Programming Fundamentals

This chapter introduces the essential components of the Visual Basic language. After creating the interface for your application using forms and controls, you will need to write the code that defines the application's behavior. As with any modern programming language, Visual Basic supports a number of common programming constructs and language elements.

Visual Basic is an object-based programming language. The mere mention of objects may cause undue anxiety in many programmers. Don't worry: whether you realize it or not, you've been dealing with objects most of your life. Once you understand a few basic concepts, objects actually help to make programming easier than ever before.

If you've programmed in other languages, much of the material covered in this chapter will seem familiar. While most of the constructs are similar to other languages, the event-driven nature of Visual Basic introduces some subtle differences. Try and approach this material with an open mind; once you understand the differences you can use them to your advantage.

If you're new to programming, the material in this chapter will serve as an introduction to the basic building blocks for writing code. Once you understand the basics, you will be able to create powerful applications using Visual Basic.

Topics

The Structure of a Visual Basic Application

An introduction to the various components or modules that make up a Visual Basic application.

Before You Start Coding

A brief discussion of some of the considerations in designing an application.

Code Writing Mechanics

An introduction to the features of the Code Editor, plus the rules and regulations for writing code.

Introduction to Variables, Constants and Data Types

An introduction to the basic elements of the Visual Basic language.

Introduction to Procedures

An introduction to Sub and Function procedures.

Introduction to Control Structures

An introduction to decision structures and loops.

Working with Objects

An introduction to using objects in Visual Basic code.

Sample application

Vcr.vbp

Many of the code samples in this chapter are taken from the Vcr.vbp sample application which is listed in the Samples directory.

Send feedback to MSDN. Look here for MSDN Online resources.

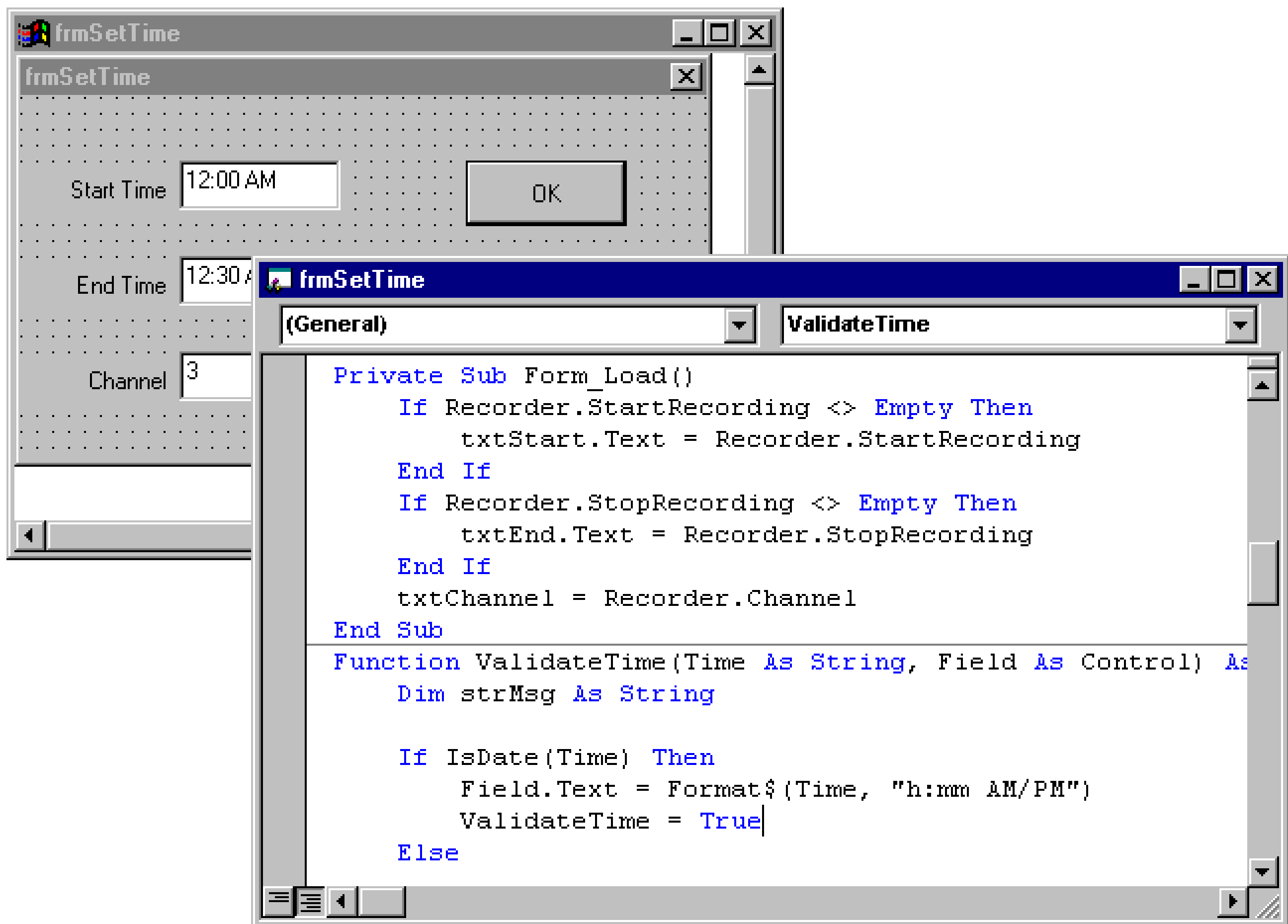
The Structure of a Visual Basic Application

An application is really nothing more than a set of instructions directing the computer to perform a task or tasks. The structure of an application is the way in which the instructions are organized; that is, where the instructions are stored and the order in which instructions are executed.

Simple applications such as the classic "hello world" example have a simple structure; organization isn't very important with a single line of code. As applications become more complex, the need for organization or structure becomes obvious. Imagine the chaos that would result if your application's code was allowed to execute in random order. In addition to controlling the execution of an application, the structure is important to the programmer: how easily can you find specific instructions within your application?

Because a Visual Basic application is based on objects, the structure of its code closely models its physical representation on screen. By definition, objects contain data and code. The form that you see on screen is a representation of the properties that define its appearance and intrinsic behavior. For each form in an application, there is a related *form module* (with file name extension .frm) that contains its code.

Figure 5.1 A form and its related form module



Each form module contains *event procedures* — sections of code where you place the instructions that will execute in response to specific events. Forms can contain controls. For each control on a form, there is a corresponding set of event procedures in the form module. In addition to event procedures, form modules can contain general procedures that are executed in response to a call from any event procedure.

Code that isn't related to a specific form or control can be placed in a different type of module, a *standard module* (.BAS). A procedure that might be used in response to events in several different objects should be placed in a standard module, rather than duplicating the code in the event procedures for each object.

A *class module* (.CLS) is used to create objects that can be called from procedures within your application. Whereas a standard module contains only code, a class module contains both code and data — you can think of it as a control without a physical representation.

While "Managing Projects" describes which components you can add to an application, this chapter explains how to write code in the various components that make up an application. By default, your project contains a single form module. You can add additional form, class, and standard modules, as needed. Class modules are discussed in "Programming with Objects."

- How an Event-Driven Application Works   A discussion of the Event-driven model.

How an Event-Driven Application Works

An event is an action recognized by a form or control. Event-driven applications execute Basic code in response to an event. Each form and control in Visual Basic has a predefined set of events. If one of these events occurs and there is code in the associated event procedure, Visual Basic invokes that code.

