
{ewc mvimg, mvimage,!anim.bmp}

This section includes the following topics:

® Creating New Forms

® Working with Multiple Forms

In the previous topics, you assembled the forms you need by creating new forms or reusing old ones from other projects. Now, you're ready to use these forms in your program. Visual Basic provides several methods, properties, and program statements for managing forms. In this section, topics include:

In Visual Basic, you hide forms by using the **Hide** method and unload forms by using the **Unload** statement. (The **Hide** and **Unload** keywords are the opposites of **Show** and **Load**, respectively.)

Hiding a form makes the form invisible but keeps it in memory for use later in the program. (You can get the same results by assigning **False** to the form's **Visible** property.)

Unloading a form removes the form from memory and frees up the RAM used to store the objects and graphics on the form. However, unloading a form doesn't free up the space used by its event procedures, which remain in memory until the program ends.

To hide and unload Form2, you can use the **Hide** and **Unload** keywords in the following manner:

```
Form2.Hide
Unload Form2
```

Important After you unload a form, its run-time values and properties are lost. If you reload the form, it will contain its original design-time settings.

In most of the programs you write, you'll probably create a standard form, on which the user does most of the work. Then, you'll add special-purpose forms to handle program input and output. However, in Visual Basic, you can also set up a form hierarchy (a special relationship among forms in a program that work best as a group). These hierarchies create special forms called Multiple Document Interface (MDI) forms, which are distinguished by their roles as parent and child forms.

Parent and child forms are especially useful in document-centered applications, in which many windows are used to display or edit several documents. (To learn more about using MDI forms, search for *forms, MDI* in the Visual Basic online Help.)

u **To create MDI parent and child forms**

1. To create an MDI parent form, click the **Project** menu **Add MDI Form** command.
2. To create an MDI child form, click the **Project** menu **Add Form** command, and then set the form's MDIChild property to **True**.

MDI Form Run-time Behavior

At run time, MDI forms behave like regular forms, with the following exceptions:

® Child forms appear within the parent form window.

® When the user minimizes a child form, it shrinks to a small title bar on the MDI form instead of appearing as a button on the taskbar.

® When the user minimizes a parent form, the parent form and all its children shrink to a single button on the Windows taskbar.

® All child menus appear on the parent form menu bar.

® When the user maximizes a child form, its caption appears in the parent form title bar.

® You can display all of a parent form's children by setting the parent form **AutoShowChildren** property to **True**.

When you are ready to display a loaded form, you call it by using the **Show** method and noting whether the form is modal or nonmodal. The **Show** method syntax looks like this:

```
formname.Show mode
```

where *formname* is the name of the form., *mode* is 0 for nonmodal forms (the default) or 1 for modal forms.

For example, to display Form2 as a nonmodal form (the default), you could use the **Show** method without specifying a mode:

```
Form2.Show
```

To display Form2 as a modal form, you would type this statement:

```
Form2.Show 1
```

{ewc mvimg, mvimage,!tip.bmp}

You can maximize a form (expand it to fill the screen) or minimize a form (place it on the Windows taskbar) by using the **WindowState** property. In Visual Basic, assigning the following values to the **WindowState** property gives you these results:

® A 2 maximizes the window.

® A 1 minimizes the window.

® A 0 returns a window to a default state.

For example, to maximize Form1 in a program, you would use this statement:

```
Form1.WindowState = 2
```

The following statement would minimize Form1:

```
Form1.WindowState = 1
```

To return Form1 to its normal (default) view, you would use this statement:

```
Form1.WindowState = 0
```

This demonstration shows how you can display customer service information by adding a second form to an online ordering program. Demonstration topics include:

® Creating a new form.

® Saving the form to disk.

® Using program code to display the form.

To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

After you create a new form in your project, you can load it into memory and open it by using specific event procedure statements. This is the statement syntax that you would use to load a new form:

```
Load formname
```

where *formname* is the name of the form you want to load.

For example, Visual Basic would execute this statement by loading the second form in a program into memory:

```
Load Form2
```

After you load a form, you can use it from any event procedure in the program or access any property or method that you want. For example, to change the Form2 title bar, you could type this statement:

```
Form2.Caption = "Sorting Results"
```

As an object, each new form maintains its own objects, properties, and event procedures. By default, the first form in a program is Form1; subsequent forms are Form2, Form3, and so on. There are two ways to create new forms:

® By clicking the **Add Form** command on the **Project** menu.

® By clicking the **Add Form** button on the toolbar.

This section includes these topics:

To create a new, nonmodal form and prepare it for use with other forms, follow these procedures.

u To create a new form

1. On the Visual Basic toolbar, click **Add Form**.

   The **Add Form** dialog box, which displays the predesigned forms that you can insert, appears.

2. In the dialog box, double-click the Form icon to open a new, general-purpose form in your project.

   A blank form appears in the VB development environment. The new form, which is the same size as the original form (Form1), is named Form2. This illustration shows you a blank new form. To view the illustration, click this icon:
   {ewc mvimg, mvimage,!illust.bmp}

u To set new form properties

1. Resize the second form so that it’s the size and shape that you want.
2. In the Properties window, click (Name) and type an object name with a *frm* prefix. (For example, frmSplashScreen.)
3. To create a title for the form title bar, click the **Caption** property, and then type the title (for example, Welcome).

u To switch back and forth between forms

1. Display the Project window.
2. Double-click the Forms folder.
3. Select the form you want to work with and double-click it.

Now you're ready to add objects to your new form and use it in your program.

In Visual Basic, you can reuse existing forms in new programming projects. With this capability, you can design a splash screen or dialog box once and use it over and over again in your programs. (This is the reason you've been saving your forms as separate .frm files.)

u To add preexisting forms to a project

1. From the Visual Basic **Project** menu, click **Add Form**.
2. In the **Add** dialog box, click the **Existing** tab.

   This illustration shows the **Add Form** dialog box. To view the illustration, click this icon:
   {ewc mvimg, mvimage,!illust.bmp}
3. If necessary, use the dialog box file navigation controls to locate the form on disk, and then click **Open**.

   Visual Basic adds the specified form (and all its event procedures) to the project.

u To view existing forms

1. In the Project window, highlight the form you want.
2. Click **View Object**.
3. To view the form event procedures, click **View Code**.

{ewc mvimg, mvimage,!tip.bmp}

After you create a new form in your project, you need to save it in a separate .frm file on disk.

u To save your new form

1. Make sure that the form you want to save is the active (selected) form.

   This illustration shows a project that contains three forms in the Project window. To view the illustration, click this icon:
   {ewc mvimg, mvimage,!illust.bmp}
2. To make your new form active, highlight the form name in the Project window or click the form on the screen.
3. From the **File** menu, click **Save Form2 As**.

   The **Save File As** dialog box appears.
4. Specify a folder name and file name for the form, and then click **Save**.

   Visual Basic saves the form to disk and records the new form name in the Project window.
5. To switch back and forth between forms, highlight the form name in the Project window, and then click the **View Object** button.

If you no longer need a form, you can remove it from your project. To remove a form, highlight it in the Project window and choose the **Remove *formname*** command from the **Project** menu. (In the previous sentence, *formname* is the name of your form. Visual Basic automatically places the highlighted form name in the menu.)

Now and then, you might want to print a copy of the information your program produces. Visual Basic makes this easy by supplying a **Printer** object.   With this object, you can send output to the active printer registered in Windows Control Panel. Although the **Printer** object can't handle every formatting request, it provides basic font control and error handling capabilities that can give you usable printouts.

This section includes the following topics:

® The Printer Object

® Printing Text

® Printing Graphics

When a default printer is set up in Windows, Visual Basic automatically makes the **Printer** object available to your programs. You can configure the **Printer** object by using several useful properties and methods.

The Print Method

In Visual Basic, you can use the **Print** method to send output to the **Printer** object. (This is the method that you used in Using the Counter Variable in a Loop to display text on a form in a loop.)

For example, the following line sends the text string “Mariners” to the Windows default printer:

```
Printer.Print "Mariners"
```

Setting Printer Object Properties

Unlike forms and objects that you create by using toolbox controls, you can’t set **Printer** object properties by using the Properties window. You must set **Printer** object properties with program code at run time. (The Form_Load event procedure is a good place to put standard settings that you’ll use every time your program runs.)

In all, the **Printer** object has several dozen properties and methods that you can use to control different aspects of printing. But don't worry—many **Printer** object properties and methods are similar to keywords you already use with forms and objects that you create with toolbox controls. The following tables list some of the more useful **Printer** object properties and methods.

Property	Description
FontName	Sets the font name for the text.
FontSize	Sets the font size for the text.
FontBold	True sets the font style to bold.
FontItalic	True sets the font style to italic.
Page	Sets the page number that is being printed.

Method	Description
Print	Prints the specified text on the printer.
NewPage	Starts a new page in the print job.
EndDoc	Signals the end of a printing job.
KillDoc	Terminates the current print job.

Before you print text, you can use the **Printer** object to adjust certain font characteristics. For example, the following code prints “Mariners” in 14-point type:

```
Printer.FontSize = 14
Printer.Print "Mariners"
```

The printer installed in your system must support the font characteristics you choose. Picking a TrueType font usually works best, because most Windows systems automatically include several TrueType fonts and can resize them to work with a variety of printers. (However, experience shows that this process is not perfect, and different printers can produce different results.)

The following statements create two lines of printed output. One line contains the contents of the **Label1** object, and the other contains the contents of the **Text1** text box:

```
Printer.Print ""
Printer.FontName = "Arial"
Printer.FontSize = 14
Printer.FontBold = True
Printer.Print Label1.Caption
Printer.FontBold = False
Printer.Print Text1.Text
Printer.EndDoc
```

This audio clip provides a description of these eight program statements. To listen to the narration, click this icon {ewc mvimg, mvimage,!audio.bmp}

Note When you send output to the **Printer** object, Windows uses the default output device specified in the Windows Printers folder in Control Panel. When you print from Visual Basic, you're not limited to just using printers — the device can be a local printer, a network printer, or a fax modem program.

There's an alternative to printing individual lines by using the **Print** method. You can use the **PrintForm** method to send the entire contents of one or more forms to the printer. With this technique, you can arrange the text, graphics, and user interface elements on a form and then send the entire form to the printer.

Using the PrintForm Keyword

You can use the **PrintForm** keyword in two ways:

® Use **PrintForm** by itself to print the current form.

® Include a form name to print a specific form.

For example, to print the contents of the second form in a program, you could enter this statement in any event procedure in the program:

```
Form2.PrintForm
```

Note The **PrintForm** method prints your form at the current resolution of your display adapter (typically 96 dots per inch.)

Hiding Unwanted Objects

When you use the **PrintForm** method, it's important to consider that the method prints all the objects on a form that are currently visible. For example, you might want to print a bitmap stored in a picture box object but not the other labels and buttons on your form. First, you have to hide the unwanted objects by setting their **Visible** properties to **False**. After you send the form to the printer, however, you need to make the objects visible again.

The following illustration shows a form that contains an image object (with artwork), a descriptive label, and a command button:

{ewc MVIMG, MVIMAGE,!B07G005.BMP}

To print the image object’s contents (but not the label and button objects), you might use program code that looks like this:

```
Label1.Visible = False 'hide objects
Command1.Visible = False
Form1.PrintForm        'print graphic
Label1.Visible = True  'display objects
Command1.Visible = True
```

Try adding printer support to one of your program menus or command buttons. Your users will appreciate the opportunity to create printed output.

Have you experienced a run-time error in a Visual Basic program yet? A run-time error (program crash) is an unexpected event that occurs at run time and that Visual Basic can't recover from.

Run-time Errors

A run-time error occurs whenever the compiler executes a statement that for some reason can't be completed "as dialed". It's not that Visual Basic isn't tough enough to handle the glitch; it's just that the compiler hasn't been told what to do when something goes wrong.

Fortunately, you don't have to live with occasional errors that cause your programs to crash. In Visual Basic, you can write special routines, called error handlers, to respond to run-time errors. An error handler handles a run-time error by telling the program how to continue when one of its statements doesn't work.

Placing Error Handlers

You use error handlers by placing them in the same event procedures as the potentially unstable statements. Error handlers trap a problem by using a special error handling object named **Err**. **Err** has a **Number** property that identifies the error and permits your program to respond to it. For example, if a floppy disk causes an error, your error handler might display a custom error message and then shut down disk operations until the user fixes the problem.

This section includes the following topics:

You can use error handlers in any situation in which an unexpected action might result in a run-time error. Typically, programmers use error handlers to process external events that influence a program — a failed network drive, an open floppy drive door, or an offline printer.

This table lists potential problems that you can address successfully by using error handlers:

Problem source	**Description**
Network	Network drives or resources that fail, or “go down,” unexpectedly.
Floppy disk	Unformatted or incorrectly formatted disks, open drive doors, or bad disk sectors.
Offline printers	Printers that are offline, out of paper, or otherwise unavailable.
Overflow errors	Printing or drawing too much information on a form.
Out-of-memory errors	Running out of application space or resource space in Windows.
Clipboard	Problems with data transfer or the Windows Clipboard.
Other errors	Syntax or logic errors undetected by the compiler and previous tests (such as an incorrectly spelled file name).

The program statement used to detect a run-time error is **On Error**. You place **On Error** in an event procedure, just before you use the statement you're worried about. The **On Error** statement sets (enables) an error handler by telling Visual Basic where to branch if it encounters an error.

Syntax

An **On Error** statement syntax looks like this:

```
On Error GoTo label
```

where *label* is the name of your error handler.

Error handlers are typed near the bottom of an event procedure, following the **On Error** statement. Each error handler has its own label, which is followed by a colon (:) for identification purposes. Examples include:

`ErrorHandler:` or `PrinterError:`

Error Handler Structure

An error handler usually has two parts:

- The first part typically uses the **Err** object's **Number** property in a decision structure (such as **If...Then** or **Select Case**) and then displays a message or sets a property based on the error.
- The second part is a **Resume** statement that sends control back to the program so that the program can continue.

The **Err** object contains a few other properties that you might want to use to display additional error handler information. The **Description** property contains the error message that’s returned to Visual Basic when a run-time error occurs. Whether or not you plan to respond to the error programmatically, you can use this message as an additional source of information for the user.

Using the Description Property

For example, if an error occurs when you load artwork from a floppy disk, this error handler uses the **Description** property to display an error message:

```
    On Error GoTo DiskError
    Image1.Picture = LoadPicture("a:\prntout2.wmf")
    Exit Sub                  'exit procedure

DiskError:
    MsgBox Err.Description, , "Loading Error"
    Resume                    'try LoadPicture function again
```

Trapping Floppy Disk Problems

You can use this technique to trap floppy disk problems such as unformatted disks, missing files, and open drive doors. When the user fixes the problem and clicks **OK** in the message box, the error handler uses the **Resume** statement to try the loading operation again. When the file eventually loads, the **Exit Sub** statement ends the event procedure.

{ewc mvimg, mvimage,!tip.bmp}

Another error handler strategy you can use is to try an operation a few times and then jump over the problem if it isn’t resolved.

Using a Counter Variable

This sample code is an error handler that uses a counter variable. The variable, named Retries, tracks the number of times an error message is displayed. If the program fails more than once, it skips the loading statement. To view the sample code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

If the error you’re handling is a problem that the user might be able to fix, this is a useful technique. The important thing to remember here is that **Resume** retries the statement that caused the error, and **Resume Next** skips the statement and moves on to the next line in the event procedure.

Using Resume Next

When you use **Resume Next**, be sure that the next statement really is the one you want to execute, and when you continue, make sure that you don’t accidentally run the error handler again. A good way to skip over the error handler is to use the **Exit Sub** statement; or you can use **Resume Next** with a label that directs Visual Basic to continue executing after the error handler.

You can use the **Resume** keyword three different ways:

® The **Resume** statement alone.

® With the **Resume Next** keywords.

® With the **Resume** keyword and a label you want the program to branch to. The branch you choose depends on where in the program you want to continue. This label must be in the same procedure as the error handler.

Resume

The **Resume** keyword returns control to the statement that caused the error (in hopes that the error condition will be fixed or won't happen again). If you're asking the user to fix the problem — by closing the floppy drive door or tending to the printer, for example — using the **Resume** keyword is a good strategy.

Resume Next

The **Resume Next** keywords return control to the statement *following* the one that caused the error. If you want to skip the command and continue working, using the **Resume Next** keywords is the strategy to take.

Resume with a Label

You can also follow the **Resume** keyword with a label you'd like to branch to. This gives you the flexibility of moving to any place in the event procedure you want to go. A typical location to branch to is the last line of the procedure.

Demonstration: Creating an Error Handler

This demonstration shows how you can create an error handler to recover from run-time errors associated with floppy disk drives. You can use the same technique to add error handling support to any Visual Basic program — just change the error numbers and messages. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

1. Which of the following ready-to-use forms is *not* supplied in the Add Form dialog box?

{ewc mvimg, mvimage,!answer.bmp} A. About Dialog.

{ewc mvimg, mvimage,!answer.bmp} B. Splash Screen.

{ewc mvimg, mvimage,!answer.bmp} C. Log In Dialog.

{ewc D. Error Handler.

mvimg,mvimage,!answer.bmp}

2. What is the proper term for a form that the user can switch away from freely in a program?

{ewc mvimg,mvimage,!answer.bmp} A. Modal form.

{ewc mvimg,mvimage,!answer.bmp} B. Nonmodal form.

{ewc mvim C. MDI Form.

g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p} D. Startup form.

3. What window state does the following program statement create for Form1?

```
Form1.WindowState = 1
```

{ewc mvimg,mvimage,!answer.bmp} A. Minimized.

{ewc mvimg, B. Maximized.

mvimage,!answer.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. Open.

{ewc mvimg, mvimage,!answer.bmp} D. Hide.

4. What is the purpose of this program statement?

```
Printer.EndDoc
```

{ewc mvimg, mvimage, !answer.bmp} A. Starts a new page in the print job.

{ewc mvimg, mvimage, !answer.bmp} B. Terminates the current print job.

{ewc mvimg, mvimage, ! C. Signals the end of a printing job.

answer.bmp}

{ewcmvimg,mvimage,!answer.bmp} D. Adds a text string to the current print job.

5. What type of information does the PrintForm method send to the printer?

{ewcmvimg,mvimage,!answer.bmp} A. Objects containing graphics only.

{ewcmvimg,mvimage,!ans B. Objects containing text only.

w
er
.b
m
p}

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p} C. MDI forms only.

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p} D. All the visible objects on the form.

6. In an error handler, the Resume Next statement:

{ew
c
mvi
mg,
mvi
ma
ge,!
ans
wer A. Returns control to the statement that caused the error.

.bmp}

{ewc mvimg, mvimage, !answer.bmp} B. Returns control to the statement following the one that caused the error.

{ewc mvimg, mvimage, !answer.bmp} C. Enables the error handler.

{ewc mvimg, mvimage, !answer.bmp} D. Uses an error number to diagnose the run-time error.

{ewc mvimg, mvimage, !a

n
s
w
er
.b
m
p}

7. **What property would you use to print a description of the error message that the compiler generates when it encounters a run-time error?**

{ewc mvimg, mvimage,!answer.bmp} A. **Resume Next**.

{ewc mvimg, mvimage,!answer.bmp} B. **Err.Number**.

{ewc mvimg, mvimage,!ans C. **Err.Description**.

w
er
.b
m
p}

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

D. **MsgBox**.

In this lab, you edit an existing foreign-language vocabulary program by adding a second form to display definitions in a list box. Currently, the program displays definitions by using a **MsgBox** function. However, this method is rather crude and doesn't give you the chance to dress up your output with custom buttons or bitmaps.

To complete this lab, you should add a new form to the program. This new form, named Definition, includes these objects:

® An image box.

® A label.

® A text box.

® A command button.

The foreign language for this program is Italian, but don't worry—Visual Basic supplies a list of words and their definitions. As you create this basic application, you review concepts related to adding new forms and controlling them with program code.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **40 minutes**

Objectives

After completing this lab, you will be able to:

® Open and modify an existing program.

® Add a second form to a program.

® Display and hide the new form with program code.

Lab Setup

When you're ready to begin the lab, follow these steps:

1. Start Visual Basic.
2. Open Lab7.vbp.

   If you installed *Learn Microsoft Visual Basic 6.0 Now* with the default settings, this file will be in the \LVB6\Ch07 folder on your hard drive.
3. In the Project window, highlight the Lab7.frm form.
4. To examine the Italian Vocabulary form, click **View Object**.

Then, complete the exercises to add the form. When you're asked to open a bitmap of the Italian flag, load the Flgitaly.ico file from the same folder where you found Lab7.vbp.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Create the Second Form

In this exercise, you add a new form to the project and create these objects on the form:

— An image.

— A label.

— A text box.

— A command button.

® Exercise 2: Use the Hide Method

In this exercise, you use the **Hide** method in an event procedure to close the Definition form when the user clicks **Close**.

® Exercise 3: Display the Form

In this exercise, you add code to the List1_DblClick event procedure in Form1. This code displays the new form when the user double-clicks a word in List1.

In this exercise, you add a new form to the project and create these objects on the form:

® An image

® A label

® A text box

® A command button

u To add a form

1. In the \LVB6\Ch07 folder on your hard disk drive, open Lab7.vpb. (For more information, see Review Lab 7.)
2. From the **Project** menu, click the **Add Form** command to add a second form.
3. In the **Add Form** dialog box, click the Form icon, and then click **Open**.

u To create form objects

1. Resize the form to a small rectangular window about the size of an input box.
2. Create a small image object on the left side of the form.
3. Create a label in the middle-top of the form.
4. Create a large text box below the label in the middle of the form.
5. Create a command button on the right side of the form.

u To set properties for the objects on your new form

To set object properties, use this table:

Object	**Property**	**Setting**
Image1	**Picture**	"*<drive>*\LVB6\Ch07\Flgitaly.ico"
	Stretch	True
Label1	**Font**	Times New Roman, Bold, 14-point
Text1	**TabStop**	False
Command1	**Caption**	"Close"
Form2	**Caption**	"Definition"

Note Setting the text box **TabStop** property to **False** keeps the text box from getting the focus when the user presses the TAB key. If you don't set this property, the cursor will blink in the text box when the form appears.

u To save the project to disk

1. From the **File** menu, click **Save Project As**.
2. Save your forms under the names **Def1** and **Def2** and save your project under the name **Def**.

In this exercise, you use the **Hide** method in an event procedure to close the Definition form when the user clicks **Close**.

u **To use the Hide method**

1. To write an event procedure that closes the Definition form when the user is finished with it, double-click the **Close** button.
2. In the Command1_Click event procedure, type the following statement:

   ```
   Form2.Hide
   ```

   When the user clicks **Close**, this statement makes Form2 invisible. Form2 appears in a nonmodal state, so the user is free to switch between Form1 and Form2 while the program runs. The user closes Form2 by clicking **Close** on the form.
3. In the Project window, right-click the project name and click **Save Project** on the short-cut menu to save the revised event procedure.

In this exercise, you add code to the List1_DblClick event procedure in Form1. This code opens the new form (Form2) when the user double-clicks a word in List1.

u **To open the new form**

1. In the Project window, click **Form1** to select the first form for editing.
2. In the Project window, click **View Object**.
3. On the form, double-click the list box to display the List1_DblClick event procedure in the Code window.
4. Scroll to the bottom of the procedure so that this line of code is visible:

   ```
   MsgBox Def, , List1.Text
   ```

   This is the **MsgBox** statement you want to replace.
5. Delete the **MsgBox** statement, and then type the following program code in its place:

   ```
   Load Form2
   Form2.Label1 = List1.Text
   Form2.Text1 = Def
   Form2.Show
   ```

   The first statement loads Form2 into memory. The second statement puts a copy of the selected Italian word into the label on Form2. The third statement puts the definition of the selected Italian word into the text box on Form2. The final statement displays the completed form on the screen.
6. To save your changes, click **Save Project** on the Standard toolbar.

   Your program is complete.

u **To run and test the program**

1. To run the utility, click **Start**.
2. In the list box, double-click the verb *cucinare*.

   The program displays the definition for the word on the second form. Now, practice switching back and forth between the forms.
3. Click the first form, and then double-click the word *scrivere.*

   The program displays the definition for *scrivere* (to write) on the second form. The forms are nonmodal, so you can switch between them as you wish.
4. On the second form, click **Close**.

   The program hides the form.
5. To exit the program click **Quit** on the first form.

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter describes how to enhance your program user interface with artwork and special effects. The skills you learn will teach you more about graphics programming and using a coordinate system — and the results will make your applications more interesting to use. In this chapter, you will learn how to:

® Create basic artwork with the **Line** and **Shape** controls.

® Add drag-and-drop support to your user interface.

® Create animation effects with the **Timer** control.

In Windows-based applications, users execute many commands by clicking menus and buttons with the mouse. With Visual Basic, you can give users another way to perform some actions in your programs — the drag-and-drop mouse motion.

To use the drag-and-drop technique, you hold the mouse button down on an object and drag it from one location to another. Then, you release the mouse button to drop the object, which relocates it or issues a command. Moving text from one location to another in a word processing program is a useful drag-and-drop application. Another would be dragging unwanted items to a trash can or other receptacle to remove them from the screen.

In this section, you learn how to create drag-and-drop effects in your programs. This section includes the following topics:

® Setting Drag Mode

® Using the Tag Property

Visual Basic controls provide drag-and-drop support with the **DragMode** and **DragIcon** properties and with the DragDrop and DragOver event procedures.

To set drag mode (allow the user to drag an object), you first set the object’s **DragMode** property to 1. As an option, you can also use the **DragIcon** property to specify a drag icon, which appears while the mouse pointer drags the object.

When the user drops an object on the form, Visual Basic executes the DragDrop event procedure for the object on which the icon was dropped. Similarly, when one object is dragged over another, Visual Basic executes the DragOver event procedure.

In both event procedures, the *Source* parameter identifies the object that has been dropped or moved over. This special parameter and other program statements in the DragDrop and DragOver event procedures are how you control what happens during drag-and-drop events.

To view an animation demonstrating the steps of drag-and-drop programming, click this icon:
{ewc mvimg, mvimage,!anim.bmp}

This section includes the following topics:

® Drag and Drop

® Mouse Pointer

Mouse pointers are useful tools that use graphics to show the user how to use the mouse. In the drag-and-drop demonstration, you learned how to set the **DragIcon** property to change the mouse pointer. You can also change the standard mouse pointer to one of 16 ready-made pointer designs.