Although objects in Visual Basic automatically recognize a predefined set of events, it is up to you to decide if and how they will respond to a particular event. A section of code — an event procedure — corresponds to each event. When you want a control to respond to an event, you write code in the event procedure for that event.

The types of events recognized by an object vary, but many types are common to most controls. For example, most objects recognize a Click event — if a user clicks a form, code in the form's Click event procedure is executed; if a user clicks a command button, code in the button's Click event procedure is executed. The actual code in each case will most likely be quite different.

Here's a typical sequence of events in an event-driven application:

1. The application starts and a form is loaded and displayed.
2. The form (or a control on the form) receives an event. The event might be caused by the user (for example, a keystroke), by the system (for example, a timer event), or indirectly by your code (for example, a Load event when your code loads a form).
3. If there is code in the corresponding event procedure, it executes.
4. The application waits for the next event.

Note Many events occur in conjunction with other events. For example, when the DblClick event occurs, the MouseDown, MouseUp, and Click events also occur.

Send feedback to MSDN. Look here for MSDN Online resources.

Before You Start Coding

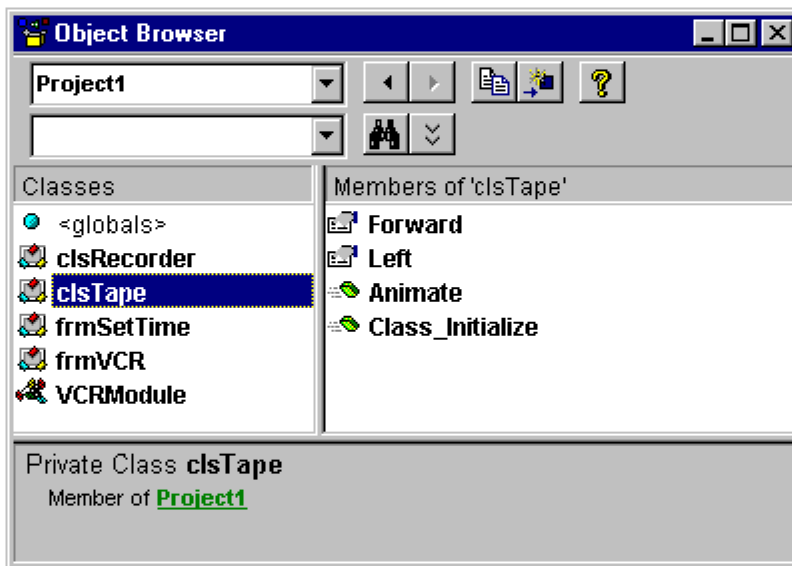
Perhaps the most important (and often overlooked) part of creating an application in Visual Basic is the design phase. While it's obvious that you need to design a user interface for your application, it may not be as obvious that you need to design the structure of the code. The way you structure your application can make a difference in its performance as well as in the maintainability and usability of your code.

The code in a Visual Basic application is organized in a hierarchical fashion. A typical application

consists of one or more modules: a form module for each form in the application, optional standard modules for shared code, and optional class modules. Each module contains one or more procedures that contain the code: event procedures, Sub or Function procedures, and Property procedures.

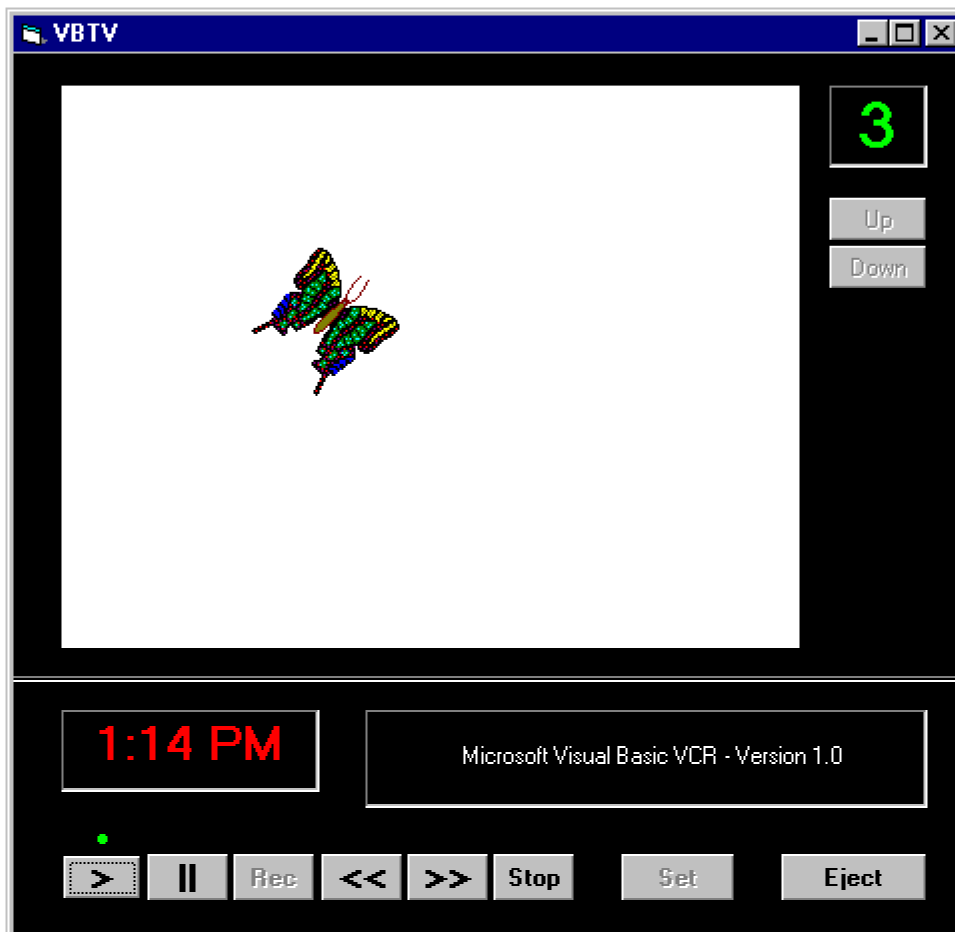
Determining which procedures belong in which module depends somewhat on the type of application that you are creating. Because Visual Basic is based on objects, it helps to think of your application in terms of the objects that it represents. The design of the sample application for this chapter, Vcr.vbp, is based on the objects that comprise a video cassette recorder and a television. The VCR application consists of two form modules, a standard module, and two class modules. You can use the Object Browser to examine the structure of the project (Figure 5.2).

Figure 5.2   The structure of the VCR project is shown in the Object Browser



The main form for the VCR application (frmVCR) is a visual representation of a combination VCR and television screen (Figure 5.3). It is composed of several objects that model those found in the real world version. A group of Command buttons (cmdPlay, cmdRecord, and so on) mimic the buttons used to operate a VCR. The software VCR also contains a clock (lblTime), a channel indicator (lblChannel), function indicators (shpPlay, shpRecord, and so on), and a "picture tube" (picTV). The event procedures for all of these objects are contained in the Vcr.frm form module.

Figure 5.3   The main form for the VCR application



In many cases there are repetitive procedures that are shared by more than one object. For example, when the Play, Rewind, or Record buttons are "pushed," the Pause and Stop buttons need to be enabled. Rather than repeat this code in each button's Click event procedure, it's better to create a shared Sub procedure that can be called by any button. If these procedures need to be modified in the future, all of the modifications can be done in one place. This and other shared procedures are contained in the standard module, Vcr.bas.