In a Visual Basic program, you can change the mouse pointer at any time. To change the pointer, you can use either of these methods:

® Loading a standard pointer shape by using the **MousePointer** property.

® Loading a custom pointer shape by using the **MouseIcon** property.

MousePointer Property

There are two ways to use the **MousePointer** property. The first is to set the property for an object on a form. When the user moves the mouse pointer over the object, the mouse pointer changes to the specified shape. The second way is to set the property for the form itself. In this case, the mouse pointer will change to the shape you specify. (Unless, of course, the pointer is positioned over an object whose **MousePointer** property is not set to the default of 0.)

The following table lists a few of the pointer shapes that you can select by using the **MousePointer** property. (For a complete list of pointer shapes, check the Properties window.)

Pointer Design	Shape #	Description
Cross	2	Crosshairs pointer for drawing.
I-Beam	3	I-beam pointer for text-based applications.
Size	5	Sizing pointer (pointers that have the arrows pointing in other directions are available).
Hourglass	11	Hourglass pointer, which indicates that the user needs to wait.
No Drop	12	No-drop pointer, which indicates that the action the user is attempting to perform can’t be performed.
Custom	99	No standard pointer — use the mouse icon property.

MouseIcon Property

If you specify shape 99 (Custom), Visual Basic uses the **MouseIcon** property to load a custom pointer shape.

Here's a demonstration that shows you how to add drag-and-drop functionality to your applications. The program creates a burn barrel icon that is similar in some ways to the Windows Recycle Bin. However, you empty this trash can by dragging a match icon to the burn barrel and setting it on fire! To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

The program uses image objects for the screen elements and hides them in a DragDrop event procedure, which you set by changing their **Visible** property to **False**. You can use the burn barrel in your programs as a way for the user to dispose of a variety of objects, including:

® Unwanted documents

® Files

® Artwork

® Electronic mail

® Network connections

® Screen elements

With the **Tag** property, you place an identification note (tag) in an object. A tag is simply a piece of descriptive text. Your program checks tags to identify the object it's working with in an event procedure. Programmers typically use the **Tag** property to store these types of information:

Stored Information	**Example**
State of toolbar buttons	Up, Down, or Disabled.
Object type	Button.
Object contents	File, Trash.

For example, the following program statements check the **Tag** property of an object being dropped into a trash can image, EmptyBarrel. (The *Source* parameter contains the object name.) If the **Tag** property contains the word "Fire", then the routine copies a picture from the FlameBarrel image into EmptyBarrel.

```
If Source.Tag = "Fire" Then
    imgEmptyBarrel.Picture = imgFlameBarrel.Picture
End If
```

Creating shapes and dragging objects add visual interest to a program. For programmers, though, animation has always been the king of graphical effects. Animation is the simulation of movement produced by rapidly displaying a series of related images on the screen.

In a way, the drag and drop technique is the poor man’s animation — it lets you move images from one place to another on a form. But real animation requires you to move objects with program code. Often, animation involves changing the size or shape of the images along the way. This section includes the following topics:

® The Form's Coordinate System

® Moving Objects

® Other Animation Effects

Moving images in relation to a predefined coordinate system on the screen is a common trait of animation routines. In Visual Basic, each form has its own coordinate system. The starting point (origin) of this system is the form's upper left corner.

The Default Coordinate System

The default coordinate system consists of rows and columns of twips. (A twip is 1/20 point, or 1/1440 inch.) In the Visual Basic coordinate system, rows of twips are aligned to the x (horizontal) axis, and columns of twips are aligned to the y (vertical) axis. To define locations in the coordinate system, you identify the intersection of a row and column with the notation (*x*, *y*). You can change units in the coordinate system scale, but the (*x*, *y*) coordinates of the upper left corner of a form are always (0, 0).

To view an illustration showing how the Visual Basic coordinate system describes an object's location on a form, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

Visual Basic includes a special method, with which you move objects in a form’s coordinate system. This method, named the **Move** method, uses the following syntax:

```
object.Move left, top
```

where:

object is the name of the object on the form that you want to move.

left is the x screen coordinate of the new location for the object (measured in twips).

top is the y screen coordinate of the new location for the object (measured in twips).

The *left* measurement is also the distance between the left edge of the form and the object; the *top* measurement is the distance between the top edge of the form and the object.

For example, this Visual Basic statement moves the **Picture1** object to the location (1440, 1440) on the screen, or exactly one inch from the top edge of the form and one inch from the left edge of the form:

```
Picture1.Move 1440, 1440
```

This section includes the following topics:

® Relative Movement

® Moving a Collection

® Making an Exception

In Visual Basic terminology, the entire set of objects on a form is called the Controls collection. To animate all the objects on your form, you use the Controls collection with a special **For…Each** loop.

This loop processes each object according to a pattern that you set. For example, if you put the following program statements into a command button event procedure, Visual Basic moves each object on your form 200 twips to the right each time you click the command button:

```
Dim Ctrl as Object
For Each Ctrl In Controls
    Ctrl.Left = Ctrl.Left + 200
Next Ctrl
```

The result is a simple animation effect that makes objects seem to march in unison.

You can also use the **Move** method to specify relative movement. Relative movement is the distance the object should move from its current location. When you specify relative movements, you use:

® The **Left** and **Top** properties of the object. (These values maintain the object's x- and y-axis location.)

® A plus (**+**) or minus (**-**) operator.

For example, this statement moves the **Picture1** object from its current position on the form to a location 50 twips closer to the left edge and 75 twips closer to the top edge:

```
Picture1.Move Picture1.Left - 50, Picture1.Top - 75
```

You don't have to move all of the objects in the Controls collection at once. To make just some of them move, simply assign the ones you want to remain stationary with a **Tag** property. With this tag, your **For…Each** loop skips over them. In the following program statements, for example, each object on your form (except the ones containing the “Button” tag) move right:

```
Dim Ctrl as Object
For Each Ctrl in Controls
    If Ctrl.Tag <> "Button" Then
        Ctrl.Left = Ctrl.Left + 200
    End If
Next Ctrl
```

Here’s an illustration that shows how you might use a **For...Each** loop with exceptions. Each time the user clicks the command button, the four image objects move to the right—the command button itself, however, remains stationary. (Its **Tag** property has been set to "Button".) To view the illustration, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

Collections are useful tools in other programming situations, including controlling forms, switching printers, and managing databases. For more information, search for *collections* in the Visual Basic online Help.

This section highlights a few entertaining ways to add animation effects to your programs. Topics include:

In addition to maintaining **Top** and **Left** properties, Visual Basic maintains **Height** and **Width** properties for most objects on a form. You can use the **Height** and **Width** properties in clever ways to expand and shrink objects while a program runs.

Expanding an Object

If you want to expand (zoom in on) an object, increase its **Height** and **Width** properties in an event procedure. For example, this code expands an image object by 150 twips in height and 200 in width:

```
Image1.Height = Image1.Height + 150
Image1.Width = Image1.Width + 200
```

You'll see this technique demonstrated in Review Lab 8.

Shrinking an Object

If you want to shrink (zoom out on) an object, decrease its **Height** and **Width** properties in an event procedure. For example, this code shrinks an image object by 150 twips in height and 200 in width:

```
Image1.Height = Image1.Height - 150
Image1.Width = Image1.Width - 200
```

The trick to creating animation in a program is placing one or more **Move** methods in a timer event procedure. That way, the timer will cause one or more objects to drift across the screen at set intervals.

In Creating a Digital Clock, you learned to use a timer object that updated a simple clock utility every second, so that the object displayed the correct time. When you create animation, you set the timer Interval property at a much faster rate — 1/5 second (200 milliseconds), 1/10 second (100 milliseconds), or less. The exact rate you choose depends on how fast you want the animation to run.

{ewc mvimg, mvimage,!tip.bmp}

Stopping the Animation

Using the **Top** and **Left** object properties to sense the top and left edges of the form is another important animation technique. By using these properties in an event procedure, you can stop the animation (disable the timer) when an object reaches the edge of the form. You can also make an object appear to bounce off one or more edges of the form. To achieve this feat, you would use any combination of the **Top** and **Left** properties in an **If...Then** or a **Select Case** decision structure.

Using the Top Property

For example, the following program statements use the **Top** property to check if an object has reached the top edge of the form during an animation. If the object reaches the edge, the object becomes invisible, and the timer object (which runs the animation) is disabled.

```
If Picture1.Top > 0 Then
    Picture1.Move Picture1.Left - 50, Picture1.Top - 75
Else
    Picture1.Visible = False
    Timer1.Enabled = False
End If
```

The decision structure uses a **Move** method to move the cloud 50 twips closer to the left edge of the form and 75 twips closer to the top edge of the form. When the timer is set to 65 milliseconds per clock tick, the clouds drift gently by.

This demonstration shows how to use the **Move** method, a picture box, and a timer object to create a drifting-cloud animation sequence. The example is an enhancement to the Drag program demonstrated earlier in this chapter. This time, when you drop a match in the burn barrel, a smoke cloud appears and gently drifts off in the wind. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

1. Which control would you use to create a rounded rectangle on your form?

{ewc mvimg, mvimage, !answer.bmp} A. **Line**.

{ewc mvimg, mvimage, !answer.bmp} B. **Shape**.

{ewc mvimg, mvimage, !a C. **PSet**.

n
s
w
er
.b
m
p}

{e w c m vi m g, m vi m a g e, ! a n s w er .b m p} D. **Timer**.

2. Which property enables the user to use the drag-and-drop technique on an object?

{e w c m vi m g, m vi m a g e, ! a n s w er .b m p} A. **DragDrop**.

{e w c m vi m g, B. **DragOver**.

mvimage, !answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} C. **DragIcon**.

{ewc mvimg, mvimage, !answer.bmp} D. **DragMode**.

3. In a Visual Basic program, how is the Tag property typically used?

{ewc mvimg, mvimage, !answer.bmp} A. To receive a control name at the beginning of a DragDrop or DragOver event procedure.

{ewc mvimg, mvimage, !answer.bmp} B. To hold a piece of descriptive text that can be used to identify the object state, type, or contents in a decision structure.

{ewc mvimg, mvimage, !an C. To hold an icon used during drag-and-drop events.

swer.bmp}

{ewc mvimg, mvimage, !answer.bmp} D. To display the mouse pointer in one of 16 predefined shapes.

4. In the DragDrop and DragOver event procedures, which of the following items identifies the object that triggered the drag-and-drop event?

{ewc mvimg, mvimage, !answer.bmp} A. The *Source* parameter.

{ewc mvimg, B. The **Picture** property.

mvimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} C. The **Tag** property.

{ewc mvimg,mvimage,!answer.bmp} D. The **Visible** property.

5. What are twips?

{ewc mvimg, mvimage, !answer.bmp} A. A device-independent measuring system used to position graphics on a form.

{ewc mvimg, mvimage, !answer.bmp} B. The rows and columns of dots on the Visual Basic form.

{ewc mvimg, mvimage, !an C. A measuring system with units approximately equal to one inch.

s
w
er
.b
m
p}

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

D. A breakfast cereal popular in the 1970s.

6. In what direction does the following program statement move the Image1 object?

```
Image1.Move Image1.Left + 50, Image1.Top + 75
```

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

A. Right.

{e
w
c
m
vi
m

B. Left.

g,mvimage,!answer.bmp}

{ewc mvimg,mvimage,!answer.bmp} C. Right and Down.

{ewc mvimg,mvimage,!answer.bmp} D. Left and Up.

7. To end an animation sequence in a program, which statement should you use?

{ewc mvimg, mvimage,!answer.bmp} A. Picture1.Visible = False.

{ewc mvimg, mvimage,!answer.bmp} B. Source.Visible = False.

{ewc mvimg, mvimage,!an C. Timer1.Enabled = True.

s
w
er
.b
m
p}
{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

D. Timer1.Enabled = False.

In this lab, you use the Image control's **Height** and **Width** properties to expand a planet bitmap each time you click it. When you run the program, it creates an animation effect that simulates a spaceship zooming through space up to a planet.

To complete this lab, you should create an image object in the upper left corner of the form, and create an event procedure that expands the height and width of the image object 200 twips each time you click it. As you create this simple application, you will review the information in this chapter related to moving objects and using a form's coordinate system.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **20 minutes**

Objectives

After completing this lab, you will be able to:

® Control an image object with **Height** and **Width** properties.

® Create an animation effect by zooming in on an object.

Lab Setup

In this lab you load the Earth.ico bitmap from disk. If you installed *Learn Microsoft Visual Basic 6.0 Now* with the default settings, Earth.ico will be in the \LVB6\Ch08 folder on your hard drive.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Open a Planet Bitmap

In this exercise, you create an image object on the form and use its **Picture** property to open a bitmap.

® Exercise 2: Create a Zoom Effect

In this exercise, you use the image object’s **Height** and **Width** properties to write an event procedure that creates the animation.

In this exercise, you create an image object on the form and use its **Picture** property to open a bitmap.

u **To design the form**

1. Start Visual Basic and open a new, standard Visual Basic application.
2. Click the **Image** control in the toolbox, and then draw a small image object near the upper-left corner of the form.

u **To set the object properties**

To set the form and image object properties, use this table:

Object	Property	Setting
Form1	**Caption**	“Approaching Earth”
Image1	**Stretch**	True
	Picture	“<*drive*>\LVB6\Ch08\Earth.ico”

u **To save the project**

1. From the **File** menu, click **Save Project As**.
2. Save your form and project to disk under the name **Zoom**.

In this exercise, you use the image object's **Height** and **Width** properties to write an event procedure that creates the animation.

u To write the code

1. Double-click the **Image1** object on the form.
2. Type the following program code in the Image1_Click event procedure:

   ```
   Image1.Height = Image1.Height + 200
   Image1.Width = Image1.Width + 200
   ```

   These two lines increase the height and width of the Earth icon by 200 twips each time the user clicks the image.

u To run the program

1. On the form, click **Start** to run the program.
2. To expand the Earth icon on the screen, click it several times.

u To exit the program and save your changes

1. When you get close enough to establish a standard orbit, click **Close** on the title bar to quit the Zoom program.
2. To save your changes, click **Save Project**.

You’ve already learned how to add bitmaps, icons, and Microsoft Windows metafiles to a form by creating picture boxes and image objects. In Visual Basic, adding ready-made artwork to your programs is easy. In fact, you’ve had practice working with graphics in almost every chapter.

In this section, you’ll learn how to create original artwork on your forms by using two drawing controls and a collection of useful graphics methods. You can find these handy tools in the Visual Basic toolbox, and you can use them to build images in a variety of different shapes, sizes, and colors. The objects you create with the toolbox controls do have a few limitations — they can’t receive the focus at run time, and they can’t appear on top of other objects — but they are powerful, fast, and easy to use. This section includes the following topics:

Creating a line on a Visual Basic form is a three-step process. You start by creating the line with the **Line** control. Next, you set a variety of properties that change the appearance of the line you create, just as you do for other objects. Finally, if it's necessary to move or resize the line, Visual Basic surrounds it with selection handles.

To view an illustration showing several lines created with the **Line** control, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Here's a list of the most important **Line** control properties and the purpose of each:

Property	Purpose
BorderWidth	Adjusts the thickness of the line on your form. This is especially useful when you are creating an underline, or a line that separates one object from another.
BorderStyle	Makes the line solid, dotted, or dashed.
BorderColor	Sets the line color to any Visual Basic standard color.
Visible	Hides or displays the line as needed in your program.

You can use the **Shape** control to build complex images on your forms by combining lines with rectangles, squares, ovals, and circles. To create shapes with the **Shape** control, you first use the control to draw the image you want. Then, you use the Properties window to adjust the image characteristics.

The **Shape** control has properties similar to those of the **Line** control. Here's a list of the most important **Shape** control properties and the purpose of each:

Property	Purpose
Shape	Controls the shape of the image after you create it.
FillColor	Specifies the color of the shape.
FillStyle	Specifies a pattern for the fill color.
BorderColor	Specifies a separate color for the shape's border.
Visible	Hides or displays your artwork in your program as needed.

Embellishing a Splash Screen

This illustration shows how to use the **Line** and **Shape** controls to embellish a splash screen. The screen advertises a fictitious business named Northwest Window Screens. To view the illustration, click this icon: {ewc mvimg, mvimage,!illust.bmp}

In the illustration, note the following property settings:

® The line **BorderColor** property is set to dark blue.

® The line **BorderWidth** property is set to 5.

® Two shapes demonstrate the rectangle and rounded rectangle properties of the **Shape** control.

® In both shapes, the **FillStyle** is set to Cross.

In addition to the **Line** and **Shape** controls, Visual Basic also supports several graphics methods, with which you can add artwork to your programs. By using keywords in event procedures, you can create special visual effects on screen or on paper.

Advantages and Disadvantages

Graphics methods give you the advantage of creating visual effects (such as arcs and individually painted pixels) that you can't create by using the **Line** or **Shape** control. Here's a list of the most important graphics methods and the purpose of each:

Method	Purpose
Line	Creates a line, rectangle, or solid box.
Circle	Creates a circle, ellipse, or pie slice.
Pset	Sets the color of an individual pixel on the screen.

The disadvantages of using graphics methods include the considerable planning and programming time needed to use them. You need to learn the command syntax, understand the coordinate system used on your form, and refresh the images if they become covered by another window.

{ewc mvimg, mvimage,!tip.bmp}

For example, the following **Circle** statement draws a circle with a radius of 750 twips at (x, y) coordinates (1500, 1500) on a form:

```
Circle (1500, 1500), 750
```

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter describes how to create a special file in your project called a standard module. Standard modules contain code and variables that you want to be accessible from all the event or general procedures in your program. You'll find that standard modules are especially useful in larger programming projects. In this chapter, you will learn how to:

® Create a standard module for code you use often.

® Use public variables to share important information.

® Create your own Function and Sub procedures.

A standard module is a special .bas file that you can open in your project. This type of module defines public variables and procedures you want to access in all parts of your program. Creating a new standard module is easy. From the **Project** menu, click **Add Module**, and then double-click the Module icon in the **Add Module** dialog box.

When you create a new standard module, it appears immediately in the Code window, as this illustration shows. To view the illustration, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

You can declare public variables and procedures in the Code window. For more information, see Working with Public Variables and Creating General-Purpose Procedures.

This section includes the following topics:

® Module Files

® Module Property

Because a standard module contains no objects (only public variables and code), its sole property is **Name**.

To view an illustration showing how you can change the standard module name with the Properties window, click this icon
{ewc mvimg, mvimage,!illust.bmp}

Using the **Name** property creates an object name that distinguishes modules in the program code and in the Project window. Assigning your module a name makes your projects easier to manage.

Remember By convention, module names start with the prefix *bas*.

Visual Basic stores modules in their own folder in the Project window. The first standard module in a program is named Module1. However, when you save the module to disk, you can change the name with the **File** menu **Save Module1** As command.

Note All standard modules have the .bas file name extension.

To view an illustration showing a new standard module in the Project window, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

You can load standard modules into other projects, just as you do with forms. To load standard modules, start from the **Project** menu, and then click **Add File**. You can also remove standard modules from a project — first, highlight the module in the Project window. Then from the **Project** menu, click **Remove**.

If you want to change a variable in more than one event procedure, you need to use the **Public** keyword and create a public variable in a standard module. After you declare the variable, you can read it, change it, or display it in any procedure in your program.

For example, placing the following program statement in a standard module creates TotalSales, a public variable that you can use anywhere in your program.

```
Public TotalSales
```

When you declare a public variable in a standard module, the variable appears in the Declarations section (the only place you can declare a **Public** variable), as this illustration shows. To view the illustration, click this icon: {ewc mvimg, mvimage,!illust.bmp}

Public Variable Types

By default, public variables in standard modules are declared as variant data types. However, you can declare a public variable as a fundamental data type by using the As keyword and by indicating the data type.

For example, this statement declares a public string variable, LastName, in your program:

```
Public LastName As String
```

Demonstration: Enhanced Lucky Seven

This demonstration shows how you can use a public variable named Wins in a standard module to enhance the Lucky Seven program in Building a Simple Program Step by Step. To view the demonstration, click this icon: {ewc mvimg, mvimage,!democlip.bmp}

In addition to public variables, standard modules can contain general-purpose procedures. As with public variables, you can call general-purpose procedures from anywhere in the program.

General-Purpose vs. Event Procedures

General-purpose procedures aren't like event procedures. Event procedures are associated with a run-time event or an object you created with a toolbox control; general-purpose procedures aren't. In fact, general-purpose procedures are similar to built-in Visual Basic statements and functions — they're called by name, they can receive arguments, and each procedure performs a specific task.

Why Use General-Purpose Procedures?

When you use general-purpose procedures, you can associate an often-used routine with an easy-to-recognize name in a standard module. The resulting ease of use provides these benefits:

Eliminates repeated lines in the program code	You can define a procedure once, but your program can execute it any number of times.
Makes programs easier to read	It's easier to read and understand a program divided into a collection of smaller parts than it is a program that consists of one large chunk of code.
Simplifies program development	Programs separated into logical units are easier to design, write, and debug. Also, if you write programs in a group setting, you can exchange procedures and modules rather than entire programs.
Extends the Visual Basic language	Often, procedures can perform tasks that individual Visual Basic keywords can't accomplish.

In this section, you learn about two kinds of general-purpose procedures and how they can enhance your Visual Basic programs. This section includes the following topics:

A Function procedure is a group of statements that do meaningful work — typically processing text, handling input, or calculating a numeric value. You write Function procedures in a standard module, between the **Function** and **End Function** statements.

You execute (call) a program function by placing the function name and any required arguments in a program statement. (Arguments are the data that functions manipulate.) In other words, using a function procedure is exactly like using other built-in functions, such as **MsgBox**, **InputBox**, or **Time**.

This section includes the following topics:

® Function Syntax

® Function Calls

The basic syntax of a function is:

```
Function FunctionName([arguments]) [As Type]
    procedure statements
End Function
```

where

® *FunctionName* is the name of the Function procedure you are creating in the standard module.

® *arguments* is a list of optional arguments (separated by commas if there's more than one argument) that you use in the function.

® *As Type* is an option that specifies the variable type that the function returns (the default type is Variant).

® *procedure statements* is a block of statements that accomplish the work of the procedure.

The last statement in a function should always assign the final calculation of the function to *FunctionName*.

Example: TotalTax

The following Function procedure computes the state and city taxes for an item and then assigns the result to the **TotalTax** function name. To view this code sample, click this icon:
{ewc mvimg, mvimage,!code.bmp}

Note that the last statement in the body of the function assigns a final value to the **TotalTax** function name. This important convention is the Visual Basic method for returning information to the program.

In the previous topic, you used the **TotalTax** function to compute state and city taxes for an item and then assign the result to the TotalTax name. To call the **TotalTax** function in an event procedure, you would use a statement that looks like this:

```
lblTaxes.Caption = TotalTax(500)
```

This statement computes the total taxes required for a $500 item and then assigns the result to the **Caption** property of the **lblTaxes** object.

The **TotalTax** function can also take a variable as an argument. This line of code uses the **TotalTax** function to determine the taxes for the number in the SalesPrice variable and then adds them to SalesPrice to get the total cost of an item:

```
TotalCost = SalesPrice + TotalTax(SalesPrice)
```

If you want to practice using functions, complete the exercises in Review Lab 9.

A Sub procedure is similar to a Function procedure, with one exception — a Sub procedure doesn't return a value associated with its name. Also, most functions can return only one value, but Sub procedures can return many.

Typically, Visual Basic programmers use Sub procedures to:

® Obtain user input.

® Display or print information.

® Manipulate several properties associated with a condition.

® Process and return several variables during a procedure call.

This section includes the following topics:

The basic syntax for a Sub procedure looks like this:

```
Sub ProcedureName([arguments])
    procedure statements
End Sub
```

where

® *ProcedureName* is the name of the Sub procedure you're creating in the standard module.

® *arguments* is a list of optional arguments (separated by commas if there's more than one argument) to be used in the Sub.

® *procedure statements* is a block of statements that accomplish the work of the procedure.

In the procedure call, the number and type of arguments sent to the Sub procedure must match the number and type of arguments in the Sub declaration. Arguments can be passed to a procedure by reference or by value.

If variables passed by reference to a Sub are modified during the procedure, the updated variables are returned to the program.

Example: Adding Names to a List Box

The following Sub procedure adds names to a list box on a form at run time. The procedure receives one string variable passed by reference. If this Sub procedure is declared in a standard module, it can be called from any event procedure in the program.

```
Sub AddNameToListBox(person As String)
    Dim Msg As String
    If person <> "" Then
        Form1.List1.AddItem person
        Msg = person & " added to list box."
    Else
        Msg = "Name not specified."
    End If
    MsgBox Msg, , "Add Name"
End Sub
```

This audio narration describes the Sub procedure. To listen to the narration, click this icon:
{ewc mvimg, mvimage,!audio.bmp}

To call a Sub procedure in a program, you specify the name of the procedure, and then you list the arguments required by the Sub.

For example, to call the AddNameToListBox procedure by using a literal string (to call it by value), you could type the following statement:

```
AddNameToListBox "Kimberly"
```

Similarly, you could call the procedure by using a variable (call it by reference) by typing this statement:

```
AddNameToListBox NewName
```

In both cases, the AddNameToListBox procedure would add the specified name to the list box. In this Sub procedure, calls by value and calls by reference produce similar results because the argument in the procedure is declared by reference (the default) and isn't modified.

When you call the procedure many times, the space-saving advantages of a Sub procedure become clear. To view a code example, click this icon:
{ewc mvimg, mvimage,!code.bmp}

Notice that by using a **Do** loop, the user can add as many names to the list box as he or she would like. The exit condition in the **Do** loop is pressing the ENTER key without specifying a name in the **InputBox**.

Demonstration: Using a Sub Procedure

To view a demonstration showings how you can use a Sub procedure to manage the text in two list boxes, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

Note   Although most programmers write their general procedures in a standard module, you can write a Function or Sub procedure in a form module. However, if you do so, the general procedure can only be called from other procedures within its own form module.

---

1. **The scope of a local variable is:**

{ewc mvimg, mvimage,!answer.bmp} A. Limited to standard modules only.