Some parts of a VCR aren't visible, such as the tape transport mechanism or the logic for recording a television program. Likewise, some of the functions of the software VCR have no visual representation. These are implemented as two class modules: Recorder.cls and Tape.cls. Code to initiate the "recording" process is contained in the clsRecorder module; code to control the direction and speed of the "tape" is contained in the clsTape module. The classes defined in these modules have no direct references to any of the objects in the forms. Because they are independent code modules, they could easily be reused to build an audio recorder without any modifications.

In addition to designing the structure of your code, it's important to establish naming conventions. By default, Visual Basic names the first form in a project Form1, the second Form2, and so on. If you have several forms in an application, it's a good idea to give them meaningful names to avoid confusion when writing or editing your code. Some suggested naming conventions are presented in "Visual Basic Coding Conventions."

As you learn more about objects and writing code, you can refer to the VCR sample application for examples of various different coding techniques.

Send feedback to MSDN. Look here for MSDN Online resources.

Code Writing Mechanics

Before you begin, it's important to understand the mechanics of writing code in Visual Basic. Like any programming language, Visual Basic has its own rules for organizing, editing, and formatting code.

The following topics introduce code modules and procedures, discuss the basics of using the Code Editor, and cover basic rules for writing code.

- Code Modules   An introduction to the different types of code modules and the procedures that they contain.
- Using the Code Editor   An introduction to working with code using the Code Editor.
- Code Basics   A discussion of the rules of the Visual Basic language.

Send feedback to MSDN. Look here for MSDN Online resources.

Code Modules

Code in Visual Basic is stored in modules. There are three kinds of modules: form, standard, and class.

Simple applications can consist of just a single form, and all of the code in the application resides in that form module. As your applications get larger and more sophisticated, you add additional forms. Eventually you might find that there is common code you want to execute in several forms. You don't want to duplicate the code in both forms, so you create a separate module containing a procedure that implements the common code. This separate module should be a standard module. Over time, you can build up a library of modules containing shared procedures.

Each standard, class, and form module can contain:

- Declarations. You can place constant, type, variable, and dynamic-link library (DLL) procedure declarations at the module level of form, class or standard modules.
- Procedures. A Sub, Function, or Property procedure contains pieces of code that can be executed as a unit. These are discussed in the section "Procedures" later in this chapter.

Form Modules

Form modules (.FRM file name extension) are the foundation of most Visual Basic applications. They can contain procedures that handle events, general procedures, and form-level declarations of variables, constants, types, and external procedures. If you were to look at a form module in a text editor, you would also see descriptions of the form and its controls, including their property settings. The code that you write in a form module is specific to the particular application to which the form belongs; it might also reference other forms or objects within that application.

Standard Modules

Standard modules (.BAS file name extension) are containers for procedures and declarations commonly accessed by other modules within the application. They can contain global (available to the whole application) or module-level declarations of variables, constants, types, external procedures, and global procedures. The code that you write in a standard module isn't necessarily tied to a particular application; if you're careful not to reference forms or controls by name, a standard module can be reused in many different applications.

Class Modules

Class modules (.CLS file name extension) are the foundation of object-oriented programming in Visual Basic. You can write code in class modules to create new objects. These new objects can include your own customized properties and methods. Actually, forms are just class modules that can have controls placed on them and can display form windows.

For More Information For information about writing code in class modules, see "Programming with Objects."

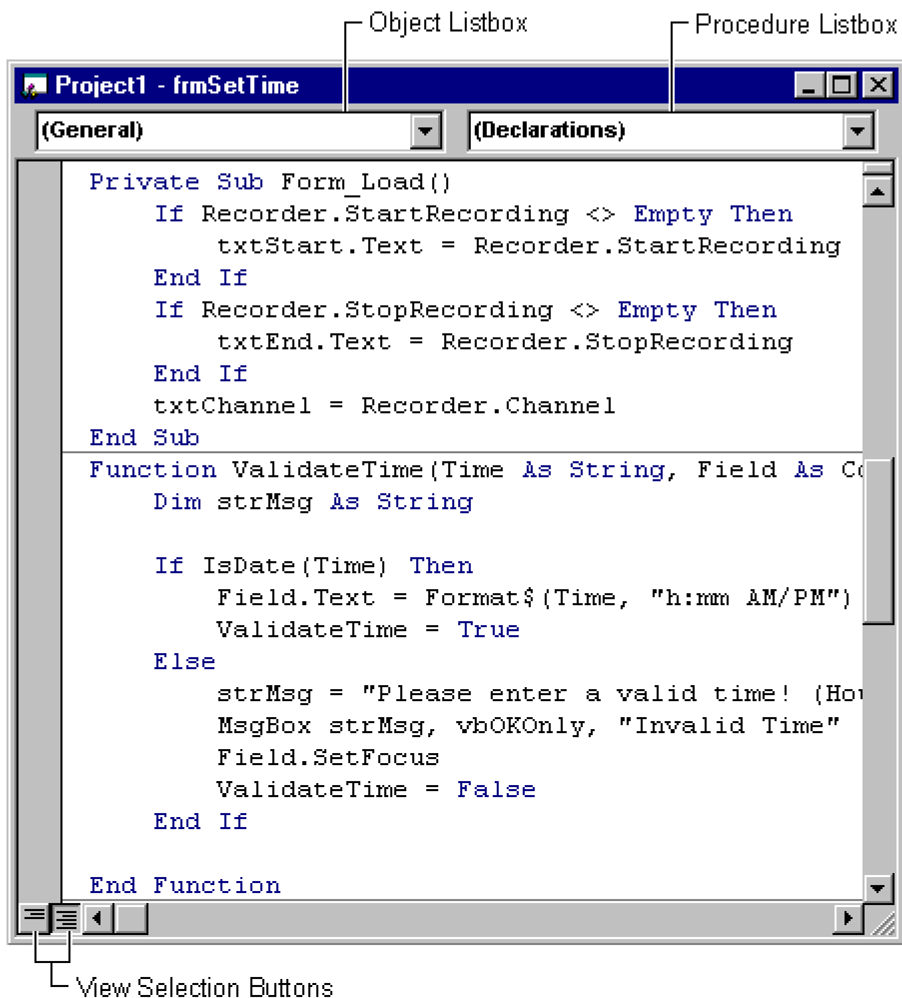
> **Note** The Professional and Enterprise editions of Visual Basic also include ActiveX Documents, ActiveX Designers, and User Controls. These introduce new types of modules with different file name extensions. From the standpoint of writing code, these modules should be considered the same as form modules.

Send feedback to MSDN. Look here for MSDN Online resources.

Using the Code Editor

The Visual Basic Code Editor is a window where you write most of your code. It is like a highly specialized word processor with a number of features that make writing Visual Basic code a lot easier. The Code Editor window is shown in Figure 5.4.

Figure 5.4 The Code Editor window



Because you work with Visual Basic code in modules, a separate Code Editor window is opened for each module you select from the Project Explorer. Code within each module is subdivided into separate sections for each object contained in the module. Switching between sections is accomplished using the Object Listbox. In a form module, the list includes a general section, a section for the form itself, and a section for each control contained on the form. For a class module, the list includes a general section and a class section; for a standard module only a general section is shown.

Each section of code can contain several different procedures, accessed using the Procedure Listbox. The procedure list for a form module contains a separate section for each event procedure for the form or control. For example, the procedure list for a Label control includes sections for the Change, Click, and DblClick events, among others. Class modules list only the event procedures for the class itself — Initialize and Terminate. Standard modules don't list any event procedures, because a standard module doesn't support events.

The procedure list for a general section of a module contains a single selection — the Declarations section, where you place module-level variable, constant, and DLL declarations. As you add Sub or Function procedures to a module, those procedures are added in the Procedure Listbox below the Declarations section.