{ewc mvimg, mvimage,!answer.bmp} B. Limited to the event procedure it is declared in.

{ewc mvimg, mvimage,!a C. Extended to all event procedures.

nswer.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. Unlimited.

2. Which of the following elements *cannot* be created in a standard module?

{ewc mvimg,mvimage,!answer.bmp} A. Public variable.

{ewc mvimg, B. Function procedure.

mvimage,!answer.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. Sub procedure.

{ewc mvimg, mvimage,!answer.bmp} D. Toolbox controls.

3. Public variables are created:

{ewc mvimg, mvimage, !answer.bmp} A. In event procedures.

{ewc mvimg, mvimage, !answer.bmp} B. With the **Public** keyword.

{ewc mvimg, mvimage, !an C. In Function procedures only.

swer.bmp}

{ewc mvimg,mvimage,!answer.bmp} D. With the **Variant** data type only.

4. What is an argument?

{ewc mvimg,mvimage,!answer.bmp} A. A variable or value passed to a Function or Sub procedure when the procedure is called.

{ewc mvimg,m B. The value returned to the program after a function call.

vimage,!answer.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. A keyword used when declaring public variables.

{ewc mvimg, mvimage,!answer.bmp} D. A local variable declared inside a Function or Sub procedure.

5. The following ComputeCost function is called in a program with the statement ComputeCost (500):

```
Function ComputeCost(num)
    SubTotal = num + 100
    ComputeCost = SubTotal / 2
End Function
```

What value does the function return?

{ewcmvimg,mvimage,!answer.bmp} A. 50.

{ewcmvimg,mvimage,!answer.bmp} B. 100.

{ewcmvimg,mvima C. 300.

g
e,
!
a
n
s
w
er
.b
m
p}

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

D. 500.

6. **The following statements are used to call the SquareIt procedure:**

```
Sum = 4
SquareIt Sum
```

And the SquareIt procedure looks like this:

```
Sub SquareIt(num)
    num = num * num
End Sub
```

After the procedure call, what is the value of the Sum variable?

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a

A. 4.

nswer.bmp}		
{ewc mvimg, mvimage, !answer.bmp}	B.	8.
{ewc mvimg, mvimage, !answer.bmp}	C.	16.
{ewc mvimg, mvim	D.	0.

age,!answer.bmp}

7. Arguments passed by reference to a procedure:

{ewc mvimg, mvimage,!answer.bmp} A. Must be enclosed in quotation marks.

{ewc mvimg, mvimage,!answer.bmp} B. Cannot be modified by the procedure call.

{ew C. Can be modified by the procedure and returned with their new values to the c mvimg,mvimage,!answer.bmp} calling statement.

{ewc mvimg,mvimage,!answer.bmp} D. Are passed by value.

In this lab, you will create a new version of the Lucky Seven slot machine that you explored in Chapter 1. You will:

® Use a function named **Rate** to determine the Lucky Seven slot machine win rate.

® Use a label to display the win rate, two public variables to record spins, and a standard module to hold the **Rate** function.

® Practice calling the Lab9.vbp **Rate** function from an event procedure.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **30 minutes**

Objectives

After completing this lab, you will be able to:

® Create a standard module in a project.

® Add public variables and a function procedure to a standard module.

® Call a function from an event procedure.

Lab Setup

In this lab, you load the Lab9.vbp project. If you installed *Learn Microsoft Visual Basic 6.0 Now* with the default settings, this file will be in the \LVB6\Ch09 folder on your hard drive. This project is identical to the Lucky.vbp project that you created in Chapter 1. (The name change preserves the original copy.)

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Create a Standard Module

In this exercise, you:

— Open the lab9.vbp project.

— Add a new label to the form.

— Create a standard module, Rate.bas.

® Exercise 2: Create the Rate Function

In this exercise, you write a function named **Rate**, which calculates the slot machine win rate and returns it to the calling procedure, and declare two public variables.

® Exercise 3: Call the Function

In this exercise, you call the **Rate** function from an event procedure.

In this exercise, you:

® Open the lab9.vbp project.

® Add a new label to the form.

® Create a standard module, Rate.bas.

u To open the project and add a label

1. Start Visual Basic and open the Lab9.vbp project.
2. In the Project window, click Lab9.frm. If it's necessary to display the form, click the **View Object** object.
3. Click the **Label** control, and then create a new rectangular label below the Lucky Seven label.
4. Set these properties for the new label:

Object	Property	Setting
Label5	**Alignment**	2 - Center
	Caption	"0.0%"
	Font	Arial, Bold Italic, 12-point
	ForeColor	Red
	Name	lblRate

5. From the **File** menu, click **Save Lab9.frm As**, and then save your new form to disk under the name **Rate.frm**.
6. From the **File** menu, click **Save Project As**, and then save your project under the name **Rate.vbp**.

u To use the Add Module command

1. From the **Project** menu, click **Add Module**, and then double-click the Module icon.
2. From the **File** menu, click **Save Module1 As**, type **Rate.bas**, and then press ENTER to save the new module to disk.

In this exercise, you write a function named **Rate**, which calculates the slot machine win rate and returns it to the calling procedure, and declare two public variables.

u To write the Rate function

1. In the Declarations section of the Rate.bas module's Code window, type the following public variable declarations:

```
Public Wins, Spins
```

2. Type the following Function procedure:

```
Function Rate(Hits, Attempts) As String
    Dim Percent
    Percent = Hits / Attempts
    Rate = Format(Percent, "0.0%")
```

When you begin typing the **Rate** function, Visual Basic places it in a new section in the Code window and adds **End Function** to the bottom of the procedure.

Note: If you paste this function declaration into the standard module, you'll need to add an **End Function** statement to the bottom of the routine yourself.

---

3. To save your completed standard module, click **Save Project**.
4. Close the Code window.

In this exercise, you call the **Rate** function from an event procedure.

u To edit the Command1_Click event procedure

1. On the form, double-click **Spin** to open the Command1_Click event procedure.
2. Below the fourth line of the event procedure (the third statement containing the **Rnd** function), type this statement:

   ```
   Spins = Spins + 1
   ```

   Each time the user clicks **Spin**, the statement increments the Spins public variable, and new numbers appear in the Spin windows.
3. Scroll down the Code window.
4. In the **If…Then** decision structure, under the **Beep** statement, type the following statement:

   ```
   Wins = Wins + 1
   ```

   Each time a seven appears in a spin window, the statement increments the Wins public variable.
5. Type the following line between the **End If** and **End Sub** statements at the bottom of the Command1_Click event procedure:

   ```
   lblRate.Caption = Rate(Wins, Spins)
   ```

   This statement calls the **Rate** function to determine the win percentage and displays the result with the lblRate label.

u To run the program

1. Click **Start**.
2. Click **Spin** 10 times.

   For the first five spins, the rate stays at 100%—you're hitting the jackpot every time. As you continue to click, however, the win rate adjusts to 83.8%, 71.4%, 75.0% (another win), 66.7%, and 60% (a total of 6 for 10).
3. Continue clicking **Spin** until the rate hits about 28% (the statistical average win rate), and then stop.
4. To stop the program, click **End**.

To help you avoid accidentally mixing up your variables, Visual Basic limits the scope of variables to the event procedure they were declared in. Variables with this limited scope are local to event procedures and do not maintain their values in other event procedures. For example, if you create a TotalSales variable in the cmdSale_Click event procedure, you can't add numbers to it in any other event procedure.

Variables that maintain their values in all event procedures are known as public variables. These variables let you share important values, such as a running total of expenses or a user's name and phone number, everywhere in a program.

To view an animation that describes the relationship between local and public variables in a program, click this icon:
{ewc mvimg, mvimage,!anim.bmp}

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter describes how you can open and modify text files and databases on your system. In this chapter, you will learn how to:

® Display a text file with the **TextBox** control.

® Create a new text file.

® Open a database with the **Data** control.

® Search, add, and delete database records.

If you regularly work with databases, you should consider using Visual Basic to enhance and display your data. Visual Basic is a powerful tool that implements the same database engine that is included with Microsoft Access. With just a few dozen lines of program code, you can use Visual Basic to create efficient and fully customized database applications. This section includes the following topics:

Most objects you create with the Visual Basic toolbox controls have the built-in ability to display information from a database table. In database terminology, these objects are called bound controls.

You can use these standard toolbox controls to create objects that display database information:

® **CheckBox**

® **ComboBox**

® **Image**

® **Label**

® **ListBox**

® **PictureBox**

® **TextBox**

After you create the bound control, you can connect it to your data object by specifying the following properties:

Property	Setting	Purpose
DataSource	Data object name	Select a data object (such as Data1) from the list.
DataField	Field name	Select the database field you want to display from the list.

This demonstration uses one data object and four text boxes to display four database fields from the Biblio.mdb database. The demonstration shows how you can create a custom database application (front end) to view only the information you want to display. In this application, the **ReadOnly** property of the data object is set to **True**, so the content of the database cannot be accidentally (or intentionally) altered. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

As you learned in Displaying Data, you can open a database in Visual Basic by using the **Data** control and setting several of its properties. This table lists the name and purpose of four **Data** control property settings:

Property	**Setting**	**Purpose**
Connect	Database product name	Select Access, dBASE, FoxPro, or Paradox from the list.
DatabaseName	File name	Select the path and file name of the database you want to open.
RecordSource	Table name	Select the database table you want to use from the list.
Caption	Text description	Display the database filename or purpose.

1. **What does the following program statement do?**

```
Open CommonDialog1.Filename For Output As #1
```

{ewc mvimg, mvimage,!answer.bmp}

A. It opens the **Save As** common dialog box to store the output of your program.

{ewc mvimg, mvimage,!answer.bmp}

B. It uses a file name from the common dialog box to create a new file on disk.

{ewc mvimg, mvimage,

C. It opens an existing file on disk, and prepares the common dialog box to receive output from the file.

!answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} D. It assigns a file number to the CommonDialog1 dialog box.

2. What is a sequential file?

{ewc mvimg, mvimage, !answer.bmp} A. A text file that is opened with the **Output** statement and with either the **Input** or **Output** keywords.

{ewc mvi B. A file created by a database program such as Microsoft Access, dBASE, or Paradox.

mg, mvimage, !answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} C. A document file that contains formatting codes.

{ewc mvimg, mvimage, !answer.bmp} D. An executable file that contains coded instructions that the operating system executes sequentially.

3. Which control would be best for holding text in a simple text editor program?

{ewc mvimg, mvimage, !answer.bmp} A. **InputBox**.

{ewc mvimg, mvimage, !answer.bmp} B. **Image**.

{ewc mvimg, mvimage, ! C. **Data**.

answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} D. **TextBox**.

4. For what purpose would you use the CommonDialog control's Filter property?

{ewc mvimg, mvimage, !answer.bmp} A. To test for the end of a file.

{ewc mvim B. To close a sequential file.

g, mvimage, !answer.bmp}

{ewc mvimg, mvimage, !answer.bmp}

C. To control the type of files that are listed in a common dialog box.

{ewc mvimg, mvimage, !answer.bmp}

D. To configure a data object that displays the source of records in your database.

5. What is the purpose of the Data control’s ReadOnly property when it's set to True?

{ewc mvimg,mvimage,!answer.bmp} A. To prohibit users from opening databases they do not have permission to use.

{ewc mvimg,mvimage,!answer.bmp} B. To set the database type to Microsoft Access.

{ewc mvimg,mvimage,!an C. To allow users to view but not modify database information.

s
w
er
.b
m
p}

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

D. To allow users to both view and modify database information.

6. Which of the following is *not* a bound control?

{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

A. **CommandButton**.

{e
w
c
m
vi
m
g,
m

B. **CheckBox**.

vimage,!answer.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. **Image**.

{ewc mvimg, mvimage,!answer.bmp} D. **TextBox**.

7. What is the purpose of the Recordset object’s Index property?

{ewc mvimg, mvimage, !answer.bmp}	A.	Set to True if no match is found in a database search.
{ewc mvimg, mvimage, !answer.bmp}	B.	Defines the database field that will be used in a database search.
{ewc mvimg, mvimage, !ans	C.	Contains the current record visible in the database. (For example, record 23 of 100.)

w
er
.b
m
p}
{e
w
c
m
vi
m
g,
m
vi
m
a
g
e,
!
a
n
s
w
er
.b
m
p}

D. Sets the focus (the object containing the cursor) on the form.

In this lab, you first open a form that has been created as a database front end, and then you add functionality to it. The database is Biblio.mdb, a Microsoft Access file that contains a bibliography of database programming books. Next, you use the **Data** control to open the database. Finally, you add command buttons and program code to search, add, and delete database records.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **40 minutes**

Objectives

After completing this lab, you will be able to:

® Open a database with the **Data** control.

® Search database records.

® Add a database record.

® Delete a database record.

Lab Setup

In this lab, you need to load the Lab10.vbp project from the \LVB6\Ch10 folder on your hard drive. When you open the database, you will use Biblio.mdb, located in the same folder.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Open the Biblio.mdb Database

In this exercise, you open the Lab10.vbp project and set the required properties for the data object and the four bound text boxes on the form.

® Exercise 2: Search Database Records

In this exercise, you create a **Find** command button and write an event procedure that searches for a particular entry in the Title field.

® Exercise 3: Add a Record

In this exercise, you create an **Add** command button and write an event procedure that adds a new record to the database.

® Exercise 4: Delete a Record

In this exercise, you:

— Create a **Delete** command button.

— Write an event procedure that deletes an unwanted record from the database.

— Run the finished program.

In this exercise, you open the Lab10.vbp project and set the required properties for the data object and the four bound text boxes on the form.

u To open the Data Browser form

1. Start Visual Basic and open the Lab10.vbp project.
2. In the Project window, click **Lab10.frm**. If the form isn't visible, display it by clicking **View Object**.

u To set data object and bound text box properties

Use this table to set these data object and bound text box properties. (The **Name** properties listed have already been set, but review them now to become familiar with them.)

Object	Property	Setting
datBiblio	**Caption**	"Biblio.mdb"
	Connect	Access
	DatabaseName	"<*drive*>\Lvb6\ch10\ biblio.mdb"
	Name	datBiblio
	ReadOnly	True
	RecordsetType	0 - Table
	RecordSource	Titles
txtTitle	**DataField**	Title
	DataSource	datBiblio
	Name	txtTitle
txtInfo	**DataField**	Description
	DataSource	datBiblio
	Name	txtInfo
txtISBN	**DataField**	ISBN
	DataSource	datBiblio
	Name	txtISBN
txtYear	**DataField**	Year Published
	DataSource	datBiblio
	Name	txtYear

u To save the project to disk

1. From the **File** menu, click **Save Lab10.frm As**, and then save your new form to disk under the name **MyDataView.frm**.
2. From the **File** menu, click **Save Project As**, and then save your project under the name **MyDataView.vbp**.

In this exercise, you create a **Find** command button and write an event procedure that searches for a particular entry in the Title field.

u To create a Find button

1. Click the **CommandButton** control in the toolbox, and then create a command button object on the lower left side of the form.
2. Set the following properties for the command button:

Object	**Property**	**Setting**
Command1	Caption	“Find”
	Name	cmdFind

3. Double-click the **Find** button object on the form to open the cmdFind_Click event procedure.
4. Type the following code in the event procedure:

```
Dim prompt As String
Dim SearchStr As String
prompt = "Enter the full (complete) book title."
'get the string to be used in the Title field search
SearchStr = InputBox(prompt, "Book Search")
datBiblio.Recordset.Index = "Title"      'use Title table
datBiblio.Recordset.Seek "=", SearchStr 'and search
If datBiblio.Recordset.NoMatch Then      'if no match
    MsgBox "Sorry, I couldn’t find your book."
    datBiblio.Recordset.MoveFirst        'go to first record
End If
```

5. Close the Code window.

In this exercise, you create an **Add** command button and write an event procedure that adds a new record to the database.

u **To create an Add command button**

1. On the form, click **datBiblio** (the data object).
2. In the Properties window, find the datBiblio **ReadOnly** property and set it to **False**.

Note The **ReadOnly** property determines how the Biblio.mdb database is opened. By setting this property to **False**, you can modify the database and add new records.

3. Click the **CommandButton** control in the toolbox, and then create a command button to the right of the **Find** button on the form.
4. Set the following command button properties:

Object	Property	Setting
Command1	**Caption**	"Add"
	Name	cmdAdd

5. Double-click the **Add** button object on the form to open the cmdAdd_Click event procedure.
6. Type the following program statements in the event procedure:

```
Dim prompt As String
Dim reply As Integer
prompt = "Enter new record, and click left arrow button."
reply = MsgBox(prompt, vbOKCancel, "Add Record")
If reply = vbOK Then                'if the user clicks OK
    txtTitle.SetFocus               'move cursor to Title box
    datBiblio.Recordset.AddNew      'and get new record
    'set PubID field to 14          (this field is required
    datBiblio.Recordset.PubID = 14  'by Biblio.mdb)
End If
```

7. Close the Code window.

In this exercise, you:

® Create a **Delete** command button.

® Write an event procedure that deletes an unwanted record from the database.

® Run the finished program.

u To create a Delete button

1. In the Visual Basic toolbox, click the **CommandButton** control, and then create a command button to the right of the **Add** button on the form.
2. Set the following properties for the new button:

Object	Property	Setting
Command1	**Caption**	"Delete"
	Name	cmdDelete

3. On the form, double-click the **Delete** button object to open the cmdDelete_Click event procedure.
4. Type the following program statements:

```
Dim prompt As String
Dim reply As Integer
prompt = "Do you really want to delete this record?"
reply = MsgBox(prompt, vbOKCancel, "Delete Record")
If reply = vbOK Then              'if the user clicks OK
    datBiblio.Recordset.Delete    'delete current record
    datBiblio.Recordset.MoveFirst 'move to first record
End If
```

5. Close the Code window.
6. To save your changes click **Save Project**.

u To run the program

1. To run your database front end, click **Start**.
2. On the form, click **Find** and search for the book title, *Using SQL*.
3. Click **Find** again and search for a book that does not exist in the database (like *Pez*).
4. Click **Add**, and then type a new database record. To enter the new record, click the outside left arrow button on the data object.
5. Display the new record you just entered, and then click **Delete** to remove it.
6. When you're finished experimenting, click **Quit** to close the program.

The simplest way to display a text file in a program is to use a text box object. You can create text box objects in a variety of sizes; and, if the contents of the text file don't fit neatly in the text box, you can add scroll bars to the text box so that the user can examine the entire file.

To load the contents of a text file into a text box, you need to use three statements and one function. These keywords are described in the following table:

Keyword	Description
Open	Opens a text file for input or output
Line Input #	Reads a line of input from the text file
EOF	Checks for the end of the text file
Close	Closes the text file

This section includes the following topics:

® Opening Text Files

® The Open Statement

® Demonstration: Text Browser Program

In Visual Basic, you can use a text box to open text files, but not document files or executable files. Typical text files on your system will be identified by Windows Explorer as Text Documents or will have one of these filename extensions:

® .TXT

® .INI

® .DAT

® .LOG

® .BAT

To let the user decide which text file to open, prompt the user for the file name by using an ActiveX **CommonDialog** control. The file name that the user selects in the dialog box is returned to the program in the **FileName** property. You use this property and the **Open** keyword to open the file.

After you get the file name from the common dialog object, you open the file in the program by using the **Open** statement. The syntax for the **Open** statement looks like this:

```
Open pathname For mode As #filenumber
```

where:

® *pathname* is a valid Windows file name.

® *mode* is a keyword indicating how the file will be used (**Input** for reading the file, **Output** for writing the file, and **Append** for adding to the file).

® *filenumber* is an integer from 1 through 511, a number that is used to identify the file in program code.

An **Open** statement using the common dialog box's **FileName** property looks like this:

```
Open CommonDialog1.Filename For Input As #1
```

Notice that in the statement:

® The **Filename** property of the common dialog object holds the complete path and file name.

® **Input** is the mode.

® 1 is the file number.

Note Text files opened with **Open** and with the **Input** or **Output** keywords are called *sequential files.* This is because the file contents must be worked with in sequence. By contrast, you can access the information in a database file in any order.

A full-featured program that opens and displays text files should include these features:

® Menu commands.

® A common dialog box.

® The **Open**, **Line Input**, **EOF**, and **Close** keywords in program code.

This window shows the sample code for a complete text browser program that uses **Open** and **Close** menu commands to open and close files. To view the code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

To view a demonstration showing the finished text browser, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

Now and then, you may want to create a new text file with Visual Basic. For example, you might need to create a custom report or transaction log, save important calculations or values, or create a special-purpose word processor or text editor.

This section includes the following topics:

The most important code in a basic text editor is the code associated with the **Save** command. When the user clicks **Save** on the **File** menu, your program should:

® Prompt the user for a file name.

® Open a new file for output with the **Output** keyword.

® Save the text to disk with the **Print #** statement.

If the user clicks **Cancel** instead of specifying a file name in the **Save As** dialog box, your Save routine should terminate gracefully.

This window shows code you can use to save text to a file that is stored in a text box named **txtNote**. The code is located in the mnuItemSave_Click event procedure. This is the procedure that runs when the user clicks the **File** menu **Save** command. To view the code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

This demonstration shows how you can build a simple note-taking utility. You can use this tool to take notes at home or at work and then save them to disk. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

Here's how to create that new text file from within your program.

u **To create a new text file**

1. Prompt the user for input, if necessary, and perform any required calculations.
2. Assign the results of your processing to one or more variables.

   For example, you could assign the contents of a text box to a variable named InputForFile.
3. With the **CommonDialog** control **SaveAs** method, prompt the user for a file name.
4. Use the file name to open the file for output (Open...For Output).
5. To send output to the open file, use the **Print #** statement.
6. When you finish, use the **Close** keyword to close the file.

You can create a simple database front end by using the **Data** control and a few bound objects. But if you want to perform more sophisticated database operations like searching, adding, and deleting records, you need to use program code.

The primary mechanism for manipulating a database programmatically is the **Recordset** object. The **Recordset** object includes special properties and methods that let you search, add, and delete records.

This section includes the following topics:

The **Recordset** object has two properties and two methods that you can use together to locate a particular record in the database. These keywords are defined in the following table:

Recordset Property or Method	**Description**
Index	A property used to define the database field that will be used for the search.
Seek	A method used to search for the record. In addition to =, the relational operators >=, >, <=, and < can be used to compare the search string to text in the database.
NoMatch	A property set to True if no match is found in the search.
MoveFirst	A method that makes the first record in the Recordset the current record.

To view sample code showing how you can use the **Recordset** object to search for an database item in a field named Title, click this icon:
{ewc mvimg, mvimage,!code.bmp}

The previous routine displays a search dialog box to get a search string (SearchStr) from the user. The procedure uses the **Seek** method to search the Title field from beginning to end until it finds a match or reaches the end of the list. If no match is found, the **MsgBox** function displays a sympathetic message, and the **MoveFirst** method opens the first record in the database.

To add a new record to the database, you set the data object’s **ReadOnly** property to False in design mode, and then you use the Recordset’s **AddNew** method to open a new record. The user completes the transaction by filling in the necessary fields and clicking one of the navigation buttons on the data object.

To view sample code showing how you can use the **ReadOnly** property and the **AddNew** method to add a new record, click this icon:
{ewc mvimg, mvimage,!code.bmp}

In the previous routine, a **MsgBox** function displays entry instructions and waits for the user to click the **OK** button to add a record. The PubID field, which identifies book publishers by using a special code, is a unique requirement of the Biblio.mdb database.

To delete a record from a database, you display the record you want to delete and then you use the Recordset object’s **Delete** method to remove the record.

After you delete the record, you should display a new record in the database (the data object doesn’t display one automatically). The best technique is to use the **MoveFirst** method to display the first record in the database.

{ewc mvimg, mvimage,!tip.bmp}

To view sample code showing how you can use the **Delete** method to delete records from a database, click this icon:
{ewc mvimg, mvimage,!code.bmp}

Pay particular attention to the use of the **MsgBox** function in the previous routine. Because the data object doesn’t provide an Undo feature, it's important that you verify the user’s intentions before you use the **Delete** method to permanently delete a record.

If you're like most people, the information you keep in databases is very important, and you'd be in bad shape if something happened to it. For this reason, you should always make a backup copy of each database you use before making changes to it. If anything goes wrong when you're working with the copy, you can simply replace the copy with the original database.

Creating a Copy

Your backup routine might include using a backup program, Windows Explorer, or a special backup feature in your database program. As an additional safeguard, you can create a backup copy of one or more files from within a Visual Basic program by using the **FileCopy** statement. **FileCopy** makes a separate copy of the file (as Windows Explorer's Edit Copy command does) when you use the statement with the following syntax:

```
FileCopy source destination
```

where:

® *source* = the path and file name of the file you want to copy.

® *destination* = the path and file name of the file you want to create.

Using the FileCopy Statement

This code window shows how you can use the **FileCopy** statement to make a backup copy of your database. The code should be placed in the Form_Load event procedure, so that it is executed before the database is opened. To view the sample code, click this icon:
{ewc mvimg, mvimage,!code.bmp}

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

This chapter describes how you can explore the application objects in your system, and control Microsoft Office applications with Automation. In this chapter, you will learn how to:

® Understand and use Automation principles in your programs.

® Use the Object Browser to view application objects.

® Control Microsoft Office applications from your programs.

1. What is Automation?

{ewc mvimg, mvimage,!answer.bmp}

A. An industry-standard communication protocol that allows one program to use the features of another.

{ewc mvimg, mvimage,!answe

B. A remote communication technology specifically designed for the Internet.

r.bmp}

{ewc mvimg, mvimage, !answer.bmp}

C. A set of standards that allows Visual Basic programmers to use the commands in any Windows-based application on their system.

{ewc mvimg, mvimage, !

D. A technology that allows Visual Basic programmers to use the features of Windows-based applications that they don't currently have on their system.

answer.bmp}

2. Which Visual Basic tool would you use to explore the objects, properties, and methods exposed by Microsoft Excel?

{ewc mvimg, mvimage,!answer.bmp} A. Project Explorer

{ewc mvimg, B. Windows Explorer.

=mvimage,!answer.bmp}

{ewc =mvimg, =mvimage,!answer.bmp} C. Properties window.

{ewc =m D. Object Browser.

vimg, mvimage, !answer.bmp}

3. What Visual Basic command would you use to add an application object library to your program?

{ewc mvimg, mvimage, !answer.bmp

A. The **References** command on the **Project** menu.

}

{ewc mvimg, mvimage, !answer.bmp}

B. The **Components** command on the **Project** menu.

{ewc mvimg, mvimage, !answe

C. The **Object Browser** command on the **View** menu.

r.bmp}

{ewc mvimg, mvimage,!answer.bmp} D. The **Project Explorer** command on the **View** menu.

4. What does the following program statement do in a program?

```
Set X = CreateObject("Word.Application")
```

{ewc mvimg, mvi A. Declares X as a new object variable in your program.

mage, !answer.bmp}

{ewc mvimg, mvimage, !answer.bmp}

B. Copies text to Microsoft Word.

{ewc mvimg, mvimage, !answer.bmp}

C. Prepares your program for Automation by assigning the Word application variable to an object variable in your program.

{ewc mvimg

._mvimage.!answer.bmp}

{ewc_mvimg._mvimage.!answer.bmp} D. Releases the object variable in your program that holds the Word application object.

5. What is Visual Basic for Applications?

{ A. A version of Visual Basic designed especially for application macros.

ewc mvimg, mvimage, !answer.bmp}

{ewc mvimg, mvimage, !answer.b

B. A programming language featured in all Microsoft Office 97 applications.

mp}

{ewc_mvimg,_mvimage,!answer.bmp} C. A programming system that supports Automation.

{ewc_mvimg,_mvimage,!ans D. All of the above.

wer.bmp}

6. What Automation method would you use to open a new document in Microsoft Word?

{ewc mvimg. mvimage.!answer.bmp} A. **Documents.Add**

{ewc mvimg. mvi B. **Selection.Text**

mage,!answer.bmp}

{ewc mvimg, mvimage,!answer.bmp} C. **Word.Application**

{ewc mvimg D. **Dim X As Object**

.=mvimage.!answer.bmp}

7. **X is an object variable that holds an Office 97 Application object. What does the following program statement do?**

```
X.Visible = False
```

{ewc=mvimg.=mvimage.!answer.bmp} A. Closes the application.

{ewc mvimg, mvimage, !answer.bmp} B. Displays the application.

{ewc mvimg, mvimage, !answer C. Opens a new document.

.bmp}

{ewc mvimg, mvimage, !answer.bmp} D. Hides the application.

In this lab, you will create a front end application for a music reviewer who wants to make notes about the artists she evaluates at live performances. This front end will open a Microsoft Access database named Talent.mdb and display the Name, Address, City, and State fields for local bands. It will also contain a Comments field that the music reviewer can spell check with Word Automation.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **40 minutes**

Objectives

After completing this lab, you will be able to:

® Create the front end for a music database.

® Use Word Automation to spell check review notes.

Lab Setup

In this lab you will load the Lab11.vbp project from the \LVB6\Ch11 folder on your hard drive. The basic database front end has been designed for you, but you need to set the properties and type the program code. The Talent.mdb database that you will open is located in the same folder on your hard drive.

To complete the exercises in this lab, you must have the required software. For detailed information about the labs and setup for the labs, see Labs in this course.

Exercises

® Exercise 1: Set Data Object Properties

In this exercise, you:

— Open the Lab11.vbp file.

— Connect to the Talent.mdb database.

— Set properties for the bound objects on the form.

® Exercise 2: Write the Automation Code

In this exercise, you write an event procedure that uses Word Automation to check the spelling of the Comments field in the database.

® Exercise 3: Run the Program

In this exercise, you run the database front end and test the Automation routine.

In this exercise, you will:

® Open the Lab11.vbp project.

® Connect to the Talent.mdb database.

® Set properties for the bound objects on the form.

u To open the project

1. Start Visual Basic and open the Lab11.vbp project.
2. In the Project window, click Lab11.frm. If necessary, display the form by clicking the **View Object** button.

u To set the bound object properties

Set the following properties for the bound objects on the form:

Object	Property	Setting
Data1	**Caption**	"Talent"
	DatabaseName	*<drive>*\Lvb6\ch11\talent.mdb
	RecordSource	Artists
Text1	**DataSource**	Data1
	DataField	Name
Text2	**DataSource**	Data1
	DataField	Address
Text3	**DataSource**	Data1
	DataField	City
Text4	**DataSource**	Data1
	DataField	State
Text5	**DataSource**	Data1
	DataField	Comments

u To save the project to disk

1. From the **File** menu, click **Save Lab11.frm As**, and then save your new form to disk under the name **Music.frm**.
2. From the **File** menu, click **Save Project As**, and then save your project as **Music.vbp**.

In this exercise, you write an event procedure that uses Microsoft Word Automation to check the spelling of the Comments field in the database.

u **To write the program code**

1. From the **Project** menu, click **References**.
2. To include the Microsoft Word 8.0 Object Library in your project, make sure that there's an "x" in the check box next to its listing, and then click **OK**.
3. Double-click the **Check Spelling** button, and then type the following program code in the Code window:

```
Dim X As Object                 'create Word object
Set X = CreateObject("Word.Application")

X.Visible = False               'hide Word
X.Documents.Add                 'open a new document
X.Selection.Text = Text5.Text   'copy text box to document
X.ActiveDocument.CheckGrammar   'run spell/grammar checker
Text5.Text = X.Selection.Text   'copy corrected text to VB
X.ActiveDocument.Close SaveChanges:= wdDoNotSaveChanges
X.Application.Quit              'quit Word
Set X = Nothing                 'release object variable
```

Note This code requires that you have Microsoft Word 97 on your hard disk. If you don't have Word, you won't be able to add the spell check feature, but you can still create the database front end.

4. To save your completed event procedure, click **Save**.
5. Close the Code window.

In this exercise, you run the database front end and test the Automation routine.

u To start up the database front end

1. Click **Start**.
2. Scroll through the Talent database records and verify that information appears in each of the five text boxes.
3. To evaluate the spelling in the first database record’s Comments field, click **Check Spelling**.
4. Correct any misspelled words in the field, and then watch to see if the Automation code copies the corrections back to your form.
5. When you're finished checking spelling, click **Quit** to stop the program.

In this chapter, you'll learn how to use a technology called Automation. You can use this technology to incorporate the functionality of Windows-based applications into your program code. Windows-based applications that fully support Automation make available (expose) their application features as a collection of objects with associated properties, methods, and events.

The Windows-based applications that expose their objects are called object or server applications. Programs that use these objects are called controlling or client applications.

A collection of Automation protocols (previously known as OLE Automation) links these server and client applications. For Automation to work, you must own a copy of the application you want to communicate with.

To view an expert point of view introducing key Automation concepts, click this icon.
{ewc mvimg, mvimage,!exppov.bmp}

The Visual Basic Object Browser is a viewing utility that has two uses:

® It displays the objects, properties, and methods used by a Visual Basic program.

® It displays the objects, properties, and methods exposed by an application on your system that supports Automation.

For example, you can use the Object Browser to examine the application objects provided by these applications in the Microsoft Office software suite:

® Microsoft Excel

® Microsoft Word

® Microsoft PowerPoint

® Microsoft Access

® Microsoft Outlook

Visual Basic organizes the features of each application program by class in a master object library. Within each class, the Object Browser lists the properties, events, and methods exposed by the application — the commands you can use to enhance your Visual Basic programs.

Browsing an Object Library

u **To browse an object library and explore the features of a Microsoft Office application**

1. From the **Project** menu, click **References**.
2. Pick the application object library you want to browse for Automation objects.
3. From the **View** menu, click **Object Browser**.
4. Click the first drop-down list box, and then select the object library you want to examine.

   The library you added in Step 2 now appears in the list.
5. In the **Classes** list box, click the object you want to examine.
6. In the **Members** list box, scroll through the properties, methods, and events.
7. To examine the syntax for an item in the **Members** list box, click it, and then read the description at the bottom of the window.
8. To see a complete online help description of an item, select it and click the **Question (?)** button at the top of the Object Browser window.
9. When you're finished, click **Project** on the **Window** menu to return to your project.

This illustration shows the **CheckSpelling** method located in the **Worksheet** class of the Microsoft Excel 8.0 object library. As you can see, Microsoft Excel has hundreds of features just waiting to be used. To view the illustration, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

Each application that supports Automation provides a different collection of objects and a different set of calling conventions. However, the Microsoft Office 97 software suite provides a basic object model that is consistent from application to application. In this section, you'll learn how to use Microsoft Word, Microsoft Excel, and Microsoft Outlook via Automation.

This section includes the following topics:

Microsoft Word 97 features the new Visual Basic for Applications programming language that you can use to write sophisticated word processing utilities. If you have Microsoft Word on your system, you can also use the features of Word immediately through Automation.

u To add a reference to the Microsoft Word object library

1. From the **Project** menu, click the **References** command and select the Microsoft Word 8.0 Object Library.
2. Examine the features you want to use (as necessary) with the Object Browser.

This sample code demonstrates the **CheckGrammar** method, a command that runs the Microsoft Word 97 grammar and spelling checker. The programming steps apply to most application objects, so you can use them as guidelines to incorporate the functionality of most applications that support Automation.

u To write your Visual Basic program

1. In the event procedure in which you plan to use Automation, use the **Dim** statement to create an object variable:

   ```
   Dim X As Object                'use X as variable name
   ```

2. Use the **CreateObject** function to load an **Automation** object into the object variable:

   ```
   Set X = CreateObject("Word.Application")
   ```

3. Use the methods and properties of the **Application** object in the event procedure.

   ```
   X.Visible = False              'hide Word
   X.Documents.Add                'open a new document
   X.Selection.Text = Text1.Text  'copy text box to document
   X.ActiveDocument.CheckGrammar  'run spell/grammar checker
   Text1.Text = X.Selection.Text  'copy results back
   ```

   Consult the online Visual Basic Help files in the Object Browser or the product documentation for the proper syntax.

4. When you've finished using the object application, use this program code to close the document, quit Word, and release the object variable to conserve memory:

   ```
   X.ActiveDocument.Close SaveChanges:=wdDoNotSaveChanges 'close
   X.Quit                           'quit Word
   Set X = Nothing                  'release object variable
   ```

This demonstration shows you how to use Word to check the spelling and grammar of sentences you type in a Visual Basic text box. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

If you have Microsoft Excel on your system, you can use Automation to access any of its powerful number crunching and data analysis features. The procedure is very similar to the one detailed in the preceding section for Microsoft Word.

u To add a reference to the Microsoft Excel object library

1. From the **Project** menu, click the **References** command and select the Microsoft Excel 8.0 Object Library.
2. Examine the features you want to use (as necessary) with the Object Browser.

This sample code demonstrates how you can use Excel's **Pmt** function to calculate a mortgage payment when you know the interest rate, term, and loan amount (or principal). This sample code assumes that you have already built a form with three text boxes for the interest rate (**Text1**), the term in months (**Text2**), and the principal (**Text3**). The result is calculated via a **Calculate** button (Command1) that displays the periodic payment with a message box. (The output is formatted with the **Format** and the **Abs** functions.)

u To write your Visual Basic program

1. In the event procedure in which you plan to use Automation, use the **Dim** statement to create an object variable and declare other variables:

```
Dim xl As Object                'use out as variable name
Dim loanpmt                     'declare return value
```

2. Test the values you will use in the calculation (to verify that they are not null or otherwise out of range) and then load the **Automation** object with the **Create Object** function and call the methods and properties you need in Excel:

```
If Text1.Text <> "" And Text2.Text <> "" _
And Text3.Text <> "" Then  'create object and call Pmt
    Set xl = CreateObject("Excel.Sheet")
    loanpmt = xl.application.WorksheetFunction.Pmt _
        (Text1.Text / 12, Text2.Text, Text3.Text)
    MsgBox "The monthly payment is " & _
        Format(Abs(loanpmt), "$#.##"), , "Mortgage"
    xl.application.quit
    Set xl = Nothing
Else
    MsgBox "All fields required", , "Mortgage"
End If
```

To see the results of running the source code testing **Pmt**, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

To view an introduction to this course, click this icon.
{ewc mvimg, mvimage,!exppov.bmp}

Visual Basic is an extremely flexible programming product designed for a variety of applications. Students, managers, and people in various technical fields use Visual Basic to learn how to write practical, Windows-based programs; business professionals use Visual Basic to write macros that leverage the documents and capabilities of their Microsoft Office applications; and experienced software developers use Visual Basic to build powerful commercial applications and corporate productivity tools.

Whether you fit into one of these categories or you are just curious about programming, Visual Basic 6.0 has features designed specifically for you. When you complete this programming course, you will have the skills necessary to write useful and interesting applications for the Windows and Windows NT operating systems.

Michael Halvorson

Michael Halvorson worked for Microsoft Corporation from 1985 to 1993, where he was employed as a technical editor, acquisitions editor, and localization manager. He received a B.A. in Computer Science from Pacific Lutheran University, and an M.A. in History from the University of Washington.

Michael is the author of *Microsoft Visual Basic 6 Step by Step*, co-author (with Chris Kinata) of *Microsoft Word 97/Visual Basic Step by Step*, and co-author (with Michael Young) of *Running Microsoft Office 97*, all published by Microsoft Press.

{ewc mvimg, mvimage,!exppov.bmp}	You Did It!

Learning Resources

The following learning resources will help you take your next steps with Visual Basic:

® To learn intermediate and advanced Visual Basic topics in a multimedia programming course that looks like this course, try *Mastering Microsoft Visual Basic 6* (Microsoft Corporation, 1998).

® For information about using Visual Basic for Applications to manage documents and access the Internet, purchase *Microsoft Word 97/Visual Basic Step by Step* (Microsoft Press, 1997), by Michael Halvorson and Chris Kinata, ISBN 1-57231-388-9.

® If you like to learn by using sample code and running real-world applications, I recommend *Microsoft Visual Basic 6.0 Developer's Workshop, Fifth Edition* (Microsoft Press, 1998), by John Clark Craig and Jeff Webb, ISBN 1-57231-883-X.

® For advanced information about professional programming techniques and using Windows API functions, try *Hardcore Visual Basic (5.0), Second Edition* (Microsoft Press, 1997), by Bruce McKinney, ISBN 1-57231-422-2 or *Advanced Microsoft Visual Basic (6.0), Second Edition* (Microsoft Press, 1998), by The Mandelbrot Set, ISBN 1-57231-893-7.

® For a comprehensive discussion of professional data access techniques using Visual Basic, see *Hitchhiker's Guide to Visual Basic and SQL Server, Sixth Edition* (Microsoft Press, 1998), by William R. Vaughn, ISBN 1-57231-848-1.

Best of luck!

{ewc mvimg, mvimage,!tip.bmp}

To view the Microsoft Press developer product Web site, click this icon.
{ewc mvimg, mvimage,!intjump.bmp}

Even before you know *how* the program will do what you want it to, you should have a good idea of *what* it should do and how it should look. You might find it useful to draw a rough sketch of the user interface you'd like to present to your users. Doing this will not only help you design the interface, it will also help you think about how your program should work.

To maximize a form, click the **Maximize** button, the middle button in the upper-right corner of the form.

You can only set properties after you have planned the user interface and created objects with the toolbox controls. For more information, see Developing Visual Basic Programs.

B. For more information, see Developing Visual Basic Programs.

Careful up-front planning is always the first step for the happy programmer. For more information, see Developing Visual Basic Programs.

Compiling your program is the final step in the development process. (But how sweet it is!) For more information, see Developing Visual Basic Programs.

The toolbox is a window containing the controls available to the current project. For more information, see Starting Visual Basic.

The Properties window displays the property settings for the currently highlighted object. For more information, see Starting Visual Basic.

The Immediate window, which appears at the bottom of the screen, is used for debugging. For more information, see Starting Visual Basic.

The Control Panel is part of the Windows operating system, not Visual Basic. For more information about the actual components of the development environment, see Starting Visual Basic.

You’re thinking of the Code window. For more information, see The Form Window.

You’ll create a form for each window in your program. For more information, see The Form Window.

The Form window has nothing to do with formulas. For more information, see The Form Window.

The Form window is not dockable, but you can position it on the screen with the Form Layout window. For more information, see The Form Window.

The Property window only works while your program is under development (in design mode.) You must use program code to change object properties while your program runs. For more information, see Properties Window.

The Property window works when you are in design mode, not runtime mode. For more information, see Properties Window.

You’re probably thinking about the capable Project window. For more information, see Properties Window.

The Code window is the tool you use to write code. For more information, see Properties Window.

You can’t set properties with the Project window. For more information, see Project Window.

The Project window does too have those buttons! (Check the ToolTip.) For more information, see Project Window.

Outlining symbols (those + and - signs) show the project hierarchy. For more information, see Project Window.

The Project window shows you the filenames of each component in a project. For more information, see Project Window.

Although the CommandButton control can display text in a button, its formatting capabilities are limited. For more information, see Creating the User Interface.

The PictureBox control is used to display electronic artwork. For more information, see Creating the User Interface.

Although you can use other controls to display text, Label has the best formatting capabilities. For more information, see Creating the User Interface.

Although we haven’t covered it yet, the Timer control functions like an egg timer, and it's invisible on the form. For more information, see Creating the User Interface.

The purpose of program code is not to protect programs or make them difficult to read (although some programmers have been accused of using it that way!) Well-written program code should be neat, tidy, and quite decipherable. For more information, see Writing Program Code.

You’re probably thinking about the files shown in the Project window. For more information, see Writing Program Code.

Although it might look like a foreign language, program code is not the official tongue of any country. However, localized versions of Visual Basic are available, so that programmers can write applications in a variety of languages. For more information, see Writing Program Code.

Yup, you nailed it. You'll learn plenty about Visual Basic program code as you work though this course. For more information, see Writing Program Code.

```
Private Sub Command1_Click()
    Label1.Caption = "Welcome"
    Label2.Caption = "Please enter your name below:"
    lblDigitalClock.Caption = Time
End Sub
```

```
Private Sub Command1_Click()
    'The following line places text in a text box:
    Text1.Text = "Enter your delivery note here."
 End Sub

Private Sub Command2_Click()
    'This line saves input from a text box in a variable:
    DeliveryNote = Text1.Text
End Sub
```

Visual Basic also provides several other controls that you can use to collect user input in a program:

text boxes	Accept typed input.
menus	Present commands that can be clicked.
dialog boxes	Offer a variety of elements that can be chosen individually or selected in a group.

You'll learn about these features in other chapters in this course.

```
Private Sub Form_Load()
    List1.AddItem "Extra hard disk"
    List1.AddItem "Printer"
    List1.AddItem "Satellite dish"
End Sub
```

```
Private Sub Form_Load()
    Combo1.AddItem "U.S. Dollars"
    Combo1.AddItem "Check"
    Combo1.AddItem "English Pounds"
End Sub
```

Temporary containers used to hold data while a program is running are called variables. For more information about methods, see Programmer Talk.

Objects on forms are created with toolbox controls. For more information about methods, see Programmer Talk.

In program code, properties and methods have a similar syntax, but they perform different functions. For more information about methods, see Programmer Talk.

When used in program code, method keywords (methods) and property settings have a similar syntax, but methods perform an action rather than hold a value. For more information about methods, see Programmer Talk.

The typical abbreviation for a check box object is chk. For more information about standard naming conventions, see Object Naming Conventions.

Although it is not a requirement, many Visual Basic programmers start their object names with a three-letter abbreviation so they can identify the object in program code. For more information about standard naming conventions, see Object Naming Conventions.

The typical abbreviation for a command button object is cmd. For more information about standard naming conventions, see Object Naming Conventions.

The typical abbreviation for a common dialog object is dlg. For more information about standard naming conventions, see Object Naming Conventions.

The DriveListBox control can only set the current drive on a system. For more information, see File System Controls.

With DirListBox control the user can to set the current open folder while a program runs. For more information, see File System Controls.

The FileListBox control can determine only the currently selected file. For more information, see File System Controls.

The OLE control is used to launch Windows applications or registered application objects. For more information about setting the current folder, see File System Controls.

The ListBox control only allows the user to select one option at a time. For more information, see Data Input Controls.

The ComboBox control allows only the user to select one option at a time. For more information, see Data Input Controls.

The OptionButton control allows only the user to select one option at a time. For more information, see Data Input Controls.

The CheckBox control allows the user to select more than one of the listed options, or none at all. For more information, see Data Input Controls.

ActiveX doesn't appear in the Insert Object dialog box. For more information, see Working with Other Applications.

Macros aren’t listed in the Insert Object dialog box. For more information, see Working with Other Applications.

Although you can open existing documents by selecting the Create From File option, they don’t appear in the Insert Object dialog box. For more information, see Working with Other Applications.

The objects that appear in the Insert Object list box are those recorded in the system registry during installation. For more information, see Working with Other Applications.

Bound controls are simply toolbox controls that use data objects as their input source. For more information, see Data.

You’re probably thinking of the CommonDialog control. For more information, see Data.

Setting a data control Connect property is one of the important steps in configuring the Data control. However, a Data control is not considered a bound control. For more information, see Data.

Alignment has nothing to do with bound controls. This answer was a red herring. For more information, see Data.

This is how you open new projects in the Visual Basic development environment. For more information, see Installing ActiveX Controls.

Custom controls are added to the toolbox by choosing the Components command and picking controls on the Controls tab. For more information, see Installing ActiveX Controls.

This is how OLE objects are added to a form. For more information, see Installing ActiveX Controls.

Although installing new software may increase your cache of application objects, it will not automatically update your toolbox. For more information, see Installing ActiveX Controls.

Object names assigned with the **Name** property are limited to 40 characters in length.

In a typical Windows application, not all menu commands are available at the same time. In a typical **Edit** menu, for example, the **Paste** command is available only when there is data on the Clipboard. When a command is disabled, it appears in dimmed (gray) type on the menu bar. You can disable a menu item by:

- Clearing the **Enabled** check box for that menu item in the Menu Editor.
- Using program code to set the item's Enable property to False. (When you’re ready to use the menu command again, set its Enable property to True.)

```
Private Sub mnuOpenItem_Click()
    CommonDialog1.Filter = "Metafiles (*.WMF)|*.WMF"
    CommonDialog1.ShowOpen
    Image1.Picture = LoadPicture(CommonDialog1.FileName)
    mnuCloseItem.Enabled = True
End Sub
```

```
Private Sub mnuTextColorItem_Click()
    CommonDialog1.Flags = &H1&
    CommonDialog1.ShowColor
    Label1.ForeColor = CommonDialog1.Color
End Sub
```

Every menu and command needs a unique name. By convention, menus and commands begin with the letters mnu. For more information, see Using the Menu Editor.

You assign the menu or command name that appears on your form by using the Caption text box. For more information, see Using the Menu Editor.

You assign shortcuts by using the Shortcut drop-down list box. For more information, see Using the Menu Editor.

You specify access keys with the & character in the Caption text box. For more information, see Using the Menu Editor.

Commands started by function keys and key combinations are called shortcut keys. For more information, see Adding Access and Shortcut Keys.

Access keys sound like they might have something to do with Microsoft Access, but they're really just a menu term. For more information, see Adding Access and Shortcut Keys.

Access keys are the underlined letters on menu items (typically the first letters). For more information, see Adding Access and Shortcut Keys.

Access keys run menu commands. For more information, see Adding Access and Shortcut Keys.

The Open dialog is displayed by using the ShowOpen method. For more information, see Using the CommonDialog control.

The Print dialog is displayed by using the ShowPrinter method. For more information, see Using the CommonDialog control.

The Font dialog is displayed by using the ShowFont method. For more information, see Using the CommonDialog control.

The Find dialog box is not supplied by the CommonDialog control. For more information, see Using the CommonDialog control.

To disable a menu command with program code, set the Enabled property of a menu command object to False. For more information, see Processing Menu Choices.

There is no Disabled property for menu command objects. For more information, see Processing Menu Choices.

ShowOpen is a method used with a common dialog object to display the Open dialog box. For more information, see Processing Menu Choices.

False is not a property; rather, it is the value you *assign* to the Enabled property to disable a command. For more information, see Processing Menu Choices.

Although metafiles are typically displayed in image box objects, this statement does not reference an image box. For more information, see Creating Dialog Boxes.

This program statement does not create an object. For more information, see Creating Dialog Boxes.

The Filter property controls the default filetypes displayed in a dialog box created a common dialog object. For more information, see Creating Dialog Boxes.

Metafiles are simply the files chosen for this example. Filters can display any file type--.doc, .xls, .bmp, and so forth. For more information, see Creating Dialog Boxes.

If you decide to always declare your variables by using the Dim statement, you can ask Visual Basic to enforce your decision by setting the **Require Variable Declaration** checkbox in the **Options** dialog box. When you set this option, Visual Basic generates an error message whenever it finds a variable that has not been explicitly declared in the code.

u To require variable declaration

1. From the **Tools** menu, click **Options**.

2. Click the **Editor** tab.

3. Select the **Require Variable Declaration** check box.

To turn off this option, simply carry out these steps in reverse order.

```
Dim LastName                 'declare LastName variable

LastName = "Smart"           'store "Smart" in variable
Label1.Caption = LastName    'display "Smart" in Label1 object

LastName = 99                'now place 99 in variable
Label2.Caption = LastName    'display this value in Label2 object
```

Variable storage size is measured in bytes—the amount of storage space required to store 8 bits (approximately 1 character).

```
Private Sub List1_Click()
    'Variable declaration section
    Dim Birds%, Loan&, Price!, Pie#, Debt@, Dog$, Total
    Dim Flag As Boolean
    Dim Birthday As Date

    'Select Case processes the user's choice
    Select Case List1.ListIndex
    Case 0
        Birds% = 37
        Label4.Caption = Birds%
    Case 1
        Loan& = 350000
        Label4.Caption = Loan&
    Case 2
        Price! = -1234.123
        Label4.Caption = Price!
    Case 3
        Pie# = 3.1415926535
        Label4.Caption = Pie#
    Case 4
        Debt@ = 299950.95
        Label4.Caption = Debt@
    Case 5
        Dog$ = "German Wire-haired Pointer"
        Label4.Caption = Dog$
    Case 6  'True is stored as -1 in code, False as 0
        Flag = True
        Label4.Caption = Flag
    Case 7  'Note # symbol and Format function here
        Birthday = #11/19/63#
        Label4.Caption = Format$(Birthday, "dddd, mmmm dd, yyyy")
    Case 8
        Price = 99.95
        Label4.Caption = Price
    End Select
End Sub
```

```
Type Employee
    Name As String
    DateOfBirth As Date
    HireDate As Date
End Type
```

Dim is the keyword you use when you want to declare variables before you use them. For more information about the Dim keyword, see Explicit Declarations

When you implicitly declare a variable, you declare it the first time you use it (without the Dim keyword). For more information about the Dim keyword, see Implicit Declarations.