Two different views of your code are available in the Code Editor window. You can choose to view a single procedure at a time, or to view all of the procedures in the module with each procedure separated from the next by a line (as shown in Figure 5.4). To switch between the

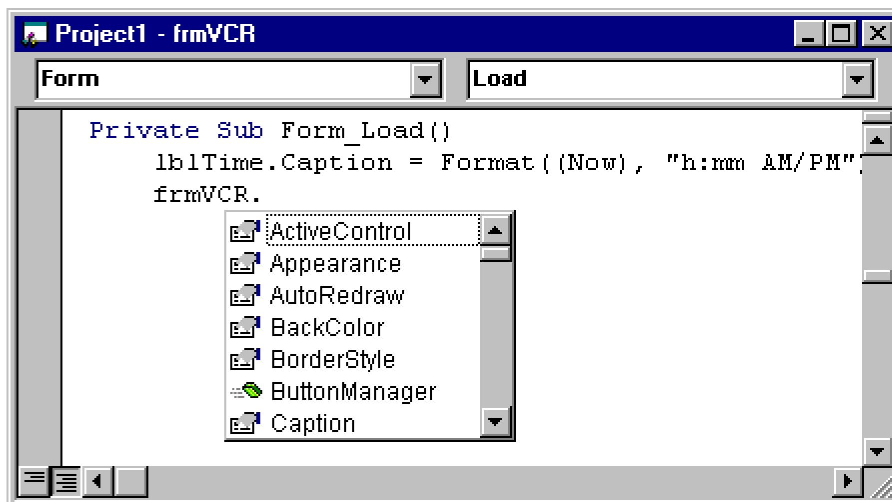
two views, use the View Selection buttons in the lower left-hand corner of the editor window.

Automatic Code Completion

Visual Basic makes writing code much easier with features that can automatically fill in statements, properties, and arguments for you. As you enter code, the editor displays lists of appropriate choices, statement or function prototypes, or values. Options for enabling or disabling these and other code settings are available on the Editor tab of the Options dialog, accessed through the Options command on the Tools menu.

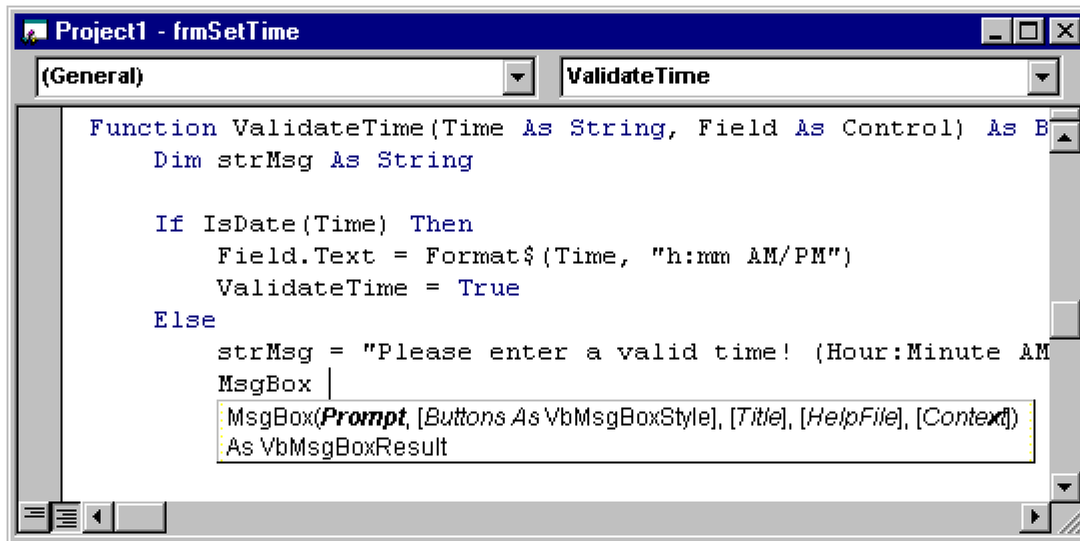
When you enter the name of a control in your code, the Auto List Members feature presents a drop-down list of properties available for that control (Figure 5.5). Type in the first few letters of the property name and the name will be selected from the list; the TAB key will complete the typing for you. This option is also helpful when you aren't sure which properties are available for a given control. Even if you choose to disable the Auto List Members feature, you can still access it with the CTRL+J key combination.

Figure 5.5 The Auto List Members feature



The Auto Quick Info feature displays the syntax for statements and functions (Figure 5.6). When you enter the name of a valid Visual Basic statement or function the syntax is shown immediately below the current line, with the first argument in bold. After you enter the first argument value, the second argument appears in bold. Auto Quick Info can also be accessed with the CTRL+I key combination.

Figure 5.6 Auto Quick Info



Bookmarks

Bookmarks can be used to mark lines of code in the Code Editor so that you can easily return to them later. Commands to toggle bookmarks on or off as well as to navigate existing bookmarks are available from the Edit, Bookmarks menu item, or from the Edit toolbar

For More Information For more information on key combinations to access these and other functions in the Code Editor window, see "Code Window Keyboard Shortcuts."

Send feedback to MSDN. Look here for MSDN Online resources.

Code Basics

This section presents information on code writing mechanics, including breaking and combining lines of code, adding comments to your code, using numbers in code, and following naming conventions in Visual Basic.

Breaking a Single Statement Into Multiple Lines

You can break a long statement into multiple lines in the Code window using the *line-continuation character* (a space followed by an underscore). Using this character can make your code easier to read, both online and when printed. The following code is broken into three lines with line-continuation characters (_):

```
Data1.RecordSource = _
"SELECT * FROM Titles, Publishers" _
& "WHERE Publishers.PubId = Titles.PubID" _
& "AND Publishers.State = 'CA'"
```

You can't follow a line-continuation character with a comment on the same line. There are also

some limitations as to where the line-continuation character can be used.

Combining Statements on One Line

There is usually one Visual Basic statement to a line, and there is no statement terminator. However, you can place two or more statements on a line if you use a colon (:) to separate them:

```
Text1.Text = "Hello" : Red = 255 : Text1.BackColor = _
Red
```

In order to make your code more readable, however, it's better to place each statement on a separate line.

For More Information For more information, see "Visual Basic Specifications, Limitations, and File Formats."

Adding Comments to Your Code

As you read through the examples in this guide, you'll often come across the comment symbol ('). This symbol tells Visual Basic to ignore the words that follow it. Such words are remarks placed in the code for the benefit of the developer, and other programmers who might examine the code later. For example:

```
' This is a comment beginning at the left edge of the
' screen.
Text1.Text = "Hi!"      ' Place friendly greeting in text
                     ' box.
```

Comments can follow a statement on the same line or can occupy an entire line. Both are illustrated in the preceding code. Remember that comments can't follow a line-continuation character on the same line.

> **Note** You can add or remove comment symbols for a block of code by selecting two or more lines of code and choosing the Comment Block or Uncomment Block buttons on the Edit toolbar.

Understanding Numbering Systems

Most numbers in this documentation are decimal (base 10). But occasionally it's convenient to use hexadecimal numbers (base 16) or octal numbers (base 8). Visual Basic represents numbers in hexadecimal with the prefix &H and in octal with &O. The following table shows the same numbers in decimal, octal, and hexadecimal.

Decimal	Octal	Hexadecimal
9	&O11	&H9
15	&O17	&HF
16	&O20	&H10

20	&O24	&H14
255	&O377	&HFF

You generally don't have to learn the hexadecimal or octal number system yourself because the computer can work with numbers entered in any system. However, some number systems lend themselves to certain tasks, such as using hexadecimals to set the screen and control colors.