The Dim keyword is used with variables, not menus. For more information about the Dim keyword, see Creating Variables.

Dim is not used with formulas. For more information about the Dim keyword, see Creating Variables.

A string variable is created by placing the $ symbol after the variable name. For more information, see Fine-tuning Variable Size.

An integer variable is created by placing the % symbol after the variable name. For more information, see Fine-tuning Variable Size.

A single-precision floating point variable is created by placing the ! symbol after the variable name. For more information, see Fine-tuning Variable Size.

If you don’t specify a data type for your variable, Visual Basic will create a variant variable, which can contain any type of data. For more information, see Fine-tuning Variable Size.

InputBox$ is the name of the function you are using. For more information, see Using Functions.

“Please enter your name” appears inside the dialog box, not on the title bar. For more information, see Using Functions.

The user name does not appear in the dialog box. In this program statement, FullName is a variable name, not an output value. For more information, see Using Functions.

As you can see when you execute this statement, Visual Basic places the name of the project in the title bar of your InputBox unless you specify a different title with program code. For more information, see Using Functions.

FavoriteMeal is the name of the variable that is displayed, not the content of the variable. For more information, see Advanced Operators.

The & symbol is not part of the contents of the FavoriteMeal variable, but the string concatenation operator, which combines text values. For more information, see Advanced Operators.

SalmonPasta (with no space between the words) is the correct answer because the & operator joins words without leaving spaces. (You need to place spaces intentionally.) For more information, see Advanced Operators.

The string concatenation (&) operator does not automatically place a space between words it combines. To produce the output "Salmon Pasta" with the space, use the syntax Label1.Caption = "Salmon " & "Pasta". For more information, see Advanced Operators.

The / operator is the division operator, which produces a remainder when necessary. For more information, see Basic Arithmetic Operators.

The \ operator is the integer division operator, which truncates (ignores) any remainder. For more information, see Advanced Operators.

The Mod operator is the reminder division operator, which produces *only* the remainder when two numbers are divided. For more information, see Advanced Operators.

The ^ operator is the exponential operator, which raises numbers to a power. For more information, see Advanced Operators.

You’re thinking of the Sqr function, which returns the square root of a number. For more information, see Mathematical Functions.

The Val function is especially useful if you need to convert a number that the user typed into a text-based control into a number that you can use in calculations. (For example, Val function converts the text string “36” into the number 36.) For more information, see Mathematical Functions.

You’re thinking of the Rnd function, which generates a random number great than or equal to zero and less than 1. For more information, see Mathematical Functions.

You’re thinking of the Str function, which converts a number to a numeric string. For more information, see Mathematical Functions.

For more information about how Visual Basic evaluates formulas, see Operator Precedence.

For more information about how Visual Basic evaluates formulas, see Operator Precedence.

For more information about how Visual Basic evaluates formulas, see Operator Precedence.

For more information about how Visual Basic evaluates formulas, see Operator Precedence.

```
If AdjustedIncome <= 24650 Then        '15% tax bracket
    TaxDue = AdjustedIncome * 0.15
ElseIf AdjustedIncome <= 59750 Then    '28% tax bracket
    TaxDue = 3697 + ((AdjustedIncome - 24650) * 0.28)
ElseIf AdjustedIncome <= 124650 Then   '31% tax bracket
    TaxDue = 13525 + ((AdjustedIncome - 59750) * 0.31)
ElseIf AdjustedIncome <= 271050 Then   '36% tax bracket
    TaxDue = 33644 + ((AdjustedIncome - 124650) * 0.36)
Else                                   '39.6% tax bracket
    TaxDue = 86348 + ((AdjustedIncome - 271050) * 0.396)
End If
```

```
Select Case Age
Case Is < 13
    Label1.Caption = "Enjoy your youth!"
Case 13 To 19
    Label1.Caption = "Enjoy your teens!"
Case 21
    Label1.Caption = "You can drink wine with your meals."
Case Is > 100
    Label1.Caption = "Looking good!"
Case Else
    Label1.Caption = "That's a nice age to be."
End Select
```

A **Select Case** decision structure is usually much clearer than an **If...Then** structure, and it's more efficient when you're making three or more branching decisions based on one variable or property. However, when you're making two or fewer comparisons, or when you're working with several different variables, you'll probably want to use an **If...Then** decision structure.

```
Select Case Age
Case 16
    Label1.Caption = "You can drive now!"
Case 18
    Label1.Caption = "You can vote now!"
Case 21
    Label1.Caption = "You can drink wine with your meals."
Case 65
    Label1.Caption = "Time to retire and have fun!"
End Select
```

```
Select Case Age
Case 16
    Label1.Caption = "You can drive now!"
Case 18
    Label1.Caption = "You can vote now!"
Case 21
    Label1.Caption = "You can drink wine with your meals."
Case 65
    Label1.Caption = "Time to retire and have fun!"
Case Else
    Label1.Caption = "You're a great age! Enjoy it!"
End Select
```

```
Private Sub Command1_Click()
    Stop    'enter break mode
    Age = Text1.Text

    If Age > 13 And Age < 20 Then
        Text2.Text = "You're a teenager."
    Else
        Text2.Text = "You're not a teenager."
    End If
End Sub
```

Conditional expressions evaluate to True or False, not a numeric value. For more information, see Conditional Expressions.

Conditional expressions evaluate to True or False, not a numeric value. For more information, see Conditional Expressions.

Since 30 is greater than 20, the conditional expression evaluates to True. You can use conditional expressions like this in If…Then decision structures (although they are more useful if they contain variables or properties). For more information, see Conditional Expressions.

30 is greater than 20, so the expression evaluates to True. For more information, see Conditional Expressions.

The Else keyword is used to execute statements if none of the conditions in the If…Then decision structure is true. For more information, see If…Then Decision Structures.

The problem here is a typo. The proper spelling of the keyword that introduces multiple conditional expressions is ElseIf (with no space). For more information, see If…Then Decision Structures.

The End keyword is used to close the If…Then decision structure. For more information, see If…Then Decision Structures.

The Then keyword, which is placed after the conditional expression, identifies a block of statements that will be executed if the conditional expression evaluates to True. For more information, see If…Then Decision Structures.

Both expressions in the first conditional expression evaluate to False, so the answer cannot be the first option.
For more information, see Logical Operators.

The conditional expression in the first ElseIf clause evaluates to False, so the second option is not executed. For more information, see Logical Operators.

Since no conditional expression evaluates to True, the Else clause is executed. For more information, see Logical Operators.

Since the If…Then decision structure contains an Else clause, one of the selections must be executed. (The Label1.Caption object won't be blank.) For more information, see Logical Operators.

A syntax error is a programming mistake that violates the rules of Visual Basic, such as a misspelled keyword. For more information, see Three Types of Errors.

A runtime error is a mistake that causes a program to stop running unexpectedly. Typically, a runtime error is caused by an outside event or a missing file or component causes this type of error. For more information, see Three Types of Errors.

A logic error is a human error that results from incomplete or inaccurate reasoning. For more information, see Three Types of Errors.

A typing error is a syntax error. For more information, see Three Types of Errors.

The View Code button, which displays the Code window, is located in the Project window. For more information, see Using Break Mode.

The Step In button, which executes program statements one line at a time, is located on the Debug toolbar. For more information, see Using Break Mode.

The Quick Watch button, which displays the selected variable in the Watch window, is located on the Debug toolbar. For more information, see Using Break Mode.

The Break button, which stops run mode and enters break mode (so you can begin debugging), is located on both the Debug toolbar and the Standard toolbar. For more information, see Using Break Mode.

Nicely done. This is a trick question that highlights an important debugging concept—although your program code may appear syntactically correct to the compiler, it may contain a logic error that produces unexpected results. (Try running the code in Visual Basic to test it!)

In this case, the problem is the order of the conditional expressions in the If…Then and ElseIf clauses. For this code to work correctly, the order needs to be reversed. For more information, see Select Case Decision Structures and Three Types of Errors.

This is a trick question that highlights an important debugging concept—although your program code may appear syntactically correct to the compiler, it may contain a logic error that produces unexpected results. (Try running the code in Visual Basic to test it!)

In this case, the problem is the order of the conditional expressions in the If…Then and ElseIf clauses. For this code to work correctly, the order needs to be reversed. For more information, see Select Case Decision Structures and Three Types of Errors.

This is a trick question that highlights an important debugging concept—although your program code may appear syntactically correct to the compiler, it may contain a logic error that produces unexpected results. (Try running the code in Visual Basic to test it!)

In this case, the problem is the order of the conditional expressions in the If…Then and ElseIf clauses. For this code to work correctly, the order needs to be reversed. For more information, see Select Case Decision Structures and Three Types of Errors.

This decision structure assigns a value to the Caption property when the Temp variable is 95. Actually, this is a trick question that highlights an important debugging concept—although your program code may appear syntactically correct to the compiler, it may contain a logic error that produces unexpected results. (Try running the code in Visual Basic to test it!) For more information, see Select Case Decision Structures and Three Types of Errors.

In the previous **For…Next** loop, you probably noticed that Visual Basic included a carriage return after each **Print** statement. Visual Basic keeps track of the cursor location on the form and increments it after each statement producing text output. If you run the **For…Next** loop again before stopping the program, a new set of lines will be printed starting right where the last ones left off. This could send some of the output off the bottom edge of the form. (You can move the cursor by using the form's CurrentX and CurrentY properties in program code.)

To exit an endless loop, press CTRL+BREAK, and then click **End** on the toolbar.

In addition to user entry, you can use the egg timer technique to display a welcome or copyright message on the screen, or you can use it to repeat an event at a set interval, such as saving a file to disk every 10 minutes.

For is a keyword that begins a For…Next loop. For more information, see Anatomy of a For…Next loop.

Next is a keyword that ends a For…Next loop. For more information, see Anatomy of a For…Next loop.

10 is the number of times the For…Next loop is repeated. For more information, see Anatomy of a For…Next loop.

i is the name of the counter variable in this loop. If more intuitive names such as Days, Tries, or Count make more sense in the context of your program, you can use them. For more information, see Anatomy of a For… Next loop.

You're thinking of the semicolon (;) symbol. For more information, see Using the Counter Variable in a Loop.

The comma inserts a tab space between elements in an expression list when used with Print. For more information, see Using the Counter Variable in a Loop.

Not quite. Although the Print method places the cursor on the next line when it is finished displaying output, the comma character is used to adjust the spacing within the line. For more information, see Using the Counter Variable in a Loop.

The comma symbol does not assign values. For more information, see Using the Counter Variable in a Loop.

Nicely done. This loop will print the numbers 2 and 4, then stop. For more information, see Complex For…Next Loops and Using the Exit For Statement.

Not quite. This loop will print the numbers 2 and 4, then stop. Although the counter variable reaches 6, the Exit For statement terminates the loop before the Print statement is executed. For more information, see Complex For…Next Loops and Using the Exit For Statement.

If there was no Exit For statement, you'd be exactly right. However, Exit For causes the loop to end early. Try another guess, or see Complex For…Next Loops and Using the Exit For Statement.

You're not taking into account the Step and Exit For keywords, which cause this loop to run a few less times. For more information, see Complex For…Next Loops and Using the Exit For Statement.

When you know how many times the loop should be executed, you should use a For…Next loop rather than a Do loop. For more information, see Writing Do Loops.

Do loops are most useful when you're not sure how long you want to repeat a loop but you do know when you're finished. For more information, see Writing Do Loops.

Actually, Do loops are more susceptible to endless loops than For…Next loops, because you can inadvertently create conditions, in which the exit condition never evaluates to True. For more information, see Writing Do Loops.

The Step keyword is used with For…Next loops. For more information, see Writing Do Loops.

Although Done is clearly identified as the exit word in the InputBox prompt, it does not work to end the loop. For more information, see Using the Until Keyword.

The input variable is tested against the "Quit" string in the body of the loop, but this test does not end the loop. For more information, see Using the Until Keyword.

Although the word is non-intuitive and not mentioned in the dialog box prompt, Continue is actually the only word the user can type to exit this loop. What an unfortunate loop, indeed! For more information, see Using the Until Keyword.

Although the user of this loop will probably never figure out how to exit the loop, it is not *technically* an endless loop. That's because there's a word that will make the exit condition evaluate to True. (However, if you picked this answer recalling your own frustration with poorly designed programs, you are forgiven.) For more information, see Using the Until Keyword.

Time is a program statement that displays the current time from your computer's system clock. For more information about Timer properties, see Using the Timer Control.

Interval is a property used to set how long the timer "ticks," but it doesn't start the timer. For more information about Timer properties, see Using the Timer Control.

True is a value that you can assign to certain object properties. However, True itself is not a property name. For more information about timer properties, see Using the Timer Control.

Enabled is a timer property that you can set either at design time with the Properties window or at run time with program code. To enable a timer, assign Enable a value of True. To disable a timer (stop it from running), assign Enable a value of False. For more information about timer properties, see Using the Timer Control.

The Interval property requires a number. For more information, see Setting a Time Limit for Input.

The Interval property is measured in milliseconds, not minutes. For more information, see Setting a Time Limit for Input.

The Interval property is measured in milliseconds, not seconds. For more information, see Setting a Time Limit for Input.

Bingo. Although specifying the interval in milliseconds can create some big numbers, this level of detail helps you fine-tune timing of animations and other operations that require extremely fast clock ticks. For more information, see Setting a Time Limit for Input.

If you use the **Show** method before using the **Load** statement, Visual Basic will load and display the specified form automatically. Visual Basic provides a separate **Load** statement to let programmers preload forms in memory so that the **Show** method works quickly and users don’t notice any performance lag. It’s a good idea to preload your forms, especially if they contain several objects or pieces of artwork.

By default, Visual Basic uses Form1 as your startup form (the first form that is opened when the program runs). If you would like to specify a different startup form, choose **Properties** from the **Project** menu, click the **General** tab, and pick a project form in the **Startup Form** drop-down list box.

If you get stuck in an error loop and can’t get out, press CTRL+BREAK.

```
Retries = 0                'initialize counter variable
On Error GoTo DiskError
Image1.Picture = LoadPicture("a:\prntout2.wmf")
Exit Sub                   'exit the procedure

DiskError:
MsgBox (Err.Description), , "Loading Error"
Retries = Retries + 1     'increment counter on error
If Retries >= 2 Then
    Resume Next
Else
    Resume
End If
```

The About Dialog form does appear, and you can use it to display copyright and version information about your program. For more information, see Creating New Forms.

The Splash Screen form does appear, and you can use it to display the title of your program while it loads. For more information, see Creating New Forms.

The Log In form does appear, and you can use it to prompt the user for a login name and password. For more information, see Creating New Forms.

Right, there is no pre-defined form for an error handler. For more information, see Creating New Forms.

Actually, a modal form must be used when it is displayed—the user can't switch away from it. For more information, see Creating New Forms.

Right! Nonmodal or modeless forms are flexible forms that the user can switch away from if necessary. For more information, see Creating New Forms.

An MDI Form is a parent form in a document-centered application. MDI forms can be either modal or nonmodal. For more information, see Creating New Forms.

In a Visual Basic application, the startup form is the first form that appears.. For more information, see Creating New Forms.

Right. A WindowState value of 1 minimizes the form (places it on the taskbar). For more information, see Hiding and Unloading Forms.

To maximize a form window, assign the WindowState property a value of 2. To close forms use the Hide statement. For more information, see Hiding and Unloading Forms.

You open forms with the Show statement. For more information, see Hiding and Unloading Forms.

You close forms with the Hide statement. For more information, see Hiding and Unloading Forms.

You're thinking of the NewPage method. For more information, see The Printer Object.

You're thinking of the KillDoc method. For more information, see The Printer Object.

Right. EndDoc finishes a print job and sends it off to the print spooler. For more information, see The Printer Object.

You're thinking of the Print method, which sends the specified output to the Printer object. For more information, see The Printer Object.

Not quite. PrintForm is not limited to printing graphic objects only. For more information, see Printing Graphics.

PrintForm can print graphic objects too. For more information, see Printing Graphics.

PrintForm can print the objects on any type of form. For more information, see Printing Graphics.

Although PrintForm is often used for printing graphics, it actually prints all the objects on a form—unless you make them invisible. For more information, see Printing Graphics.

The Resume statement returns control to the statement that caused the error and directs the compiler to try it again. For more information, see Using Resume and Resume Next.

Resume Next skips the statement that caused the error in your program—your only choice if the error condition cannot be fixed and you must return to the program. For more information, see Using Resume and Resume Next.

The On Error statement sets an event trap for the error handler. For more information, see Detecting the Error and Describing the Error.

Typically the Err.Number property is used to diagnose the runtime error. For more information, see Detecting the Error.

Resume Next directs the course of the error handler, but does not contain a description of the error message. For more information, see Describing the Errors.

Err.Number contains the message error number, but not a description of the problem. For more information, see Describing the Errors.

Err.Description contains a text string from the compiler that identifies the reason for the runtime error. For more information, see Describing the Error.

MsgBox is not a property, but it would be a useful function for displaying the error message. For more information, see Describing the Error.

A picture box object is usually used with the **Move** method because it creates less flicker on the screen than an image object. For this reason, picture boxes are typically recommended for animation effects.

The Line control creates different types of lines. For more information, see Line Control and Shape Control.

You can create rectangles, squares, circles, and ovals with the Shape control Shape property. For more information, see Shape Control.

PSet is a graphics method used in program code to create graphic effects. For more information, see Shape Control and Graphical Methods.

The Timer control is used for animation but not for creating artwork. For more information, see Shape Control.

DragDrop is the name of an event procedure used to control the drag-and-drop effect. For more information, see Setting Drag Mode.

DragOver is the name of an event procedure used to perform an action when one object is moved over another. For more information, see Setting Drag Mode.

With the DragIcon property you define an icon that Visual Basic displays during the drag motion. For more information, see Setting Drag Mode.

To enable drag and drop for an object, set its DragMode property to 1. For more information, see Setting Drag Mode.

You're thinking of the Source parameter. For more information, see Setting Drag Mode.

Although there are no formal rules that control how a Tag property is used, it is commonly used by programmers to carry an identification note about an object. For more information, see Setting Drag Mode.

The DragIcon property contains the icon used during a drag-and-drop operation. For more information, see Setting Drag Mode.

The mouse pointer is defined by a form's MousePointer property. For more information, see Mouse Pointer.

The Source parameter contains a copy of the object used during the drag and drop, so that you can process it with program code. For more information, see Using the Tag Property.

The Picture property contains a piece of artwork, not a copy of the drag object. For more information, see Using the Tag Property.

Although by convention the Tag property is used to identify key objects in the user interface, this property does not contain a copy of the drag object. For more information, see Using the Tag Property.

The Visible property is used to hide or show an object. For more information, see Using the Tag Property.

One twip is equal to 1/20 of a printer's point or 1/1440 inch. For more information, see The Form's Coordinate System.

The dots on the form grid are not twips. For more information, see The Form's Coordinate System.

Twip units are much smaller than one inch. Their small size helps you make very fine measurements. For more information, see The Form's Coordinate System.

Nice try! For more information, see The Form's Coordinate System.

This statement moves the object in two directions. For more information, see Moving Objects.

This statement moves the object in two directions. For more information, see Moving Objects.

By adding positive values to the Left and Top properties, the object moves right and down. For more information, see Moving Objects.

To move left and up, you would need to subtract values from the Left and Top properties. For more information, see Moving Objects.

Setting the Visible property to False makes an object invisible, but it will not stop an animation sequence. (Visual Basic will keep working, without displaying any output.) For more information, see Creating Animation with the Timer.

Hiding a drag-and-drop object does not stop animation. For more information, see Creating Animation with the Timer.

Enabling a timer object usually starts an animation sequence. For more information, see Creating Animation with the Timer.

When your objects have moved a sufficient distance, you can stop the animation sequence by disabling the timer object. For more information, see Creating Animation with the Timer.

To learn more about available graphics methods, search for *line, circle,* or *Pset* in the Visual Basic online Help.

```
Function TotalTax(Cost)
    StateTax = Cost * 0.05  'State tax is 5%
    CityTax = Cost * 0.015  'City tax is 1.5%
    TotalTax = StateTax + CityTax
End Function
```

```
AddNameToListBox "Kimberly"  'always add two names
AddNameToListBox "Rachel"
Do                           'then let user add extra names
    NewName$ = InputBox("Enter a list box name.", "Add Name")
    AddNameToListBox NewName$
Loop Until NewName$ = ""
```

Local variables are not declared in standard modules. For more information, see Variable Scope.

A variable declared in an event procedure can be used only in that event procedure. For more information, see Variable Scope.

Public variables, not local variables, hold their value in all event procedures. For more information, see Variable Scope.

Local variables hold their values only in the procedure that they are declared in. For more information, see Variable Scope.

Public variables are defined in the Declarations section of a standard module. For more information, see Creating Standard Modules.

Functions shared throughout the program are created in the standard module. For more information, see Creating Standard Modules.

Sub procedures shared throughout the program are created in the standard module. For more information, see Creating Standard Modules.

Standard modules contain space for code but not for objects. You can only use toolbox controls in Form modules. For more information, see Creating Standard Modules.

Public variables are created in standard modules. For more information, see Working with Public Variables.

To create a public variable, you type the Public keyword and the variable name in a standard module's Declarations section. For more information, see Working with Public Variables.

Public variables are not created within Function procedures but in the Declarations section of a standard module. For more information, see Working with Public Variables.

By default, Public variables are variants. However you can also create Public variables with a specific data type. For more information, see Working with Public Variables.

Arguments provide a mechanism for passing information to a Visual Basic Function or Sub that you have created. They can be passed by reference or by value. For more information, see Function Syntax.

The final value returned by a function is not called an argument but simply the function's return value. It is neither associated with a function's arguments, nor its internal variables. For more information, see Function Syntax.

Procedure arguments are not used to declare variables. For more information, see Function Syntax.

Arguments received by a Function or Sub procedure are separate from local variables declared within the procedure. For more information, see Function Syntax.

Did you forget to include the argument (500) passed to function through the num parameter? For more information, see Function Syntax.

Be sure you include the value 500 passed to the function during the function call and account for the addition and division operations inside the function. For more information, see Function Syntax.

The formula calculated by the function is (500 + 100) / 2. For more information, see Function Syntax.

Be sure you include the value 500 passed to the function during the function call and account for the addition and division operations inside the function. For more information, see Function Syntax.

Since the Sum variable is enclosed in parentheses during the procedure call, it is passed by value and cannot be modified. After the procedure call, it contains the same value it started with. For more information, see Sub Procedure Syntax.

The syntax of the procedure call is the key to this question. For more information, see Sub Procedure Syntax.

You've analyzed the SquareIt procedure correctly, but the syntax of the procedure call produces a different result. For more information, see Sub Procedure Syntax.

The syntax of the procedure call is the key to this question. For more information, see Sub Procedure Syntax.

Arguments enclosed in quotation marks are literal values or character strings, and are passed by value, not by reference. For more information, see Sub Procedure Syntax.

Arguments passed by value cannot be modified by a procedure call. For more information, see Sub Procedure Syntax.

Passing arguments by reference is a useful way to receive updated information in a program. If they are updated during a procedure call, you get to use the new value. For more information, see Sub Procedure Syntax.

By reference and by value are opposites. For more information, see Sub Procedure Syntax.

The Open statement opens files, not dialog boxes. For more information, see The Open Statement.

When you use the Output keyword with the Open statement, Visual Basic creates a new file. In this case, the file's name is stored in the Filename property of the CommonDialog1 object. For more information, see The Open Statement.

The Input keyword directs the Open statement to open existing files. For more information, see The Open Statement.

The Open statement assigns a number to the opened file, not an object. The file number is used later in the Print and Line Input statements. For more information, see The Open Statement.

Using the Input or Output keyword in an Open statement makes a text file sequential. The contents of a sequential file must also be used in sequential order. For more information, see The Open Statement.

Unlike sequential files, database files can be accessed in any order. For more information, see The Open Statement.

Sequential files are typically text files, but how you open them in a program determines whether not they are sequential. For more information, see The Open Statement.

Sequential files are typically text files, but how you open them in a program determines whether or not they are sequential. For more information, see The Open Statement.

Input box cannot hold multiple lines of text and would, therefore, be unsuitable as the basis of a text editor. For more information, see Creating New Text Files.

The Data control is designed to open and configure databases, not to hold text. For more information, see Creating New Text Files.

The entire contents of a text box object are contained in the Text property, which can be written to a file with one Print statement. Text box also supports multiple lines and scrolling, so it is well suited as the basis for a text editor program. For more information, see Creating New Text Files.

The Filter property is not used with the EOF function or in end-of-file tests. To see Filter used in an event procedure, see Save Command Code.

The Filter property is not used with the Close statement. To see Filter used in an event procedure, see Save Command Code.

Filter controls the type of files displayed in the Open and Save As list boxes and the options in the Files of Type drop-down list box. To see Filter used in an event procedure, see Save Command Code.

The source of records is controlled by the data object RecordSource property. To see Filter used in an event procedure, see Save Command Code.

ReadOnly does not control password protection. For more information, see Demonstration: Data Browser Program.

The database type is controlled by the Data control Connect property. For more information, see Demonstration: Data Browser Program.

When set to True, the ReadOnly property prohibits users from changing database fields or records. This is often an important safety precaution. For more information, see Demonstration: Data Browser Program.

Read and write access (the ability to see and change the database) is granted when the ReadOnly property is set to False. For more information, see Demonstration: Data Browser Program.

CommandButton cannot display information from a database. For more information, see Using Bound Controls.

CheckBox can display information when you set its DataField and DataSource properties to an open database. For more information, see Using Bound Controls.

Image can display information when you set its DataField and DataSource properties to an open database. For more information, see Using Bound Controls.