Naming Conventions in Visual Basic

While you are writing Visual Basic code, you declare and name many elements (Sub and Function procedures, variables, constants, and so on). The names of the procedures, variables, and constants that you declare in your Visual Basic code must follow these guidelines:

- They must begin with a letter.
- They can't contain embedded periods or type-declaration characters (special characters that specify a data type.
- They can be no longer than 255 characters. The names of controls, forms, classes, and modules must not exceed 40 characters.
- They can't be the same as restricted keywords.

A *restricted keyword* is a word that Visual Basic uses as part of its language. This includes predefined statements (such as If and Loop), functions (such as Len and Abs), and operators (such as Or and Mod).

For More Information For a complete list of keywords, see the *Language Reference*.

Your forms and controls can have the same name as a restricted keyword. For example, you can have a control named Loop. In your code you cannot refer to that control in the usual way, however, because Visual Basic assumes you mean the Loop keyword. For example, this code causes an error:

```
Loop.Visible = True            ' Causes an error.
```

To refer to a form or control that has the same name as a restricted keyword, you must either qualify it or surround it with square brackets: [ ]. For example, this code does not cause an error:

```
MyForm.Loop.Visible = True    ' Qualified with the form
                             ' name.
[Loop].Visible = True          ' Square brackets also
                             ' work.
```

You can use square brackets in this way when referring to forms and controls, but not when declaring a variable or defining a procedure with the same name as a restricted keyword. Square brackets can also be used to force Visual Basic to accept names provided by other type libraries that conflict with restricted keywords.

> **Note** Because typing square brackets can get tedious, you might want to refrain from using restricted keywords as the name of forms and controls. However, you can use this

technique if a future version of Visual Basic defines a new keyword that conflicts with an existing form or control name when you update your code to work with the new version.

Send feedback to MSDN.   Look here for MSDN Online resources.

Introduction to Variables, Constants and Data Types

You often need to store values temporarily when performing calculations with Visual Basic. For example, you might want to calculate several values, compare them, and perform different operations on them, depending on the result of the comparison. You need to retain the values if you want to compare them, but you don't need to store them in a property.

Visual Basic, like most programming languages, uses *variables* for storing values. Variables have a name (the word you use to refer to the value the variable contains) and a *data type* (which determines the kind of data the variable can store). *Arrays* can be used to store indexed collections of related variables.

Constants also store values, but as the name implies, those values remain constant throughout the execution of an application. Using constants can make your code more readable by providing meaningful names instead of numbers. There are a number of built-in constants in Visual Basic, but you can also create your own.

Data types control the internal storage of data in Visual Basic. By default, Visual Basic uses the Variant data type. There are a number of other available data types that allow you to optimize your code for speed and size when you don't need the flexibility that Variant provides.

For more detailed information, see:

- Variables   An introduction to variables: what they are and how to use them.
- Understanding the Scope of Variables   A discussion of scope as it applies to variables.
- Advanced Variable Topics   Detailed information about variables.
- Static Variables   An introduction to using static variables to preserve values.
- Constants   An introduction to using constants to represent values.
- Data Types   A discussion of the data types available in Visual Basic.
- Advanced Variant Topics   Detailed information about the Variant data type.
- Arrays   An introduction to the use of arrays for groups of values.
- Dynamic Arrays   A discussion of using dynamic arrays to work with groups of values.

Send feedback to MSDN.   Look here for MSDN Online resources.

Variables

In Visual Basic, you use variables to temporarily store values during the execution of an application. Variables have a name (the word you use to refer to the value the variable contains) and a data type (which determines the kind of data the variable can store).

You can think of a variable as a placeholder in memory for an unknown value. For example, imagine you are creating a program for a fruit stand to track the sales of apples. You don't know the price of an apple or the quantity sold until the sale actually occurs. You can use two variables to hold the unknown values — let's name them ApplePrice and ApplesSold. Each time the program is run, the user supplies the values for the two variables. To calculate the total sales and display it in a Textbox named txtSales, your code would look like this:

```
txtSales.txt = ApplePrice * ApplesSold
```

The expression returns a different total each time, depending on what values the user provides. The variables allow you to make a calculation without having to know in advance what the actual inputs are.

In this example, the data type of ApplePrice is Currency; the data type of ApplesSold is an integer. Variables can represent many other values as well: text values, dates, various numeric types, even objects.

Storing and Retrieving Data in Variables

You use assignment statements to perform calculations and assign the result to a variable:

```
ApplesSold = 10      ' The value 10 is passed to the
                  ' variable.
ApplesSold = ApplesSold + 1      ' The variable is
                              ' incremented.
```

Note that the equal sign in this example is an assignment operator, not an equality operator; the value (10) is being assigned to the variable (ApplesSold).

Declaring Variables

To declare a variable is to tell the program about it in advance. You declare a variable with the Dim statement, supplying a name for the variable:

Dim *variablename* [**As** *type*]

Variables declared with the Dim statement within a procedure exist only as long as the procedure is executing. When the procedure finishes, the value of the variable disappears. In addition, the value of a variable in a procedure is *local* to that procedure — that is, you can't access a variable in one procedure from another procedure. These characteristics allow you to use the same variable names in different procedures without worrying about conflicts or accidental changes.

A variable name:

- Must begin with a letter.
- Can't contain an embedded period or embedded type-declaration character.
- Must not exceed 255 characters.
- Must be unique within the same *scope,* which is the range from which the variable can be referenced — a procedure, a form, and so on.

The optional As *type* clause in the Dim statement allows you to define the data type or object type of the variable you are declaring. Data types define the type of information the variable stores. Some examples of data types include String, Integer, and Currency. Variables can also contain objects from Visual Basic or other applications. Examples of Visual Basic object types, or classes, include Object, Form1, and TextBox.

For More Information For more information on objects, see "Programming with Objects" and "Programming with Components." Data types are discussed in detail in the section, "Data Types," later in this chapter.

There are other ways to declare variables:

- Declaring a variable in the Declarations section of a form, standard, or class module, rather than within a procedure, makes the variable available to all the procedures in the module.
- Declaring a variable using the Public keyword makes it available throughout your application.
- Declaring a local variable using the Static keyword preserves its value even when a procedure ends.

Implicit Declaration

You don't have to declare a variable before using it. For example, you could write a function where you don't need to declare `TempVal` before using it:

```
Function SafeSqr(num)
   TempVal = Abs(num)
   SafeSqr = Sqr(TempVal)
End Function
```

Visual Basic automatically creates a variable with that name, which you can use as if you had explicitly declared it. While this is convenient, it can lead to subtle errors in your code if you misspell a variable name. For example, suppose that this was the function you wrote:

```
Function SafeSqr(num)
   TempVal = Abs(num)
   SafeSqr = Sqr(TemVal)
End Function
```

At first glance, this looks the same. But because the `TempVal` variable was misspelled on the next-to-last line, this function will always return zero. When Visual Basic encounters a new name, it can't determine whether you actually meant to implicitly declare a new variable or you just misspelled an existing variable name, so it creates a new variable with that name.

Explicit Declaration

To avoid the problem of misnaming variables, you can stipulate that Visual Basic always warn you whenever it encounters a name not declared explicitly as a variable.

To explicitly declare variables

- Place this statement in the Declarations section of a class, form, or standard module:

  ```
  Option Explicit
  ```

  –or–

  From the **Tools** menu, choose **Options**, click the **Editor** tab and check the **Require Variable Declaration** option. This automatically inserts the Option Explicit statement in any new modules, but not in modules already created; therefore, you must manually add Option Explicit to any existing modules within a project.

Had this statement been in effect for the form or standard module containing the SafeSqr function, Visual Basic would have recognized `TempVal` and `TemVal` as undeclared variables and generated errors for both of them. You could then explicitly declare `TempVal`:

```
Function SafeSqr(num)
   Dim TempVal
   TempVal = Abs(num)
   SafeSqr = Sqr(TemVal)
End Function
```

Now you'd understand the problem immediately because Visual Basic would display an error message for the incorrectly spelled `TemVal`. Because the Option Explicit statement helps you catch these kinds of errors, it's a good idea to use it with all your code.

> **Note** The Option Explicit statement operates on a per-module basis; it must be placed in the Declarations section of every form, standard, and class module for which you want Visual Basic to enforce explicit variable declarations. If you select Require Variable Declaration, Visual Basic inserts Option Explicit in all subsequent form, standard, and class modules, but does not add it to existing code. You must manually add Option Explicit to any existing modules within a project.

Send feedback to MSDN. *Look here for MSDN Online resources.*

Understanding the Scope of Variables

The scope of a variable defines which parts of your code are aware of its existence. When you declare a variable within a procedure, only code within that procedure can access or change the value of that variable; it has a scope that is local to that procedure. Sometimes, however, you need to use a variable with a broader scope, such as one whose value is available to all the

procedures within the same module, or even to all the procedures in your entire application. Visual Basic allows you to specify the scope of a variable when you declare it.

Scoping Variables

Depending on how it is declared, a variable is scoped as either a procedure-level (local) or module-level variable.

Scope	Private	Public
Procedure-level	Variables are private to the procedure in which they appear.	Not applicable. You cannot declare public variables within a procedure.
Module-level	Variables are private to the module in which they appear.	Variables are available to all modules.

Variables Used Within a Procedure

Procedure-level variables are recognized only in the procedure in which they're declared. These are also known as local variables. You declare them with the Dim or Static keywords. For example:

```
Dim intTemp As Integer
```

–or–

```
Static intPermanent As Integer
```

Values in local variables declared with Static exist the entire time your application is running while variables declared with Dim exist only as long as the procedure is executing.

Local variables are a good choice for any kind of temporary calculation. For example, you can create a dozen different procedures containing a variable called `intTemp`. As long as each `intTemp` is declared as a local variable, each procedure recognizes only its own version of `intTemp`. Any one procedure can alter the value in its local `intTemp` without affecting `intTemp` variables in other procedures.

Variables Used Within a Module

By default, a module-level variable is available to all the procedures in that module, but not to code in other modules. You create module-level variables by declaring them with the Private keyword in the Declarations section at the top of the module. For example:

```
Private intTemp As Integer
```

At the module level, there is no difference between Private and Dim, but Private is preferred because it readily contrasts with Public and makes your code easier to understand.

Variables Used by All Modules

To make a module-level variable available to other modules, use the Public keyword to declare the variable. The values in public variables are available to all procedures in your application. Like all module-level variables, public variables are declared in the Declarations section at the top of the module. For example:

```
Public intTemp As Integer
```

> **Note** You can't declare public variables within a procedure, only within the Declarations section of a module.

For More Information For additional information about variables, see "Advanced Variable Topics."

Send feedback to MSDN. Look here for MSDN Online resources.

Advanced Variable Topics

Using Multiple Variables with the Same Name

If public variables in different modules share the same name, it's possible to differentiate between them in code by referring to both the module and variable names. For example, if there is a public Integer variable `intX` declared in both Form1 and in Module1, you can refer to them as `Module1.intX` and `Form1.intX` to get the correct values.

To see how this works, insert two standard modules in a new project and draw three command buttons on a form.

One variable, `intX`, is declared in the first standard module, Module1. The Test procedure sets its value:

```
Public intX As Integer      ' Declare Module1's intX.
Sub Test()
   ' Set the value for the intX variable in Module1.
   intX = 1
End Sub
```

The second variable, which has the same name, `intX`, is declared in the second standard module, Module2. Again, a procedure named Test sets its value:

```
Public intX As Integer      ' Declare Module2's intX.
Sub Test()
   ' Set the value for the intX variable in Module2.
   intX = 2
End Sub
```

The third `intX` variable is declared in the form module. And again, a procedure named Test sets its value.

```
Public intX As Integer    ' Declare the form's intX
                        ' variable.
Sub Test()
   ' Set the value for the intX variable in the form.
   intX = 3
End Sub
```

Each of the three command buttons' Click event procedures calls the appropriate Test procedure and uses MsgBox to display the values of the three variables.

```
Private Sub Command1_Click()
   Module1.Test               ' Calls Test in Module1.
   MsgBox Module1.intX         ' Displays Module1's intX.
End Sub

Private Sub Command2_Click()
   Module2.Test               ' Calls Test in Module2.
   MsgBox Module2.intX         ' Displays Module2's intX.
End Sub

Private Sub Command3_Click()
   Test                     ' Calls Test in Form1.
   MsgBox intX               ' Displays Form1's intX.
End Sub
```

Run the application and click each of the three command buttons. You'll see the separate references to the three public variables. Notice in the third command button's Click event procedure, you don't need to specify `Form1.Test` when calling the form's Test procedure, or `Form1.intX` when calling the value of the form's Integer variable. If there are multiple procedures and variables with the same name, Visual Basic takes the value of the more local variable, which in this case, is the Form1 variable.

Public vs. Local Variables

You can also have a variable with the same name at a different scope. For example, you could have a public variable named `Temp` and then, within a procedure, declare a local variable named `Temp`. References to the name `Temp` within the procedure would access the local variable; references to `Temp` outside the procedure would access the public variable. The module-level variable can be accessed from within the procedure by qualifying the variable with the module name.

```
Public Temp As Integer
Sub Test()
   Dim Temp As Integer
   Temp = 2              ' Temp has a value of 2.
   MsgBox Form1.Temp      ' Form1.Temp has a value of 1.
End Sub

Private Sub Form_Load()
   Temp = 1              ' Set Form1.Temp to 1.
End Sub
Private Sub Command1_Click()
   Test
End Sub
```

In general, when variables have the same name but different scope, the more local variable always *shadows* (that is, it is accessed in preference to) less local variables. So if you also had a procedure-level variable named `Temp`, it would shadow the public variable `Temp` within that

module.

Shadowing Form Properties and Controls

Due to the effect of shadowing, form properties, controls, constants, and procedures are treated as module-level variables in the form module. It is not legal to have a form property or control with the same name as a module-level variable, constant, user-defined type, or procedure because both are in the same scope.

Within the form module, local variables with the same names as controls on the form shadow the controls. You must qualify the control with a reference to the form or the Me keyword to set or get its value or any of its properties. For example:

```
Private Sub Form_Click ()
Dim Text1, BackColor
' Assume there is also a control on the form called
' Text1.
   Text1 = "Variable"      ' Variable shadows control.
   Me.Text1 = "Control"    ' Must qualify with Me to get
                        ' control.
   Text1.Top = 0          ' This causes an error!
   Me.Text1.Top = 0       ' Must qualify with Me to get
                        ' control.
   BackColor = 0          ' Variable shadows property.
   Me.BackColor = 0       ' Must qualify with Me to get
                        ' form property.
End Sub
```

Using Variables and Procedures with the Same Name

The names of your private module-level and public module-level variables can also conflict with the names of your procedures. A variable in the module cannot have the same name as any procedures or types defined in the module. It can, however, have the same name as public procedures, types, or variables defined in other modules. In this case, when the variable is accessed from another module, it must be qualified with the module name.