TextBox can display information when you set its DataField and DataSource properties to an open database. For more information, see Using Bound Controls.

If no match is found in a search started by the Seek method, the NoMatch property is set to True. For more information, see Searching Records.

The Index property determines what part of the database you will search. For example, you might use the Title field. For more information, see Searching Records.

The Index property is used to help define a database search. For more information, see Searching Records.

You can give an object the focus by using its SetFocus method. For more information about Index, see Searching Records. To see an example of setting the focus, see Adding Records.

```
Private Sub mnuItemClose_Click() 'when user clicks Close command
    txtFile.Text = ""            'clear text box
    lblFile.Caption = "Load a text file with the Open command."
    mnuItemClose.Enabled = False 'dim Close command
    mnuItemOpen.Enabled = True   'enable Open command
    txtFile.Enabled = False      'disable text box
End Sub

Private Sub mnuItemExit_Click()  'when user clicks Exit command
    End                          'quit program
End Sub

Private Sub mnuItemOpen_Click()  'when user clicks Open command
    Wrap$ = Chr$(13) + Chr$(10)  'create wrap character
    CommonDialog1.Filter = "Text files (*.TXT)|*.TXT"
    CommonDialog1.ShowOpen       'display Open dialog box
    If CommonDialog1.Filename <> "" Then
        Form1.MousePointer = 11  'display hour glass
        Open CommonDialog1.Filename For Input As #1
        On Error GoTo TooBig:    'set error handler
        Do Until EOF(1)          'then read lines from file
            Line Input #1, LineOfText$
            AllText$ = AllText$ & LineOfText$ & Wrap$
        Loop
        lblFile.Caption = CommonDialog1.Filename
        txtFile.Text = AllText$  'display file
        txtFile.Enabled = True
        mnuItemClose.Enabled = True
        mnuItemOpen.Enabled = False 'enable scroll
CleanUp:
        Form1.MousePointer = 0   'reset mouse
        Close #1                 'close file
    End If
    Exit Sub
TooBig:             'error handler displays message
    MsgBox ("The specified file is too large.")
    Resume CleanUp: 'then jumps to CleanUp routine
End Sub
```

```
Private Sub mnuItemSave_Click() 'When user clicks Save
'note: the entire file is stored in a string
    CommonDialog1.Filter = "Text files (*.TXT)|*.TXT"
    CommonDialog1.ShowSave      'display Save dialog
    If CommonDialog1.Filename <> "" Then
        Open CommonDialog1.Filename For Output As #1
        Print #1, txtNote.Text  'save string to file
        Close #1                'close file
    End If
End Sub
```

```
prompt$ = "Enter the full (complete) book title."
'get the string to be used in the Title field search
SearchStr$ = InputBox(prompt$, "Book Search")
datBiblio.Recordset.Index = "Title"      'use Title table
datBiblio.Recordset.Seek "=", SearchStr$ 'and search
If datBiblio.Recordset.NoMatch Then      'if no match
    MsgBox ("Sorry, I couldn’t find your book.")
    datBiblio.Recordset.MoveFirst        'go to first record
End If
```

```
prompt$ = "Enter new record, and click left arrow button."
reply = MsgBox(prompt$, vbOKCancel, "Add Record")
If reply = vbOK Then                'if the user clicks OK
    txtTitle.SetFocus               'move cursor to Title box
    datBiblio.Recordset.AddNew      'and get new record
    'set PubID field to 14          (this field is required
    datBiblio.Recordset.PubID = 14  'by Biblio.mdb)
End If
```

If you want to delete records in a database, you need to set the data object **ReadOnly** property to False before you open the database.

```
prompt$ = "Do you really want to delete this record?"
reply = MsgBox(prompt$, vbOKCancel, "Delete Record")
If reply = vbOK Then              'if the user clicks OK
    datBiblio.Recordset.Delete    'delete current record
    datBiblio.Recordset.MoveFirst 'move to first record
End If
```

```
prompt$ = "Would you like to create a backup copy of the database?"
reply = MsgBox(prompt$, vbOKCancel, datBiblio.DatabaseName)
If reply = vbOK Then     'copy the database if the user clicks OK
    FileNm$ = InputBox$("Enter the pathname for the backup copy.")
    If FileNm$ <> "" Then FileCopy datBiblio.DatabaseName, FileNm$
End If
```

Through Automation, a server application exposes its objects to a client application, and the client application uses them to do meaningful work in a program. For more information, see Understanding Automation.

Although Visual Basic Enterprise Edition can distribute Automation objects over a network, the fundamental goals of Automation are more down to earth: to allow programmers to use their existing applications and stop reinventing the wheel each time a new feature is added. For more information, see Understanding Automation.

Unfortunately, Automation only works if the server or object application has been designed to support objects. Although Automation is gaining ground as an industry standard, older Windows applications cannot take advantage of it. For more information, see Understanding Automation.

Although Visual Basic programmers can enter and edit Automation procedures without having access to a server application, a working copy of the software is required on each of the machines that runs the final program. For more information, see Understanding Automation.

Project Explorer does not list application objects. For more information, see Using the Object Browser.

Windows Explorer does not list application objects. For more information, see Using the Object Browser.

Although the Properties window shows the properties associated with objects in your program, it does not list application object properties. For more information, see Using the Object Browser.

The Object Browser lets you examine the properties, methods, events, and other key values for the application objects on your system. For more information, see Using the Object Browser.

References lists all the object libraries in your system in a simple dialog box. Object libraries are installed automatically when you install application software. For more information, see Browsing an Object Library in Using the Object Browser.

Components sets the controls, designers, and insertable objects in your program. For more information, see Browsing an Object Library in Using the Object Browser.

The Object Browser lets you view the contents of an object library. However, before you view an object library, you must add it to your project. For more information, see Browsing an Object Library in Using the Object Browser.

Project Explorer does not manage your project's object libraries. For more information, see Browsing an Object Library in Using the Object Browser.

The statement `Dim X As Object` declares a new object variable. For more information, see Microsoft Word.

The Text property copies text to Microsoft Word. For more information, see Microsoft Word.

After you declare an object variable, you can use the CreateObject statement to assign an application object to it. For more information, see Microsoft Word.

The statement `Set X = Nothing` releases the object variable X. For more information, see Microsoft Word.

Exactly right—Visual Basic for Applications contains all the standard Visual Basic keywords, **but** it also special features designed for applications. However, this is not the only correct answer! For more information, see Microsoft Word.

Exactly right—Word, Excel, PowerPoint, Access, and Outlook all include Visual Basic for Applications. But this is not the only correct answer! For more information, see Microsoft Word.

Exactly right—by using Visual Basic for Applications, you can write macros in one application and use application objects in another. But this is not the only correct answer! For more information, see Microsoft Word.

Visual Basic for Applications is a standard macro language in all Office 97 applications, capable of sharing information and features via Automation. For more information, see Microsoft Word.

If you use the Add method with the Documents object, it creates a new Word document. (Add is one of Word's most versatile methods.) For more information, see Microsoft Word.

`Selection.Text` assigns text to the current selection in a Word document. For more information, see Microsoft Word.

Application is the name of Word's main object. All supporting objects and collections are below it in the object hierarchy. For more information, see Microsoft Word.

The Dim statement creates a new variable in your program. For more information, see Microsoft Word.

The Quit method closes Office 97 applications. For more information, see Microsoft Word.

To display an application, set the Visible property to True. For more information, see Microsoft Word.

To open a new Word document, use the Documents.Add method. For more information, see Microsoft Word.

You can use the Visible property to hide an application's workspace if you don't want users to see it during an Automation session. However, setting Visible to False just hides the workplace—your users will still see any dialog boxes that appear. For more information, see Microsoft Word.

32-bit
An application specifically designed for a 32-bit computer running the Microsoft Windows 95 or later or Windows NT operating system. (An application that can work with 32 bits of information at a time.)

About Microsoft Visual Basic

The Help menu command that displays online assistance, copyright information, and version number information.

Abs
A function that returns the absolute value of *n*.

access key
The letter the user can press to execute the command when the menu is open.

Add Form
Part of the Visual Basic toolbar, which you use to add forms to your programs.

Add Project
The File menu command that you use to load additional projects into Visual Basic. When you use this command, outlining symbols that can help you organize and switch between projects appear in the Project window.

Add Watch
A command on the Debug menu, with which you add variables to the Watch window.

adding files
Adding individual files to a project by using commands on the Project menu. The changes that you make will be reflected immediately in the Project window.

adding projects
Loading additional projects into Visual Basic with the File menu Add Project command.

advanced operators
Symbols used in special-purpose mathematical formulas and text processing applications. Visual Basic uses these advanced operators:

Mathematical Operation	**Symbol**
Integer (whole number) division	/
Remainder division	Mod
Exponentiation (raising to a power)	^
String concatenation (combination)	&

algorithm
An ordered list of programming steps that you use to guide your programming efforts.

argument
A value or an expression used with an operator or passed to a subprogram (subroutine, procedure, or function). The program then carries out operations using the argument(s).

arithmetic operators

Symbols used to perform arithmetic (numeric) functions—addition, subtraction, multiplication, and division. In Visual Basic, arithmetic operators include:

Mathematical operation	**Symbol**
Addition	+
Subtraction	-
Multiplication	*
Division	/

assignment operator
An operator used to assign a value to a variable.

Atn
A function that returns the arctangent, in radians, of *n*.

Automation
A technology that allows one application to use the features of another application to do meaningful work.

.BAS file
The filename extension of Visual Basic standard module files.

Biblio.mdb

A sample database included with Visual Basic for demonstration purposes. Biblio.mdb contains a current bibliography of books about database programming and related topics.

Boolean data
In Visual Basic, a data type that uses two bytes per variable and that returns a value of True or False.

BorderColor
A property that sets the line color to any Visual Basic standard color.

BorderStyle
A property that makes a line solid, dotted, or dashed.

BorderWidth
A property that adjusts the thickness of the line on your form. This is especially useful when you are creating an underline, or a line that separates one object from another.

bound control
A toolbox control that has been linked to a data object with the DataSource property.

bound object

An object is bound to a database when its DataSource property is set to a valid database name and its DataField property is set to a valid table in the database. After an object is bound to a database, it displays database information automatically.

break mode

A condition during which you interrupt execution at a given spot in your program. Break mode gives you a close-up look at your program while the Visual Basic compiler runs it.

Break
A command on the Run menu that temporarily halts the execution of a program and puts the compiler in break mode for debugging.

by reference
A variable passed by reference can be modified by a procedure and returned to the program.

by value
A variable or literal value (such as a string in quotation marks) passed to a procedure that cannot be modified by the procedure. You can pass a variable by value if you enclose it in parentheses.

byte
In Visual Basic, a data type that requires 1 byte per variable and that returns values from 0 through 256.

Caption
The Visual Basic property that sets the object description that the user will see.

Case Else
Part of the Select Case structure that displays a message if none of the earlier cases returns a value of True.

check box
A control that displays a list of choices and gives the user the option to pick multiple items (or none at all) from the list.

child form
In the hierarchy of Visual Basic forms, a form opened or initiated by another (parent) form.

Classes list box
The part of the Object Browser that lists types of Windows-based application objects.

Clear All Breakpoints
A command on the Debug menu that removes all debugging breakpoints from your application.

Close
When added to program code, this keyword closes the specified Visual Basic object.

Code window
A special text editing window designed specifically for Visual Basic program code. You can display the Code window in either of two ways:

® By clicking the View Code button in the Project window.

® By clicking the View Menu Code command.

Color dialog box

One of the five standard dialog boxes created by the Common Dialog object. The color dialog box provides user-defined color selection from a palette.

color palette
Part of the Visual Basic development environment, with which you choose colors for lines, shapes, and graphics effects.

combo box

A data input object created with the ComboBox control. Data items can be added to, removed from, or sorted in a combo box while your program runs.

CommonDialog
A Visual Basic control that creates one of five standard dialog boxes in your programs.

comparison operator
A symbol or other character that is part of an expression that compares two or more variables, conditions, or values.

conditional expression
Part of a complete program statement that asks a True-False question about a property, variable, or other bit of data in program code.

conditional test
A true or false question in an If...Then or Select...Case decision structure.

Connect
A property that links your program to other applications. Set the Connect property for the application you want to connect to.

constant
A variable, the value of which doesn't change while your program runs.

controls
Visual Basic tools, with which you add, format, and arrange elements of your program user interface.

Controls collection
In Visual Basic terminology, the entire set of objects on a form.

Cos
A function that returns the cosine of the angle *n*. The angle *n* is expressed in radians.

counter variable

A variable that keeps a running total of the number of times a loop is repeated. By convention, counter variables are indicated by the letter i, and the first value is usually 1.

cross
One of the 16 standard Windows mouse pointer styles.

currency

A Visual Basic data type, which uses 8 bytes to store each variable and which returns values that range from –922,337,203,685,477.5808 through 922,337,203,685,477.5807.

current location
In Visual Basic animation, the spot on the Form window (as indicated by the x, y coordinate system) at which a selected object is located.

custom counter
In a loop, variable with a counter pattern other than 1, 2, 3, 4, and so on.

data field
The property you set to the name of the database field you want to display.

data object
An object you create with the data control to search for, add, sort, or delete database information.

data source
A property you use to a data object to create a bound control.

data types

A particular classification of data in a program. In Visual Basic, you can declare variables with specific data types by using a type declaration character. Data types used in Visual Basic include:

® Byte

® Integer

® Long

® Single-precision

® Double-precision

® Currency

® String

® Boolean

® Date

® Variant

database

An organized collection of information stored electronically in a file. Microsoft Visual Basic supports a number of popular database formats, including Microsoft Access, Microsoft FoxPro, Btrieve, Paradox, and dBASE. You can also use Open Database Connectivity (ODBC) client-server databases such as Microsoft SQL Server.

database engine
An underlying technology that controls information access in a database. The database engine in Visual Basic is Microsoft Jet.

database formats
These include Microsoft Access, dBASE, Paradox, FoxPro, and several spreadsheet formats.

DatabaseName
A Visual Basic property used to create data objects. For example, setting the DatabaseName property to Pathname types the pathname of the database you want to open.

Debug toolbar
A Visual Basic debugging tool, which provides tools devoted entirely to tracking down program code errors.

debugging a program
The process of finding and correcting program code errors.

decision structure

A special block of statements that use conditional expressions. Visual Basic uses these types of decision structures: If...Then and Select Case.

declaring a variable

The process of creating a variable in your Visual Basic program code. You can declare variables explicitly with the Dim keyword or implicitly by typing a new variable name in a program statement.

declaring data types

The process of specifying the data types you use in your Visual Basic program code. You can specify some fundamental data types by adding a type-declaration character to the variable name.

design mode (design time)
The part of the software development cycle you spend designing your program

destination path
A variable, which specifies the pathname of a file you want to create.

Dim statement

A program statement with which you declare (dimension) a variable explicitly. You make an explicit declaration by typing a Dim statement and the variable name.

Do loops
Program code that executes a group of statements repeatedly until a specified condition is met (until the variable returns a value of True).

docking windows
Aligning and attaching windows to make all the elements of the programming system visible and accessible.

document files
Files containing formatting codes and other information that can only be displayed by the application that created them (document files created in Microsoft Word, for example).

document-centered applications
A type of program design, in which many windows are used to display or edit a document.

double-precision floating point

A data type that requires 8 bytes to store a single variable and that returns values that range from –1.79769313486232D308 through 1.79769313486232D308.

drag and drop

Moving an object by clicking it, moving the mouse without releasing the mouse button, and releasing the mouse button where you want the object to be.

DragIcon

An object property, which you use to specify a drag icon, which appears while the mouse pointer drags the object.

DragMode
An object property, which you use to allow the user to drag an object while the program runs.

DragOver
An event procedure, which Visual Basic executes when the user drags one object over another.

dynamic link library (DLL) files

Executable routines—usually serving a specific function or purpose—that are stored and loaded only when needed by the program that calls them.

Else
A keyword used in multiple-condition If...Then decision structures.

ElseIf
A keyword used in conditional statements in multiple-condition If...Then decision structures.

enabled property

An object property, which turns on a feature or function when the setting is True and turns off a feature or function when the setting is False.

End
A program statement that stops the execution of a program in the Visual Basic development environment.

EndDoc
A method that signals the end of a printing job.

EndIf
Keywords used to end a multiple-condition decision structure.

endless loop

A loop test that never evaluates to False. An endless loop executes endlessly and makes the program unable to respond to input.

EOF
A keyword that checks for the end of the text file. You use the EOF keyword to load the contents of a text file into a text box.

error handlers
Special routines designed to respond to runtime errors.

event procedures
Procedures that run when an event (such as a mouse click) occurs.

event-driven programming
A style of programming, in which the program tests for and responds to events generated by the user, such as key pressing or mouse clicks.

executable files
Files containing coded instructions for the operating system.

Exit For
A program statement you use to exit a For...Next loop before the loop has finished executing.

Exit Sub
A program statement you use to exit a Sub procedure.

Exp
A function that returns the constant *e* raised to the power *n*.

explicit declaration

Creating a variable by using the Dim (dimension) statement. You declare the variable by typing a Dim statement and the variable name.

exponentiation
An advanced operator, which uses the caret (^) symbol to indicate exponentiation (raising a number to a given power.)

exposing application features

Making a set of application features available as a collection of objects with their associated properties, methods, and events. Automation makes it possible to expose Windows application features to Visual Basic programmers.

expression
A combination of symbols (identifiers, variables, and operators) that produce a result when they are evaluated.

FileCopy
A statement you use to create a backup copy of one or more files from within a Visual Basic program. FileCopy makes a separate copy of the file.

filename extension

A set of characters added to a filename that clarifies the content or function of the file. Visual Basic filenames use these extensions:

® .vbp	Visual Basic project file
® .frm	Visual Basic form
® .bas	Standard module

filenumber

An object property, an integer from 1 through 255, used to identify a file in your program. When you use *filename*, the file number will be associated with the file whenever you refer to the file in your program code.

FillColor
A Shape control property that specifies the color of a shape.

FillStyle
A Shape control property that specifies the fill pattern.

Font dialog box
The dialog box that appears when you click the Font property in the properties window.

FontSize
The property that specifies the size of text on a form.

For...Next
A type of loop that you can use to execute a specific group of program statements in an event procedure a specific number of times.

Form Layout window
A visual design tool, with which you can control the placement of the forms in the Windows environment while your program runs.

Form window

The Visual Basic window with which you create the user interface and up interface elements. The Form window consists of a default form and a standard grid.

Frame
A Visual Basic toolbox control, with which you create frames that group objects on a form.

FRM
An object name prefix used in naming form objects, for example, frmMedExpense.

front end

A customized database application that takes the fields and records of a database and displays them in a way that is meaningful for a specific group of users. For example, a public library might create a customized version of its card catalog for a group of scientific researchers. In addition to customized information, a front end might also feature customized database tools, including viewing filters, search commands, print commands, and a custom backup tool. With Visual Basic, you can build database front-ends that quickly display just the information your users want, and with only the commands they need to process the data.

function
A statement that performs meaningful work and returns a value to the program.

function procedure
A group of statements that actually perform the work that a function is assigned in the program.

function statement
A block of statements that accomplish the work of the function.

general-purpose procedure
A Sub or Function procedure that performs useful work in your program. Unlike event procedures, general-purpose procedures are not tied to elements in the user interface.

Hide
A method you use to make a form invisible but keep it in memory for use later in the program.

hierarchy

A type of object organization, which branches to smaller and smaller units, each of which is owned by the higher-level object immediately above it. A hierarchy provides an organizational framework that reflects logical links or relationships between its elements. (*See also:* parent file and child file.)

hourglass
A standard Windows mouse pointer design, which indicates that the user needs to wait.

I-beam
A standard Windows mouse pointer design used to indicate cursor position in text-based applications.

identifier

A variable or identifier is a special container that holds data temporarily in a program. You create variables to store calculation results, create file names, process input, and so on. Variables can store numbers, names, property values, and references to objects.

If...Then decision structure

A set of control statements that executes a block of code if a Boolean expression evaluates to True. If...Then decision structures can use one or several conditional statements (*See also:* multiple-condition decision structures.)

Image
A Visual Basic control, with which you create objects that can display graphics as bitmaps, icons, or Windows metafiles.

image objects
Objects that can display graphics as bitmaps, icons, or Windows metafiles (electronic artwork that you can resize.)

Immediate window

A programming tool that helps you debug a program while you are in break mode. When you type a statement in the Code window, Visual Basic executes it and displays the output in the Immediate window. This quick response makes it easier to test the value of specific key variables as your program runs.

implicit declaration

The process of declaring a variable without using the Dim (dimension) statement. To declare a variable implicitly, you simply use the variable on its own without using the Dim statement.

Index
A Recordset property that you can use to define the database field that will be used in a data search.

Input

A keyword used to indicate the mode (how the file will be used) in the Open statement. Using the Input keyword indicates that you want to read the file that you're opening.

input box
A type of data object, with which you gather user input.

InputBox
A function that prompts the user to supply input in a dialog box.

Int
A function that returns the integer portion of *n*. For negative numbers, Fix(*n*) returns a value >= n. (*See also:* Fix(*n*).)

Integer

A data type that requires 2 bytes to store each variable and that returns values that range from –32,768 through 32,767.

integer division
An advanced arithmetic operation, in which the result is an integer (any remainder is discarded.)

invisible controls
Controls that perform special, behind-the-scenes operations in a Visual Basic program and that are not visible to a user while the program runs.

keyword
Any one of dozens of words and phrases used to create Visual Basic program statements.

KillDoc
A printer method that terminates the current print job.

Label
A Visual Basic control, with which you create objects that identify objects on a form (*See also*: Caption.)

Left measure
The distance of an object to the right of the form's origin (top left-hand corner), measured in twips.

Level Up button
Part of the Open Project dialog box that you click to search files one level higher in the file directory.

Line control
A toolbox control that helps you specify line width, color, style, and visibility.

Line Input
A keyword that reads a line of input from a text file.

ListBox
A toolbox control you use to create a list box object.

loading a program

Transferring information from storage (such as your hard disk or CD-ROM) into memory. You load information for processing and program code for execution.

local variable
A variable that can be read or changed only in the procedure that it is declared in.

Locals window
A window in the development environment used in a debugging session to monitor procedure calls.

logic error
An error in reasoning, a programming mistake that makes the program code produce the wrong results.

logical operator
An operator designed to work with Boolean (True and False) values. Logical operators include:

® And

® Or

® Not

® Xor

Long data type

A data type that requires 4 bytes to store each variable and that returns values that range from –2,147,483,648 through 2,147,483,647.

Look In drop-down box
Part of the Open Project dialog box that you click to search for files at the current level in the file directory.

loops
A set of statements that a program executes repeatedly.

Make
A command on the File menu used to create an executable (.exe) file that is capable of running under Windows 95 or later or Windows NT.

mathematical function
A function in a program that performs a set of mathematical operations and returns a value to the program.

maximizing a form
Resizing a form to its maximum size.

MDI form
A form that can be organized in a Multiple Document Interface hierarchy.

Members list box
The part of the Visual Basic Object Browser that lists the properties, methods, and events you can view.

menu bar

The Visual Basic programming tool that provides access to most of the commands that control the Visual Basic programming environment. You can use these menus and commands by using keyboard commands or the mouse.

Menu Editor command
The Tools menu command, with which you add menus to your programs.

Menu Editor
A graphical tool, with which you add, modify, reorder, and delete menus in your programs.

method
A special statement that performs an action or service for a particular object.

Microsoft On the Web
A Help menu command with which you open a submenu that provides access to the Internet.

minimizing a form
Resizing a form to its smallest size.

mod
A standard object name prefix used as part of standard module names.

modal form
A type of form that remains open until the user responds to a program prompt.

mode
A state of a program or computer.

module
Special files in your project that contain code and variables that you want to be accessible from all the event procedures in your program.

MousePointer
An object property that sets the shape of the mouse pointer. Both standard and custom mouse pointer styles are available.

Move
The method you use to move individual objects or a collection of objects on a form’s coordinate system.

Move First
A method that makes the first record in the Recordset object the current record.

moving windows

Changing the position of windows, the toolbox, or the toolbar in the Visual Basic development environment by clicking the title bar and dragging the element to a new location.

MsgBox function

A dialog box function that displays output. MsgBox takes one or more arguments as input. You can assign the value returned (the results of the function call) to a variable.

multiple-condition decision structure
A decision structure that contains two or more conditional expressions.

Name
A property that creates an object name, which distinguishes modules in the program code and in the Project window.

nested loop

A loop that is embedded in another loop. In a nested Do loop, for example, the nested (inner) loop executes its commands repeatedly until it is finished; then, the other loop continues executing its commands.

New Project

The File menu command you click to open a new Visual Basic project. When you click this command, these elements of the Visual basic programming environment appear:

® Menu bar.

® Toolbar.

® Visual Basic toolbox.

® Form window.

® Properties window.

® Project window.

® Immediate window.

® Form Layout window

NewPage
A printer method that starts a new page in a print job.

NoMatch
A Recordset property, which is set to True if no match is found in the search.

nonmodal forms
Forms that the user can switch away from without having to respond to a prompt. (Also known as *modeless forms*.)

Object Browser
A utility program that you can use to explore the wide variety of objects in Windows-based applications.

object name prefix

Standard, three-letter prefixes that programmers use to identify the type of object the object names. These prefixes generally form the first part of the object name. For example, the name cboEnglish applies to a combo box that displays information in English.

object
A type of user interface element you create on a Visual Basic form by using a toolbox control.

.OCX file

An executable file that remains in storage until a program calls it. Unlike DLL files, however, OCX files inform users of the status of a file.

OLE control
A Visual Basic control that creates an OLE (Object Linking and Embedding) object, with which you can transfer and share information between applications.

On Error
A program statement used to detect a runtime error. The On Error statement sets an event trap by telling Visual Basic where to branch if it encounters an error.

online Help system
The online user assistance system.

Open dialog box
A dialog box that appears when the user clicks the Open command on the File menu. Open is one of the five dialog boxes you can create with the Common Dialog object.

Open Project
The File menu command you click to open an existing Visual Basic project.