While the shadowing rules described above are not complex, shadowing can be confusing and lead to subtle bugs in your code; it is good programming practice to keep the names of your variables distinct from each other. In form modules, try to use variables names that are different from names of controls on those forms.

Send feedback to MSDN.   Look here for MSDN Online resources.

Static Variables

In addition to scope, variables have a *lifetime*, the period of time during which they retain their value. The values in module-level and public variables are preserved for the lifetime of your application. However, local variables declared with Dim exist only while the procedure in which they are declared is executing. Usually, when a procedure is finished executing, the values of

its local variables are not preserved and the memory used by the local variables is reclaimed. The next time the procedure is executed, all its local variables are reinitialized.

However, you can preserve the value of a local variable by making the variable *static*. Use the Static keyword to declare one or more variables inside a procedure, exactly as you would with the Dim statement:

```
Static Depth
```

For example, the following function calculates a running total by adding a new value to the total of previous values stored in the static variable `Accumulate`:

```
Function RunningTotal(num)
   Static ApplesSold
   ApplesSold = ApplesSold + num
   RunningTotal = ApplesSold
End Function
```

If `ApplesSold` was declared with Dim instead of Static, the previous accumulated values would not be preserved across calls to the function, and the function would simply return the same value with which it was called.

You can produce the same result by declaring `ApplesSold` in the Declarations section of the module, making it a module-level variable. Once you change the scope of a variable this way, however, the procedure no longer has exclusive access to it. Because other procedures can access and change the value of the variable, the running totals might be unreliable and the code would be more difficult to maintain.

Declaring All Local Variables as Static

To make all local variables in a procedure static, place the Static keyword at the beginning of a procedure heading. For example:

```
Static Function RunningTotal(num)
```

This makes all the local variables in the procedure static regardless of whether they are declared with Static, Dim, Private, or declared implicitly. You can place Static in front of any Sub or Function procedure heading, including event procedures and those declared as Private.

Send feedback to MSDN.   Look here for MSDN Online resources.

Constants

Often you'll find that your code contains constant values that reappear over and over. Or you may find that the code depends on certain numbers that are difficult to remember — numbers that, in and of themselves, have no obvious meaning.

In these cases, you can greatly improve the readability of your code — and make it easier to

maintain — by using constants. A *constant* is a meaningful name that takes the place of a number or string that does not change. Although a constant somewhat resembles a variable, you can't modify a constant or assign a new value to it as you can to a variable. There are two sources for constants:

- *Intrinsic* or *system-defined* constants are provided by applications and controls. Visual Basic constants are listed in the Visual Basic (VB) and Visual Basic for applications (VBA) object libraries in the Object Browser. Other applications that provide object libraries, such as Microsoft Excel and Microsoft Project, also provide a list of constants you can use with their objects, methods, and properties. Constants are also defined in the object library for each ActiveX control. For details on using the Object Browser, see "Programming with Objects."
- *Symbolic* or *user-defined* constants are declared using the Const statement. User-defined constants are described in the next section, "Creating Your Own Constants."

In Visual Basic, constant names are in a mixed-case format, with a prefix indicating the object library that defines the constant. Constants from the Visual Basic and Visual Basic for applications object libraries are prefaced with "vb" — for instance, vbTileHorizontal.

The prefixes are intended to prevent accidental collisions in cases where constants have identical names and represent different values. Even with prefixes, it's still possible that two object libraries may contain identical constants representing different values. Which constant is referenced in this case depends on which object library has the higher priority. For information on changing the priority of object libraries, see the "References Dialog Box."

To be absolutely sure you avoid constant name collisions, you can qualify references to constants with the following syntax:

[*libname*.][*modulename*.]*constname*

Libname is usually the class name of the control or library. *Modulename* is the name of the module that defines the constant. *Constname* is the name of the constant. Each of these elements is defined in the object library, and can be viewed in the Object Browser.

Creating Your Own Constants

The syntax for declaring a constant is:

[**Public|Private**] **Const** *constantname*[**As** *type*] **=** *expression*

The argument *constantname* is a valid symbolic name (the rules are the same as those for creating variable names), and *expression* is composed of numeric or string constants and operators; however, you can't use function calls in *expression*.

A Const statement can represent a mathematical or date/time quantity:

```
Const conPi = 3.14159265358979
Public Const conMaxPlanets As Integer = 9
Const conReleaseDate = #1/1/95#
```

The Const statement can also be used to define string constants:

```
Public Const conVersion = "07.10.A"
Const conCodeName = "Enigma"
```

You can place more than one constant declaration on a single line if you separate them with commas:

```
Public Const conPi = 3.14, conMaxPlanets = 9, _
conWorldPop = 6E+09
```

The expression on the right side of the equal sign (=) is often a number or literal string, but it can also be an expression that results in a number or string (although that expression can't contain calls to functions). You can even define constants in terms of previously defined constants:

```
Const conPi2 = conPi * 2
```

Once you define constants, you can place them in your code to make it more readable. For example:

```
Static SolarSystem(1 To conMaxPlanets)
If numPeople > conWorldPop Then Exit Sub
```

Scoping User-Defined Constants

A Const statement has scope like a variable declaration, and the same rules apply:

- To create a constant that exists only within a procedure, declare it within that procedure.
- To create a constant available to all procedures within a module, but not to any code outside that module, declare it in the Declarations section of the module.
- To create a constant available throughout the application, declare the constant in the Declarations section of a standard module, and place the Public keyword before Const. Public constants cannot be declared in a form or class module.

For More Information For more information regarding scope, see "Understanding the Scope of Variables" earlier in this chapter.

Avoiding Circular References

Because constants can be defined in terms of other constants, you must be careful not to set up a *cycle*, or circular reference between two or more constants. A cycle occurs when you have two or more public constants, each of which is defined in terms of the other.

For example:

```
' In Module 1:
Public Const conA = conB * 2   ' Available throughout
                              ' application.
' In Module 2:
Public Const conB = conA / 2   ' Available throughout
                              ' application.
```

If a cycle occurs, Visual Basic generates an error when you attempt to run your application. You

cannot run your code until you resolve the circular reference. To avoid creating a cycle, restrict all your public constants to a single module or, at most, a small number of modules.

Send feedback to MSDN.   Look here for MSDN Online resources.

Data Types

Variables are placeholders used to store values; they have names and data types. The data type of a variable determines how the bits representing those values are stored in the computer's memory. When you declare a variable, you can also supply a data type for it. All variables have a data type that determines what kind of data they can store.

By default, if you don't supply a data type, the variable is given the Variant data type. The Variant data type is like a chameleon — it can represent many different data types in different situations. You don't have to convert between these types of data when assigning them to a Variant variable: Visual Basic automatically performs any necessary conversion.

If you know that a variable will always store data of a particular type, however, Visual Basic can handle that data more efficiently if you declare a variable of that type. For example, a variable to store a person's name is best represented as a string data type, because a name is always composed of characters.

Data types apply to other things besides variables. When you assign a value to a property, that value has a data type; arguments to functions also have data types. In fact, just about anything in Visual Basic that involves data also involves data types.

You can also declare arrays of any of the fundamental types.

For More Information   For more information, see the section, "Arrays," later in this chapter. Selecting data types to improve your application's performance is discussed in "Designing for Performance and Compatibility."