Open statement

A general-purpose statement used to open files. The Open statement has this syntax:

```
Open pathname For mode As #filenumber
```

where:

pathname is a valid Windows pathname.

mode is a keyword indicating how the file will be used (*Input* for reading the file, *Output* for adding to the file).

filenumber is an integer from 1 through 255, a number that is used to identify the file in program code.

operator precedence
The order in which Visual Basic executes operations in an expression.

operators
In programming, symbols or other characters that indicates an operation that acts on one or more elements. Visual Basic uses arithmetic, logical, and Boolean operators.

Output
A keyword used to indicate the mode (how the file will be used) in the Open statement. Using the Output keyword indicates that you want to add to the file that you're opening.

parameter
A value that is being passed into a procedure from another location. You can determine the type of the parameter by examining the procedure declaration statement at the top of the procedure.

parent form
In a hierarchical (branching) data structure, a form that is one level higher (closer to the root) than the related child form. (*See also*: child form.)

pathname
In a hierarchical (branching) filing system, a listing of the directories or folders that lead from the current directory to a file.

Picture
An image object property that opens a selected image file.

picture box
A Visual Basic toolbox control, with which you create image objects on a form.

pointer
An onscreen symbol, such as an arrow or I-beam, that is controlled by a mouse or other pointing device.

Print dialog box
One of the five types of dialog boxes you can create with the Common Dialog object.

PrintForm
A method that you can use to send the entire contents of one or more forms to the printer.

program code
The instructions you write using the statements, functions, and special characters of the Visual Basic programming language.

program crash
An unexpected event that occurs at runtime that halts execution of a Visual Basic program. (Also known as a *runtime error*.)

program flow
The order in which Visual Basic executes commands in your programs. You control program flow by using standard Visual Basic commands in the program code.

program statements

An instruction in program code, the smallest units of executable code in a program. Generally, each line in a program is considered an individual statement.

programming tools
The complete set of programming tools that help you construct your Visual Basic programs.

Project Explorer
Another name for the Project window, where you can examine and organize the components of your Visual Basic application.

Project window
The part of the Visual Basic development environment that helps you switch back and forth between windows and tools as you work on a project.

property
A value or characteristic held by a Visual Basic object, such as Caption or ForeColor. Properties can be set at design time by using the Properties window or at runtime by using statements in the program code.

Properties window

The Visual Basic window that you use to change the characteristics (property settings) of the user interface elements on a form.

public variables
Variables that maintain their values throughout all the event procedures of your program.

Public
The keyword you use to create a public variable in a standard module.

Quick Note
A simple note-taking utility, with which you can take notes and then save them to disk.

Quick Watch
A button on the Debug toolbar, which you can use to watch the content of the program code change as it executes, line by line.

ReadOnly
When set to True, this data object property allows users to view but not modify database information. To allow users to make changes to a database, use the Properties window to set the ReadOnly property to False.

Recordset
The part of the database you're currently working with. (A table, for example.) You can adjust the Recordset by using the data object's RecordsetType property.

RecordsetType

A Recordset (data object) property, which you can set to:

® 0 for a table

® 1 for a dynaset

® 2 for a snapshot

RecordSource
A data object property that you use to name the database table that is the source of your data.

remainder division

An advanced mathematical operation, whose result is the remainder of a division operation. For example, 17 Mod   3 = 2, because the quotient 17/3 has a remainder of 2.

removing files

Removing individual files from a project by using commands on the Project menu. The changes that you make will be reflected immediately in the Project window.

resizing windows
Enlarging or shrinking the size of a window by clicking and dragging it along the border. You resize a window to make elements of the programming system visible and accessible.

Restart

A toolbar button and command on the Run menu that runs a program currently in break mode.

Resume
A keyword used in an error handler, which returns control to the statement that caused the error.

Resume Next
Keywords used in an error handler, which returns control to the program statement *following* the one that caused an error.

retry period

A specified period of time, during which a program will try repeatedly to execute a command, and after which, the program will continue.

Rnd
A function that generates a random number greater than or equal to 0 and less than 1.

runtime
The time during which a program runs. You can change some Visual Basic properties during runtime.

runtime error
Any error that forces a program to stop running.

runtime properties
Object properties that you can set while the program runs. You set runtime properties with program code.

runtime values
Variable values that you can set while the program runs. You set runtime values with program code.

Save As dialog box

One of the five types of dialog boxes you can create by using with the common dialog object. The Save As dialog box gets the drive, folder name, and filename for a new file.

Save project
The File menu command that you use to save your current project.

Save Project As
The File menu command that you use to save your current project with a different name, file location, or both.

scope
The extent or range that a variable or procedure holds its value.

scrollable
A user interface element that includes too much information to be visible at once. Such elements include vertical or horizontal scroll bars.

Seek
A Recordset (data object) method used to search for a database record. To compare the search string to text in the database, use comparison operators.

Select Case
A type of decision structure, in which the results of a test case controls the direction of program flow.

selection criterion
A test made in an If...Then or Select...Case decision structure to locate or filter information.

Sgn
A function that returns a value of:

® -1 if *n* is less than zero.

® 0 if *n* is equal to zero.

® +1 if *n* is greater than zero.

Shape
A toolbox control that helps you define the shape, fill color and style, border color, and visibility of an image.

shortcut keys
Key combinations that the user can press to run a command without opening a menu.

Show

A method you can use to display a loaded form. Show method syntax looks like this:

```
formname.Show mode
```

where

formname is the name of the form.

mode is: 0 for nonmodal forms (the default).
1 for modal forms.

Show Next Statement

A command on the Debug menu (available in break mode) that moves the cursor to the line that will execute next.

ShowColor
A method used to display the Color dialog box.

ShowFont
A method used to display the Font dialog.

ShowOpen
A method used to display the Open dialog box.

ShowPrinter
A method used to display the Print dialog box.

ShowSave
A method used to display the Save As dialog box

Sin
A function that returns the sine of the angle *n*. The angle *n* is expressed in radians.

single-condition decision structures
Decision structures that contain only one conditional statement.

single-precision floating point

A data type that requires 4 bytes to store a single variable and that returns values that range from –3.042823E38 through 3.402823E38.

sizing pointer
A mouse pointer shape used for sizing objects in a window or form.

Source
A parameter used in drag-and-drop operations. Source identifies the object that has been dropped or moved over.

splash screen
An introductory or startup graphic, usually the first image you see when you open an application.

Sqr
A function that returns the square root of *n*.

standard module
A special .bas file that you can open in your project. Standard modules contain the public variables and procedures you want to make accessible to all parts of your program.

Step Into
The Debug menu command that helps you isolate program code errors. With the Step Into button on the Debug toolbar, you can watch while Visual Basic executes program instructions one at a time.

Stop
A program statement that stops program execution and makes the Code window appear.

Str
A function that converts a number to a numeric string value.

string concatenation

An advanced mathematical operation that joins the contents of two or more strings into a single string by using the & operator. Concatenating the strings "Thank you, " and "good night." yields the string "Thank you, good night."

Sub procedure
A procedure that:

® Performs useful work in a program.

® Doesn’t return a value associated with its name.

® Can return many values.

syntax
The rules of construction that you must use when you build a program statement.

syntax error
A runtime error created by a mistake in the rules of Visual Basic programming.

tag
A piece of descriptive text stored in the Tag property that can be used to identify the state of a particular object.

Tag property
An object property, with which you create an identification note (tag). Programmers typically use this property to identify the object that the program works with in an event procedure.

Tan
A function that returns the tangent of the angle *n*. The angle *n* is expressed in radians.

taskbar

The part of the Windows operating system that you use to switch between Visual Basic forms as your program runs and to activate other Windows-based programs. You can find the taskbar along the bottom of the screen.

test case
The key variable, property, or other expression in a decision structure.

testing a program
The process of checking a program against a variety of real-life operating conditions to determine whether it works correctly.

TextBox
A Visual Basic control that displays text and receives text as user input.

text files
Files containing unformatted numbers, words, or characters.

timer
A Visual Basic control that serves as an invisible stopwatch. The Timer control gives your time-related programs access to the system clock.

toolbar

A collection of buttons that serve as shortcuts for executing commands and controlling the Visual Basic development environment.

toolbox controls
Special tools that you use to add user interface elements to a form.

Tools menu
A menu on the Visual Basic menu bar containing commands that let you start the Menu Editor and customize your program.

top measure
The distance of an object below the form's origin (upper left-hand corner), measured in twips.

twip

The basic unit of measurement used to indicate an object's position on Visual Basic forms. A twip is 1/20 of a printer's point (1/1,440 inch). In the Visual Basic coordinate system, rows of twips are aligned to the x-axis (horizontal axis), and columns of twips are aligned to the y-axis (vertical axis). You can define locations in this coordinate system by identifying the intersection of a row and a column with the notation (x, y).

type declaration
A symbol or character used to declare (define) a data type in a program.

Unload

A statement that removes a form from memory and frees up the RAM used to store the objects and graphics on the form.

unloading
Removing a program or other information from memory.

Until
A keyword you can use in Do loops to repeat commands *until* a certain condition is true.

Up Level button
Part of the Open Project dialog box that you click to search for files at higher levels in the file directory.

user interface form
The default form and standard grid that appears when you start Visual Basic.

Val
A function that converts a numeric string value to a number.

variable
A special container that temporarily holds data in a program. The values that variables return can be numbers, names, or properties.

variant
A variable that can hold data of any size or format.

.VBP
The filename extension of a Visual Basic project file.

View Code button
A Project window button that you use to open the Code window and read program code.

View Object button
A Project window button that you use to open a form or other interface object.

Visual Basic development environment
The complete set of programming tools and windows, with which you construct your Visual Basic programs.

Visual Basic for Applications

An enhanced version of Visual Basic designed especially for application macros. For more information about Visual Basic for Applications in Microsoft Word, read *Microsoft Word 97 Visual Basic Step by Step* (Microsoft Press), by Michael Halvorson and Chris Kinata.

.WMF file
A type of resizable graphic image that takes up very little disk space and has a .wmf filename extension.

Watch window
A window in the Visual Basic development environment used to debug applications. You can display the content of one or more variables by using the Watch window.

x, y coordinate system

In the Visual Basic coordinate system, rows of twips are aligned to the x (horizontal) axis, and columns of twips are aligned to the y (vertical) axis. To define locations in the coordinate system, you identify the intersection of a row and column with the notation (*x*, *y*).

zoom in
To view a smaller part of an image in greater detail.

zoom out
To view a larger part of an image in less detail.

Welcome to *Learn Microsoft Visual Basic 6.0 Now*. This CD-ROM–based training application teaches you how to program in Visual Basic 6.0.

Course Content

The course content is organized into the following chapters:

Chapter 1: Writing Your First Program

This chapter introduces you to the features and capabilities of the Visual Basic 6.0 Learning Edition programming system. In this chapter, you'll learn how to:

® Explore and configure the Visual Basic development environment.

® Build your first program.

® Create an executable file.

Chapter 2: Working with Controls

This chapter introduces the standard Visual Basic toolbox controls and teaches you how to use them to build useful interface features. You will learn how to:

® Name Visual Basic objects.

® Use basic controls to display text and process input.

® Use file system controls to browse the files and folders on your computer.

® Use data input controls to display lists and check boxes.

® Use Data and OLE controls to work with Microsoft Office applications.

® Install ActiveX controls.

Chapter 3: Working with Menus and Dialog Boxes

This chapter focuses on processing input from menu commands and dialog boxes. In this chapter, you will learn how to:

® Add menus to your programs by using the Menu Editor.

® Process menu choices by using program code.

® Use a common dialog object to display standard dialog boxes.

Chapter 4: Visual Basic Variables and Operators

This chapter describes Visual Basic variables and operators and tells you how to use specific data types to improve program efficiency. In this chapter, you will learn how to:

® Use variables to store information in your program.

® Work with specific data types to streamline your calculations.

Chapter 5: Controlling Flow and Debugging

This chapter describes how to write conditional expressions that control the flow of program code, and teaches you how to use Visual Basic's debugging tools to track down software defects. In this chapter, you will learn how to:

® Understand and use the principles of event-driven programming.

® Use conditional expressions, decision structures, and mathematical operators to control the order in which your program executes commands.

® Find and correct errors in your programs.

Chapter 6: Using Loops and Timers

This chapter teaches you how to write repeating statements (loops) in program code and how to use the Timer control to create clocks and other time-related utilities. In this chapter, you will learn how to:

® Write the For…Next loop to run a block of code a set number of times.

® Write the Do… loop to run a block of code until a specific condition is met.

® Use the Timer control to create a digital clock and other special effects.

Chapter 7: Working with Forms, Printers, and Error Handlers

This chapter shows you how to build more sophisticated and robust programs by adding forms, printer support, and error-processing to your user interface. In this chapter, you will learn how to:

® Add extra forms to the user interface.

® Use the Printer object to send output to a printer.

® Create error handlers to manage runtime errors.

Chapter 8: Adding Artwork and Animation

This chapter introduces you to graphics programming and teaches you how to enhance your program user interface with artwork and special effects. In this chapter, you will learn how to:

® Create basic artwork with the Line and Shape controls.

® Add Drag and Drop support to your user interface.

® Create animation effects with the Timer control.

Chapter 9: Working with Modules and Procedures

This chapter teaches you how to create a special file in your project called a standard module. Standard modules contain code and variables that you want to be accessible from all the event procedures in your program. You'll find that standard modules are especially useful in larger programming projects. In this chapter, you will learn how to:

® Create a standard module for code you use often.

® Use public variables to share important information.

® Create your own Function and Sub procedures.

Chapter 10: Working with Text Files and Databases

This chapter describes how you can open and modify text files and databases on your system. In this chapter, you will learn how to:

® Display a text file with the TextBox control.

® Create a new text file.

® Open a database with the Data control.

® Search, add, and delete database records.

Chapter 11: Connecting to Microsoft Office

This chapter shows you how to explore the application objects in your system and control Microsoft Office applications with Automation. In this chapter, you learn how to:

® View application objects with the Object Browser.

® Control Microsoft Office applications such as Microsoft Word from your programs.

Chapter 12: Creating Your Own Objects

This chapter is an introduction to creating your own objects in a Visual Basic program by using a special file in your project called a class module. At design time, class modules contain the definition of a class in the form of its property, method, and event procedures. At run time, a class module is transformed into an object whose members (that is, properties, methods, and events) determine how that object can behave in a program. In this chapter, you will learn how to:

® Understand and use OOP terms.

® Create a class module.

® Create a Calculator object from a class module.

® Reuse the Calculator object’s properties and methods.

Labs

This course includes an extensive assortment of labs that provide hands-on experience with the skills you develop in the chapters. Each lab consists of two or more exercises that focus on how to use the information contained in the associated chapter. At any time, you can review the solution code to understand the approach taken by the author.

Other Information

You can use *Learn Microsoft Visual Basic 6.0 Now* as more than a tutorial. It contains a variety of building blocks and tools to assist you in creating solutions. Within each chapter, the **Related Information** button provides jumps to specific chapters in this course. The glossary makes it easy to find definitions for new terms and concepts. And the Visual Basic Help system provides an online reference that you can use to learn more about the Visual Basic development environment and programming language.

You can gain quick access to the following types of information directly through the Table of Contents:

Information type	Description
Labs	Labs provide step-by-step guides you can use to practice new skills.
Multimedia	Multimedia elements include expert point-of-view videos that discuss features; demonstrations that show how to perform tasks discussed in the text; and animations that illustrate a technical concept.
Self-Check Questions	Review questions are included at the end of each chapter so that you can test your understanding of the material in the chapter.

To view an animation introducing this chapter, click this icon.
{ewc mvimg, mvimage, !anim.bmp}

This chapter introduces you to a special file called a class module, which is used to create objects in a Visual Basic program. At design time, class modules contain the definition of a class in the form of its property, method, and event procedures. At run time, a class module is transformed into an object whose members (that is, properties, methods, and events) determine how that object can behave in a program.

The **object-oriented programming** methodology (hereafter referred to as OOP) has several advantages over other approaches to creating applications. Research has shown that, on average, OOP applications:

® Are composed of a greater number of reusable modules/objects.

® Have fewer bugs.

® Are easier to maintain and enhance.

In this chapter, you will learn how to:

® Understand and use OOP terms.

® Create a class module.

® Create a **Calculator** object from a class module.

® Reuse the **Calculator** object's properties and methods.

Object Oriented Programming has become very popular in the last few years. The Visual Basic programming language, like most of Microsoft's programming tools and applications, is itself based on OOP principles. When you write a Visual Basic program using OOP principles, you are programming just like the big boys do with Visual C++.

To view an animation that introduces Object Oriented Programming principles, click this icon. {ewc mvimg, mvimage,!anim.bmp}

Like any new methodology, there are terms specific to OOP that, once you grasp them, help you to understand the concepts and the process. What follows is a brief primer on OOP terminology, a list of important terms and concepts.

This section includes the following topics:

® Class

® Objects and Instantiation

® Member

® The Public Interface and Messaging

® Encapsulation

® Polymorphism

® Inheritance

A class is a formal category that defines the possible settings of an object's properties, the methods that govern an object's behavior, and the events to which an object can respond. The formal specification of an object's properties, methods, and events (generically referred to as *members*) is done only once when the class is created. In Visual Basic, you create this class with a class module.

Note All class modules have the .cls file name extension.

A commonly used metaphor for a class is that it is like a cookie-cutter that is used to create a cookie object. The dough from which the cookie object is cut by the cookie-cutter class consists of all the Visual Basic keywords and identifiers used in the class module. Examples of classes are human being, male, and female.

Classes can be organized into hierarchies: Male and female can be considered subclasses of the base class human being, or girl and woman can be considered subclasses of the subclass female. In Microsoft's terminology, a class and a type are synonymous.

An object is an example or instance of a particular class. When a class is instantiated, an object or instance of that class is created whose properties are initially set to certain default values. The settings of these properties can change during the life of the object, but only within the range defined by the object's class. An object is like a cookie that emerges from dough acted on by a cookie-cutter class or design pattern.

Examples of objects are Tom and Dick (of the male class) or Susie and Mary (of the female class). Tom has properties like HairColor and Age, whose values/settings change over the years. An object’s behavior is defined by its methods and events. Events in Tom’s or Susie’s life are birth and death (analogous to Visual Basic’s Load and Unload events) and methods are walking and running (analogous to Visual Basic’s Move method).

Instantiation is the process by which an object is created from a class. In Visual Basic, you instantiate an object:

® Either by using the **Dim As New** mode of declaration.

® Or by using the **CreateObject** function.

You will learn the details of how to instantiate objects in Instantiating a Calculator Object.

Note Previously in this course, we’ve discussed the standard control objects that Visual Basic offers for you to reuse. As a result, you’ve probably gotten used to thinking of an “object” as having a visual interface. However, many useful objects (including the ones you can create yourself with Visual Basic) don’t have a visual interface.

A member is an element of an object. It is a generic term that is equivalent, in Visual Basic's implementation of OOP, to a property, method, or event. At the code level of abstraction, a member consists of a procedure. There are several kinds of procedures that you can write in a class module to create an object’s members:

- Property Get, Property Let, and Property Set procedures, which define the characteristics of an object.
- Function or Sub procedures (referred to generically as methods in OOP terminology), which act on an object and cause it to behave in some manner.
- Event procedures, which are a special kind of Sub procedure that can react to actions taken by the user of your program (clicking the mouse, pressing a key, and so on).

The public interface of a class is the information about the class that is publicly available. A class or object instance of a class is sometimes referred to as a "black box," which hides most of the details of its implementation. What you can know about a class (that is, its public interface) is revealed, in Visual Basic's implementation of OOP, via the Object Browser.

The major elements of the public interface of a class module are its public procedures. When you write a procedure in a class module, you can specify in the procedure's declaration that it is to be one of two kinds:

- **Public**: This keyword indicates that the procedure is accessible to be called and reused by all other procedures in all modules in the project.
- **Private**: This keyword indicates that the procedure is accessible only to other procedures in the class module where it is declared.

Other important elements of the public interface of a class module are:

- A public procedure's arguments (required or optional), which specify the kind and range of values that can be passed to the procedure.
- A public procedure's return values, which specify the result of a Function method or the setting of a property.

Note If a class module has a Windows Help file associated with it, the information in the Help file can be thought of as an extension of the public interface of the class.

A programmer reuses an object instance of a class by sending messages to and receiving messages from elements of its public interface. Messaging is the process by which objects interact with each other. A message to an object is a request that a property be set/returned, a method be carried out, or an event be raised. A message from an object is a return value that says, in effect, "I did what you told me and here's the result."

To send a message, you specify the name of the object, the name of one of its members, and the parameters specified by the member's arguments (required or optional). For example, the following program statement sends a message to a **Form** object's public **Move** method, telling it to move the form to the center of the screen:

```
Form1.Move (Screen.Width - Form1.Width) * 0.5, _
           (Screen.Height - Form1.Height) * 0.5
```

Encapsulation is a key principle of the OOP methodology. It refers to the programmer’s ability to create a class that contains and hides information about an object, such as data structures and code. Encapsulation isolates the internal complexity of an object's operation from the rest of the application. If an object is properly encapsulated, it is said to be a "black box" that reveals just the information needed to reuse its members through its public interface and that hides any private implementation details.

An encapsulated object ideally should also be self-contained and not be coupled to or dependent upon other objects, except as defined by the encapsulated object’s public interface. What this means on a practical level is that, when you program using the OOP methodology, you don’t “cheat” by using Public variables. A Public variable, by definition, violates the OOP principle of encapsulation.

If two or more class modules have public members (properties, methods, or events) that share the same name, public interface, and basic purpose but that are implemented differently, such members are said to be polymorphic or to comply with the OOP principle of polymorphism. A programmer can call or invoke a polymorphic method for any object that has it as a member, without worrying about the type of object involved.

For example, Visual Basic's Move method is polymorphic because:

- ® It belongs to over 20 different classes/objects.
- ® When you send a message to a form object's **Move** method, you pass it the same number of arguments with the same data types as when you send a message to a picture box object's **Move** method.

The following statements use the **Move** method to center a form on the screen and a picture box on a form:

```
Form1.Move (Screen.Width - Width) * 0.5, _
           (Screen.Height - Height) * 0.5

Picture1.Move (Form1.ScaleWidth - Picture1.Width) * 0.5, _
              (Form1.ScaleHeight - Picture1.Height) * 0.5
```

In both cases, the public interface of the **Move** method (that is, its arguments and syntax) is the same:

Object.Move *Left*, [*Top*]

where

Object is the **Name** property of the object to which the Move message is being sent.

Left is a required argument that specifies the value for the left edge of *Object*.

Top is an optional argument that specifies the value for the top edge of *Object*.

Note Optional arguments of an object's member like *Top* are indicated in the Object Browser (F2 key) by the use of brackets.

Designing and using polymorphic methods has these advantages:

- ® It simplifies the public interface to a set of related objects (that is, a class library) by minimizing the number of its methods. Instead of a **MoveForm** method and a **MovePictureBox** method, there is a single **Move** method.
- ® It hides/encapsulates low-level, complex implementation details from the programmer who is trying to reuse the object's method. Obviously, the code that centers a form on a screen is different from the code that centers a picture box on its container object. But only the programmer who wrote Visual Basic's **Move** method had to worry about those gory details. Reusers of the **Move** method only have to know how to send a message to the **Move** method's simple, polymorphic public interface.

Inheritance is an OOP principle that allows the members of a class to be passed from a base class to one or more descendant classes. The way in which inheritance is implemented in different languages varies. Most of the time, however, it includes the ability to “inherit” the source code from a base class in order to modify or remove some of the members of the base class.

Visual Basic doesn’t currently support inheritance in the purest sense of the term, providing only interface inheritance at this time (through its Implements statement). Some OOP "purists" eagerly await the availability of inheritance in Visual Basic, but its absence isn't a limitation for most programmers.

The first step in object oriented programming with Visual Basic is creating a class module in your project to hold the definition of your new object. By writing various kinds of procedures in this file, you can create a class that can be used over and over again within your programs, saving you time and money.

Creating a new class module is easy. From the **Project** menu, click **Add Class Module**, and then double-click the Class Module icon in the **Add Module** dialog box.

In this section, topics include:

® Class Module Files

® Class Module Properties

® Class Module Events

Visual Basic stores class modules in their own folder in the Project window. The first class module in a program is named Class1. However, when you save the module to disk, you can change the file name with the **File** menu **Save Class1 As** command.

Note All class modules have the same default .cls file name extension.

To view an illustration of a new class module in the Project window, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

When you create a new class module, it appears immediately in the Code window, as this illustration shows. To view the illustration, click this icon.
{ewc mvimg, mvimage,!illust.bmp}

You can load class modules you create into other projects, just as you do with forms and standard modules. To load class modules, start from the **Project** menu, and then click **Add File**. You can also remove class modules from a project by following these steps:

1. Select the class module's name in the Project window.
2. Right-click it to display its short-cut menu.
3. Click **Remove**.

To view a demonstration that shows you how to create a class module named Calculator in a Visual Basic project, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

When you add a new class module to a standard project, it has three properties listed in the Properties window. Names and descriptions of these properties are in the following table:

Property	Description
Name	Returns the name used in code to identify the class.
DataBindingBehavior	Sets whether the class will act as a data consumer (possible values are 0 – vbNone, 1 – vbSimpleBound, and 2 – vbComplexBound).
DataSourceBehavior	Sets whether the class will act as a data source (possible values are 0 – vbNone and 1 – vbDataSource).

Generally, it is a good idea to assign your class module a descriptive name with the Name property, so that it is easy to reference in program code.

Note In most cases, you can create functional classes/objects without changing the default values of its last three properties. Also, don't confuse the private, intrinsic properties of a class module with its public properties that you create.

Class modules have two events associated with them:

- **Initialize**.
- **Terminate**.

The **Initialize** event occurs when an object instance of a class module is created. You can use this event to initialize any data used by the instance of the class. The **Terminate** event occurs when all references to an instance of a class are removed from memory. You can use this event to do cleanup work (close data files used by the object, release system resources used by object references, and so on).

Note   In many cases, you do not need to write code in a class module’s Initialize and Terminate event procedures.

Let’s take what you’ve learned so far about class modules and objects and put it into practice with a simple example. In the last section, we created a new class module named Calculator in a Visual Basic project. In this section, we’ll define property procedures and methods that prepare the **Calculator** object to perform multiplication and division operations in a program. We'll follow these steps:

1. Write two Property Let procedures that set the values to be multiplied or divided.
2. Write two methods called **Multiply** and **Divide** that carry out the functions of the **Calculator** object and return the results.

Now it’s important to understand that you could write code to perform multiplication and division in the traditional way (that is, create **Multiply** and **Divide Function** procedures that each take two arguments and return the results). However, in this case we’re going to perform these tasks using the OOP methodology.

This section includes the following topics:

After you create a class module in a project, you can write Property Let procedures in the class module to create the object's properties and set their values. Let's assume you have a class module named Calculator in a standard Visual Basic project. We will now write Property Let procedures to create the object's properties and set their values. We'll name these two properties **Value1** and **Value2** and write the following code:

```
Public Property Let Value1(Setting)

   ' Store setting in module-level variable.
   mVal1 = Setting

End Property

Public Property Let Value2(Setting)

   ' Store setting in module-level variable.
   mVal2 = Setting

End Property
```

The syntax of Property Let procedures should be pretty easy for you to understand after your exposure to Function and Sub general procedures earlier in the course. However, you should note these points about the preceding code:

- The **Public** keyword is used in the declaration of the procedures so that these properties are accessible to code outside of the class module. (The opposite of **Public** is the **Private** keyword, which specifies that the procedure is accessible only to code within the class module.)
- The *Setting* arguments' data types are Variant, which allows numbers of all sizes to be passed into the properties.
- The statement in each Property Let procedure takes the value passed into the *Setting* argument and assigns it to a module-level variable. To understand why this is done, see the next section titled Calculator Module-Level Variables.

When you set a property of a Visual Basic control, the value of the property should remain the same (or persist) as long as your program continues to run or until you set it to a different value. Up until now, you've taken this behavior of properties for granted and never given it a second thought.

However, now that you're writing your own objects with properties, you have to worry about such implementation details as how to store the values of properties. One way you could do this is to write the value to a file, but it's easier and quicker to store a property's value in a module-level variable in the Declarations section of the Calculator class module. You can do this with the following code:

```
' Module-level variables to hold property values:
Private mVal1
Private mVal2
```

You should note these points about the preceding declarations:

® The **Private** keyword is used in the declaration of the module-level variables. This is done because the details of how the **Calculator** object's properties are implemented and work is nobody's concern except the object's programmer.

® The module-level variables *mVal1* and *mVal2* are prefixed with an *m* to signify their scope. Again, this is done strictly for the convenience of the object's programmer.

So now you know how the values of the **Calculator** object's two properties are set and stored. Now let's write its **Multiply** and **Divide** methods.

In our simple **Calculator** object example, we'll only write two methods (**Multiply** and **Divide**). The code for the methods follows:

```
Public Function Multiply()
    Multiply = mVal1 * mVal2
End Function

Public Function Divide()
    If mVal2 <> 0 Then
        Divide = mVal1 / mVal2
    Else
        Divide = "Undefined"
    End If
End Function
```

Because these two methods return a value, we write them as Function general procedures. What these procedures do is:

® Read the current settings of the **Value1** and **Value2** properties (stored, remember, in the module-level variables mVal1 and mVal2).

® Perform either multiplication or division on the values.

® Return the results to any procedure that calls and reuses the methods.

Note The **Divide** function uses an If structure to skip the division if the denominator passed to the function (mVal2) is zero. This would produce an undefined result, so the function returns the "Undefined" string instead. Your functions should also provide this type of basic error handling.

To view a demonstration that shows you how to create properties and methods in a class module, click this icon. {ewc mvimg, mvimage,!anim.bmp}

Now you’ll look at the other side of the coin and learn how to reuse an object’s properties and methods (that is, its members) in the event procedures associated with a Visual Basic form. In this section, we’ll write code in a form in the **Calculator** object’s project to:

1. Initialize the **Calculator** object.
2. Set its **Value1** and **Value2** properties.
3. Call its **Multiply** and **Divide** methods and display the results.

This section includes the following topics:

Let’s assume that you have a form in the **Calculator** object’s project. To reuse the **Calculator** object’s members, you need to first create an object instance of the Calculator. As we mentioned earlier in the chapter, you can instantiate an object using:

® Either the **Dim As New** mode of declaration (the easiest way).

® Or calling the **CreateObject** function (used mostly when instantiating **Automation** objects).

We’ll use the **Dim As New** mode by writing the following statement in the Declarations section of Form1:

```
' Instantiate a calculator object.
Dim Calculator1 As New Calculator
```

In the previous statement, *Calculator1* is a variable that is assigned an object instance of the *Calculator* class. After instantiating the object, you’re now ready to reuse its methods.

Note You can use any naming scheme you want to for a variable that is assigned an object instance. In this example, however, we follow Visual Basic’s approach of naming the first object with the suffix *1*.

For the code that follows, let’s assume that there are two text boxes on Form1 in the **Calculator** object’s project: **Text1**, that receives the first number in the equation, and **Text2**, that receives the second. In addition, let's assume that **Option1** and **Option2** are a pair of option buttons that allow the user to switch between multiplication and division operations, and that there is a command button (**Command1**) that directs the program to compute the result. Finally, let's assume that there is a label object (**Label1**) that can display the result of the calculation.

When the user types numbers in the first two text boxes and click the command button, the program needs to set the **Value1** and **Value2** properties of the **Calculator** object with the following program code:

```
'set values of object's properties
Calculator1.Value1 = Text1.Text
Calculator1.Value2 = Text2.Text
```

Then it’s an easy matter to call the object’s **Multiply** and **Divide** methods and display the results:

```
'if first button clicked, multiply
If Option1.Value = True Then
    Label1.Caption = Calculator1.Multiply
End If
'if second button clicked, divide
If Option2.Value = True Then
    Label1.Caption = Calculator1.Divide
End If
```

As you can see, once you create an object in a class module, it is a simple matter to put it to work. Although this is a simple example, you can use these same techniques to build more complex objects that will save you considerable development time in the future. Give it a shot, and be sure to leverage all the skills you've learned in the preceding chapters!

To view a demonstration that shows you how to use a custom object's properties and methods in a program, click this icon.
{ewc mvimg, mvimage,!anim.bmp}

1. Some advantages of OOP applications are:

{ewc mvimg, mvimage, !answer.bmp} A. They are easier to debug.

{ewc mvimg, mvimage, !answer.bmp} B. They contain more reusable modules.

{ewc mvimg, mvimage, !answer.bmp} C. They are easier to maintain.

{ewc mvimg, mvimage, !answer.bmp} D. All of the above.

2. What default file name extension is assigned to class modules?

{ewc mvimg, mvimage, A. .frm

!answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} B. .bas

{ewc mvimg, mvimage, !answer.bmp} C. .vbp

{ewc mvimg, mvimage, !answer.bmp} D. None of the above.

3. Which of the follow items is not an object's member:

{ewc mvimg, mvimage, !answer.bmp} A. A property

{ewc mvimg, B. A method.

mvimage, !answer.bmp}

{ewc mvimg, mvimage, !answer.bmp} C. An event.

{ewc mvimg, mvimage, !answer.bmp} D. A module-level variable.

4. What does the OOP principle of encapsulation do?

{ewc mvimg, mvimage, !answer.bmp} A. Hides the implementation details of an object from the object's reuser.

{ewc mvimg, mvimage, !answer.bmp} B. Isolates the internal complexity of an object from the object's reuser.

{ew C. Prevents the object from being coupled to and dependent upon another c mvimg, mvimage, !answer.bmp} object.

{ewc mvimg, mvimage, !answer.bmp} D. All of the above.

5. Which of the following is an event of the class module?

{ewc mvimg, mvimage, !answer.bmp} A. **Click**.

{ewc mvimg, mvimage, !answer.bmp} B. **Load**.

{ewc mvimg, mvimage, !answer C. **Initialize**.

.bmp}

{ewc mvimg, mvimage, !answer.bmp} D. **Unload**.

6. Before a programmer can reuse an object's members, he/she must:

{ewc mvimg, mvimage, !answer.bmp} A. Inherit the class' source code.

{ewc mvimg, mvimage, !answer.bmp} B. Encapsulate the members' code.

{ewc mvimg, mvimage, !answer.bmp} C. Instantiate the object.

{ewc mvimg, mvimage, D. Set the object's properties.

!
ans
wer
.bm
p}

In this lab, you will modify and reuse the **Calculator** class that you explored in Chapter 12. You will:

® Open a project with an existing class module and write a new method in it called **Hypotenuse**.

® Instantiate the revised **Calculator** class in the Declarations section of Form1.

® Call the **Calculator1** object's **Hypotenuse** method, and display the result.

To see a demonstration of the lab solution, click this icon.
{ewc mvimg, mvimage,!democlip.bmp}

Estimated time to complete this lab: **20 minutes**

Objectives

After completing this lab, you will be able to:

® Add a new method to an existing class module.

® Instantiate an object instance of the class.

® Call the object's method.

Lab Setup

In this lab, you'll modify the project Lab12.vbp in the \LVB6\Ch12 folder on your hard disk. This project is almost identical to the Calculator.vbp project that you worked with earlier in this chapter. (I simply added a hypotenuse option button to the form, and changed the name to protect the original copy.)

Exercises

® Exercise 1: Revise an Existing Class Module

In this exercise, you open a project with a class module in it and write a new method for it called **Hypotenuse**.

® Exercise 2: Instantiate the Calculator Object

In this exercise, you verify that the **Calculator** object is called and instantiated in the Declarations section of Form1.

® Exercise 3: Call the New Method

In this exercise, you call the **Calculator1** object's **Hypotenuse** method, and display the result.

In this exercise, you open a project with a class module in it and write a new method for it called **Hypotenuse**.

u Open the project

1. Start Visual Basic and open the Lab12.vbp project.
2. In the Project window's Class Modules folder, select **Calculator**.
3. Click the **View Code** button to display its Code window.

u Write the new Triangulate method

1. At the bottom of the General section of the Code window, type the following Function procedure:

```
Public Function Hypotenuse()

   ' Calculate the hypotenuse of a right-angled
   ' triangle (the side opposite the right angle).
   Hypotenuse = Sqr((mVal1 ^ 2) + (mVal2 ^ 2))

End Function
```

   When you declare the **Hypotenuse** function and press ENTER, Visual Basic places it in a new section in the Code window and adds **End Function** to the bottom of the procedure.

2. To save your revised class module, click **Save Project**.
3. Close the Code window.

In this exercise, you verify that the **Calculator** object is called and instantiated in the Declarations section of Form1.

u Instantiate the object

— In the Declarations section of the Form1 module's Code window, verify that the following declaration exists:

```
' Instantiate a calculator object.
Dim Calculator1 As New Calculator
```

In this exercise, you call the **Calculator1** object's **Hypotenuse** method and display the result.

u **Edit the Command1_Click event procedure**

— Scroll to the bottom of the Command1_Click event procedure, and add the following program code to call your new **Hypotenuse** method when the third option button on the form is clicked:

```
'if third button clicked, find hypotenuse
If Option3.Value = True Then
    Label1.Caption = Calculator1.Hypotenuse
End If
```

u **Run the program and call the method**

1. Click **Start**.
2. Type **3** in the first text box (the length of the first side of the triangle).
3. Type **4** in the second text box (the length of the second side).
4. Click the **Hypotenuse option** button, and then click the **Calculate** button. The program calls the **Hypotenuse** method and displays a result of 5, the length of the missing third side (the hypotenuse) of the right triangle.
5. To stop the program, click **End**.

Trouble? Check out the solution to the project (Lab12Sol.vbp) in the \Lvb6\Ch12 folder.

Object-oriented programming (OOP) applications are all of these things! For more information, see Chapter 12: Creating Your Own Objects.

Object-oriented programming (OOP) applications are all of these things! For more information, see Chapter 12: Creating Your Own Objects.

Object-oriented programming (OOP) applications are all of these things! For more information, see Chapter 12: Creating Your Own Objects.

Object-oriented programming (OOP) applications are all of these things! For more information, see Chapter 12: Creating Your Own Objects.

The correct answer is **.cls**. For more information, see Class Module Files.

The correct answer is **.cls**. For more information, see Class Module Files.

The correct answer is **.cls**. For more information, see Class Module Files.

The correct answer is **.cls**. For more information, see Class Module Files.

A member is an element of an object such as a property, method or event. For more information, see Member.

A member is an element of an object such as a property, method or event. For more information, see Member.

A member is an element of an object such as a property, method or event. For more information, see Member.

A member is an element of an object such as a property, method or event. For more information, see Member.

Encapsulation does all of these things! For more information, see Encapsulation.

Encapsulation does all of these things! For more information, see Encapsulation.

Encapsulation does all of these things! For more information, see Encapsulation.

Encapsulation does all of these things! For more information, see Encapsulation.

The Initialize and Terminate events are associated with class modules. For more information, see Class Module Events.

The Initialize and Terminate events are associated with class modules. For more information, see Class Module Events.

The Initialize and Terminate events are associated with class modules. For more information, see Class Module Events.

The Initialize and Terminate events are associated with class modules. For more information, see Class Module Events.

To reuse an object's members, you need to first create an object instance. For more information, see Instantiating a Calculator Object.

To reuse an object's members, you need to first create an object instance. For more information, see Instantiating a Calculator Object.

To reuse an object's members, you need to first create an object instance. For more information, see Instantiating a Calculator Object.

To reuse an object's members, you need to first create an object instance. For more information, see Instantiating a Calculator Object.

For more information, see Calculator Module-Level Variables.

For more information, see Calculator Module-Level Variables.

For more information, see Calculator Module-Level Variables.

For more information, see Calculator Module-Level Variables.

You can jump directly to the self-check questions by clicking on the title below.

Chapter 1: Writing Your First Program

You can jump directly to the self-check questions by clicking on the title below.

Chapter 2: Working with Controls

You can jump directly to the self-check questions by clicking on the title below.

Chapter 3: Working with Menus and Dialog Boxes

You can jump directly to the self-check questions by clicking on the title below.

Chapter 4: Visual Basic Variables and Operators

You can jump directly to the self-check questions by clicking on the title below.

Chapter 5: Controlling Flow and Debugging

You can jump directly to the self-check questions by clicking on the title below.

Chapter 6: Using Loops and Timers

You can jump directly to the self-check questions by clicking on the title below.

Chapter 7: Working with Forms, Printers, and Error Handlers

You can jump directly to the self-check questions by clicking on the title below.

Chapter 8: Adding Artwork and Animation

You can jump directly to the self-check questions by clicking on the title below.

Chapter 9: Working with Modules and Procedures

You can jump directly to the self-check questions by clicking on the title below.

Chapter 10: Processing Text Files and Databases

You can jump directly to the self-check questions by clicking on the title below.

Chapter 11: Connecting to Microsoft Office

You can jump directly to the self-check questions by clicking on the title below.

Chapter 12: Creating Your Own Objects

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

You can jump directly to the lab exercises by clicking on the titles below.

This demonstration shows how to use the Menu Editor to create a **Clock** menu, a **Date** command, and a **Time** command on a form. The program uses two event procedures to display system clock information with a label. To view the demonstration, click this icon:
{ewc mvimg, mvimage,!democlip.bmp}

You can view code samples from within the chapter, or you can view sample code from here by clicking the icon or title below:

{ewc mvimg, mvimage, !code.bmp}	Displaying Text With Label
{ewc mvimg, mvimage, !code.bmp}	Controlling a Text Box with Code
{ewc mvimg, mvimage, !code.bmp}	Creating a List Using Code
{ewc mvimg, mvimage, !code.bmp}	Adding Items to a Combo Box

You can view code samples from within the chapter, or you can view sample code from here by clicking the icon or title below:

{ewc mvimg, mvimage, !code.bmp}	Code for an Open Dialog Box
{ewc mvimg, mvimage, !code.bmp}	Creating a Color Dialog Box

You can view code samples from within the chapter, or you can view sample code from here by clicking the icon or title below:

{ewc mvimg, mvimage, !code.bmp}	The LastName Variable
{ewc mvimg, mvimage, !code.bmp}	Using Fundamental Data Types
{ewc mvimg, mvimage, !code.bmp}	User-defined Data Types

You can view code samples from within the chapter, or you can view sample code from here by clicking the icon or title below:

{ewc mvimg, mvimage, !code.bmp}	Multiline If... Then
{ewc mvimg, mvimage, !code.bmp}	Select Case Example
{ewc mvimg, mvimage, !code.bmp}	Select Case with Else
{ewc mvimg, mvimage, !code.bmp}	Select Case with Range
{ewc mvimg, mvimage, !code.bmp}	Entering Break Mode

You can view code samples from within the chapter, or you can view sample code from here by clicking the icon or title below:

{ewc mvimg, mvimage, !code.bmp} An Error Handler

You can view code samples from within the chapter, or you can view sample code from here by clicking the icon or title below:

{ewc mvimg, mvimage, !code.bmp}	TotalTax Function
{ewc mvimg, mvimage, !code.bmp}	Entering Names in a List

You can view code samples from within the chapter, or you can view sample code from here by clicking the icon or title below:

{ewc mvimg, mvimage, !code.bmp}	Complete Text Browser
{ewc mvimg, mvimage, !code.bmp}	Saving Text to a File
{ewc mvimg, mvimage, !code.bmp}	Searching a Database
{ewc mvimg, mvimage, !code.bmp}	Adding a Record
{ewc mvimg, mvimage, !code.bmp}	The Delete Method
{ewc mvimg, mvimage, !code.bmp}	Making a Backup with FileCopy

You can go directly to an animation or demonstration by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 1 Introduction
{ewc mvimg, mvimage,!democlip.bmp}	Running a Visual Basic Program
{ewc mvimg, mvimage,!democlip.bmp}	Moving, Docking, and Resizing Windows
{ewc mvimg, mvimage,!anim.bmp}	Creating the User Interface
{ewc mvimg, mvimage,!anim.bmp}	Properties Analogy
{ewc mvimg, mvimage,!anim.bmp}	Planning Lucky Seven
{ewc mvimg, mvimage,!democlip.bmp}	Watch Me Design the Form
{ewc mvimg, mvimage,!democlip.bmp}	Watch Me Set the Properties
{ewc mvimg, mvimage,!democlip.bmp}	Watch Me Write the Code
{ewc mvimg, mvimage,!democlip.bmp}	Watch Me Run Lucky Seven
{ewc mvimg, mvimage,!democlip.bmp}	Lab 1 Overview

You can go directly to an animation or demonstration by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 2 Introduction
{ewc mvimg, mvimage,!democlip.bmp}	Label, TextBox, and CommandButton Controls
{ewc mvimg, mvimage,!democlip.bmp}	Creating a Browser Utility
{ewc mvimg, mvimage,!democlip.bmp}	Input Controls in Action
{ewc mvimg, mvimage,!democlip.bmp}	Using the OLE Control
{ewc mvimg, mvimage,!democlip.bmp}	Lab 2 Overview

You can go directly to an animation or demonstration by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 3 Introduction
{ewc mvimg, mvimage,!democlip.bmp}	The Menu Editor
{ewc mvimg, mvimage,!democlip.bmp}	Using the Color Dialog Box
{ewc mvimg, mvimage,!democlip.bmp}	Lab 3 Overview

You can go directly to an animation, demonstration or expert point of view by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 4 Introduction
{ewc mvimg, mvimage,!exppov.bmp}	Creating Program Variables
{ewc mvimg, mvimage,!democlip.bmp}	Using InputBox and MsgBox Together
{ewc mvimg, mvimage,!democlip.bmp}	Advanced Arithmetic Operators
{ewc mvimg, mvimage,!democlip.bmp}	Lab 4 Overview

You can go directly to an animation or demonstration by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 5 Introduction
{ewc mvimg, mvimage,!anim.bmp}	Event-driven Programming
{ewc mvimg, mvimage,!democlip.bmp}	User Logon Utility
{ewc mvimg, mvimage,!democlip.bmp}	How to Use Break Mode
{ewc mvimg, mvimage,!democlip.bmp}	Lab 5 Overview

You can go directly to an animation or demonstration by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 6 Introduction
{ewc mvimg, mvimage,!democlip.bmp}	The FontSize Property
{ewc mvimg, mvimage,!democlip.bmp}	A Digital Clock
{ewc mvimg, mvimage,!democlip.bmp}	Lab 6 Overview

You can go directly to an animation, demonstration or audio clip by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 7 Introduction
{ewc mvimg, mvimage,!anim.bmp}	Using Extra Forms
{ewc mvimg, mvimage,!democlip.bmp}	Adding a Second Form
{ewc mvimg, mvimage,!audio.bmp}	Printing Text
{ewc mvimg, mvimage,!democlip.bmp}	An Error Handler
{ewc mvimg, mvimage,!democlip.bmp}	Lab 7 Overview

You can go directly to an animation or demonstration by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 8 Introduction
{ewc mvimg, mvimage,!anim.bmp}	The Steps in Drag and Drop Programming
{ewc mvimg, mvimage,!democlip.bmp}	Drag and Drop Example
{ewc mvimg, mvimage,!democlip.bmp}	The Drifting Cloud Demonstration
{ewc mvimg, mvimage,!democlip.bmp}	Lab 8 Overview

You can go directly to an animation, demonstration or audio clip by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 9 Introduction
{ewc mvimg, mvimage,!anim.bmp}	Variable Scope
{ewc mvimg, mvimage,!democlip.bmp}	Lucky Seven Upgrade
{ewc mvimg, mvimage,!audio.bmp}	Sub Procedure Syntax
{ewc mvimg, mvimage,!democlip.bmp}	A Sub Procedure Managing Text
{ewc mvimg, mvimage,!democlip.bmp}	Lab 9 Overview

You can go directly to an animation or demonstration by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 10 Introduction
{ewc mvimg, mvimage,!democlip.bmp}	Text Browser Demonstration
{ewc mvimg, mvimage,!democlip.bmp}	A Note-taking Utility
{ewc mvimg, mvimage,!democlip.bmp}	Displaying Database Fields
{ewc mvimg, mvimage,!democlip.bmp}	Lab 10 Overview

You can go directly to an animation, demonstration or expert point of view by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 11 Introduction
{ewc mvimg, mvimage,!exppov.bmp}	Using Automation
{ewc mvimg, mvimage,!democlip.bmp}	Sample Word Automation
{ewc mvimg, mvimage,!democlip.bmp}	Lab 11 Overview

You can go directly to an animation or demonstration by clicking an icon or topic title below.

{ewc mvimg, mvimage,!anim.bmp}	Chapter 12 Introduction
{ewc mvimg, mvimage,!anim.bmp}	Object Oriented Programming
{ewc mvimg, mvimage,!anim.bmp}	Creating a Class Module
{ewc mvimg, mvimage,!anim.bmp}	Defining Properties and Methods in a Class
{ewc mvimg, mvimage,!anim.bmp}	Instantiating an Object
{ewc mvimg, mvimage,!democlip.bmp}	Lab 12 Overview

You can go directly to the expert point of view by clicking the icon or title below:

{ewc mvimg, mvimage,!exppov.bmp} You Did It!

If you have Microsoft Outlook on your system, you can use Automation to send electronic mail, plan meetings, manage contacts, and other tasks. Here's how you do it:

u To add a reference to the Microsoft Outlook object library

1. From the **Project** menu, click the **References** command and select the Microsoft Outlook 8.0 Object Library.
2. Examine the features you want to use (as necessary) with the Object Browser.

This sample code demonstrates how you can use Outlook's **Send** method to place an electronic mail message in Outlook's Outbox (the temporary storage location for outgoing mail). The email names used in the sample are fictitious; if you want to use this code to send your own electronic mail, modify the email addresses accordingly. The notation in the routine has been simplified by using the With statement, which adds the expression out.CreateItem(olMailItem) to each statement beginning with a period (.).

u To write your Visual Basic program

1. In the event procedure in which you plan to use Automation, use the **Dim** statement to create an object variable and declare other variables:

```
Dim out As Object               'use out as variable name
```

2. Use the **CreateObject** function to load an **Automation** object into the object variable:

```
Set out = CreateObject("Outlook.Application")
```

3. Use the methods and properties of the **Application** object in the event procedure.

```
With out.CreateItem(olMailItem) 'using the Outlook object
    'insert recipients one at a time with the Add method
    '(these names are fictitious--replace with your own)
    .Recipients.Add "maria@xxx.com"  'To: field
    .Recipients.Add "casey@xxx.com"  'To: field
    'to place users in the CC: field, specify olCC type
    .Recipients.Add("mosha@xxx.com").Type = olCC
    .Subject = "Test Message"  'include a subject field
    .Body = Text1.Text  'copy message text from text box
    'insert attachments one at a time with the Add method
    .Attachments.Add "c:\lvb6\ch02\computer.wmf"
    'finally, copy message to Outlook outbox with Send
    .Send
End With
```

Note This sample code is designed for conditions when Outlook is already running on the Windows desktop (the default setting for most Outlook users). It doesn't start Outlook first.

To view the message placed in your Outlook Outbox when you run this routine to send mail from a Visual Basic program, click this icon:
{ewc mvimg, mvimage,!illust.bmp}

Learn Microsoft Visual Basic 6.0 Now provides numerous audio/video demonstrations, animations, and expert points of view that illustrate the concepts and techniques discussed in this course.

When you see this icon {ewc MVIMG, MVIMAGE,!democlip.bmp} in the course contents, click it to launch a demonstration of the topic being discussed.

When you see this icon {ewc MVIMG, MVIMAGE,!anim.bmp} in the course contents, click it to launch an animation of a chapter introduction.

When you see this icon {ewc MVIMG, MVIMAGE,!exppov.bmp} in the course contents, click it to launch an expert point of view.

When you see this icon {ewc MVIMG, MVIMAGE,!audio.bmp} in the course contents, click it to launch an audio file.

Learn Microsoft Visual Basic 6.0 Now contains numerous code samples.

When you see this icon {ewc MVIMG, MVIMAGE,!sampcode.bmp} in the course contents, click it to view the sample code.

This course includes a number of self-check questions at the end of each chapter. You can use these multiple-choice questions to test your understanding of the information that has been covered in the course.

To see whether you answered a question correctly, click this icon.
{ewc MVIMG, MVIMAGE,!answer.bmp}

Each answer also contains a jump to the associated chapter, so you can easily review the content.

You can either review the self-check questions at the end of each chapter, or you can jump to them directly by clicking the following chapter titles.