Declaring Variables with Data Types

Before using a non-Variant variable, you must use the Private, Public, Dim or Static statement to declare it As *type*. For example, the following statements declare an Integer, Double, String, and Currency type, respectively:

```
Private I As Integer
Dim Amt As Double
Static YourName As String
Public BillsPaid As Currency
```

A Declaration statement can combine multiple declarations, as in these statements:

```
Private I As Integer, Amt As Double
Private YourName As String, BillsPaid As Currency
Private Test, Amount, J As Integer
```

> **Note** If you do not supply a data type, the variable is given the default type. In the preceding example, the variables `Test` and `Amount` are of the Variant data type. This may surprise you if your experience with other programming languages leads you to expect all variables in the same declaration statement to have the same specified type (in this case, Integer).

Numeric Data Types

Visual Basic supplies several numeric data types — Integer, Long (long integer), Single (single-precision floating point), Double (double-precision floating point), and Currency. Using a numeric data type generally uses less storage space than a variant.

If you know that a variable will always store whole numbers (such as 12) rather than numbers with a fractional amount (such as 3.57), declare it as an Integer or Long type. Operations are faster with integers, and these types consume less memory than other data types. They are especially useful as the counter variables in For...Next loops.

For More Information To read more about control structures, see "Introduction to Control Structures" later in this chapter.

If the variable contains a fraction, declare it as a Single, Double, or Currency variable. The Currency data type supports up to four digits to the right of the decimal separator and fifteen digits to the left; it is an accurate fixed-point data type suitable for monetary calculations. Floating-point (Single and Double) numbers have much larger ranges than Currency, but can be subject to small rounding errors.

> **Note** Floating-point values can be expressed as *mmm***E***eee* or *mmm***D***eee*, in which *mmm* is the mantissa and *eee* is the exponent (a power of 10). The highest positive value of a Single data type is 3.402823E+38, or 3.4 times 10 to the 38th power; the highest positive value of a Double data type is 1.79769313486232D+308, or about 1.8 times 10 to the 308th power. Using **D** to separate the mantissa and exponent in a numeric literal causes the value to be treated as a Double data type. Likewise, using **E** in the same fashion treats the value as a Single data type.

The Byte Data Type

If the variable contains binary data, declare it as an array of the Byte data type. (Arrays are discussed in "Arrays" later in this chapter). Using Byte variables to store binary data preserves it during format conversions. When String variables are converted between ANSI and Unicode formats, any binary data in the variable is corrupted. Visual Basic may automatically convert between ANSI and Unicode when:

- Reading from files
- Writing to files
- Calling DLLs
- Calling methods and properties on objects

All operators that work on integers work with the Byte data type except unary minus. Since Byte is an unsigned type with the range 0-255, it cannot represent a negative number. So for unary minus, Visual Basic coerces the Byte to a signed integer first.

All numeric variables can be assigned to each other and to variables of the Variant type. Visual Basic rounds off rather than truncates the fractional part of a floating-point number before assigning it to an integer.

For More Information For details on Unicode and ANSI conversions, see "International Issues."

The String Data Type

If you have a variable that will always contain a string and never a numeric value, you can declare it to be of type String:

```
Private S As String
```

You can then assign strings to this variable and manipulate it using string functions:

```
S = "Database"
S = Left(S, 4)
```

By default, a string variable or argument is a *variable-length string*; the string grows or shrinks as you assign new data to it. You can also declare strings that have a fixed length. You specify a *fixed-length string* with this syntax:

String * *size*

For example, to declare a string that is always 50 characters long, use code like this:

```
Dim EmpName As String * 50
```

If you assign a string of fewer than 50 characters, `EmpName` is padded with enough trailing spaces to total 50 characters. If you assign a string that is too long for the fixed-length string, Visual Basic simply truncates the characters.

Because fixed-length strings are padded with trailing spaces, you may find the Trim and RTrim functions, which remove the spaces, useful when working with them.

Fixed-length strings in standard modules can be declared as Public or Private. In forms and class modules, fixed-length strings must be declared Private.

For More Information See "Ltrim, RTrim Function and Trim Functions" in the *Language Reference*.

Exchanging Strings and Numbers

You can assign a string to a numeric variable if the string represents a numeric value. It's also possible to assign a numeric value to a string variable. For example, place a command button, text box, and list box on a form. Enter the following code in the command button's Click event. Run the application, and click the command button.

```
Private Sub Command1_Click()
   Dim intX As Integer
   Dim strY As String
   strY = "100.23"
```

```
   intX = strY      ' Passes the string to a numeric
                  ' variable.
   List1.AddItem Cos(strY)   ' Adds cosine of number in
                        ' the string to the listbox.
   strY = Cos(strY)         ' Passes cosine to the
                           ' string variable.
   Text1.Text = strY      ' String variable prints in
                        ' the text box.
End Sub
```

Visual Basic will automatically coerce the variables to the appropriate data type. You should use caution when exchanging strings and numbers; passing a non-numeric value in the string will cause a run-time error to occur.

The Boolean Data Type

If you have a variable that will contain simple true/false, yes/no, or on/off information, you can declare it to be of type Boolean. The default value of Boolean is False. In the following example, `blnRunning` is a Boolean variable which stores a simple yes/no setting.

```
Dim blnRunning As Boolean
   ' Check to see if the tape is running.
   If Recorder.Direction = 1 Then
   blnRunning = True
End if
```

The Date Data Type

Date and time values can be contained both in the specific Date data type and in Variant variables. The same general characteristics apply to dates in both types.

For More Information See the section, "Date/Time Values Stored in Variants," in "Advanced Variant Topics."

When other numeric data types are converted to Date, values to the left of the decimal represent date information, while values to the right of the decimal represent time. Midnight is 0, and midday is 0.5. Negative whole numbers represent dates before December 30, 1899.

The Object Data Type

Object variables are stored as 32-bit (4-byte) addresses that refer to objects within an application or within some other application. A variable declared as Object is one that can subsequently be assigned (using the Set statement) to refer to any actual object recognized by the application.

```
Dim objDb As Object
Set objDb = OpenDatabase("c:\Vb5\Biblio.mdb")
```

When declaring object variables, try to use specific classes (such as `TextBox` instead of `Control` or, in the preceding case, `Database` instead of `Object`) rather than the generic Object. Visual Basic can resolve references to the properties and methods of objects with specific types before you run an application. This allows the application to perform faster at run time. Specific classes are listed in the Object Browser.

When working with other applications' objects, instead of using a Variant or the generic Object, declare objects as they are listed in the Classes list in the Object Browser. This ensures that Visual Basic recognizes the specific type of object you're referencing, allowing the reference to be resolved at run time.

For More Information For more information on creating and assigning objects and object variables, see "Creating Objects" later in this chapter.

Converting Data Types

Visual Basic provides several conversion functions you can use to convert values into a specific data type. To convert a value to Currency, for example, you use the CCur function:

```
PayPerWeek = CCur(hours * hourlyPay)
```

Conversion function	Converts an expression to
Cbool	Boolean
Cbyte	Byte
Ccur	Currency
Cdate	Date
CDbl	Double
Cint	Integer
CLng	Long
CSng	Single
CStr	String
Cvar	Variant
CVErr	Error

> **Note** Values passed to a conversion function must be valid for the destination data type or an error occurs. For example, if you attempt to convert a Long to an Integer, the Long must be within the valid range for the Integer data type.

For More Information See the *Language Reference* for a specific conversion function.

The Variant Data Type

A Variant variable is capable of storing all system-defined types of data. You don't have to convert between these types of data if you assign them to a Variant variable; Visual Basic automatically performs any necessary conversion. For example:

```
Dim SomeValue       ' Variant by default.
SomeValue = "17"    ' SomeValue contains "17" (a two-
```

```
                  ' character string).
SomeValue = SomeValue - 15       ' SomeValue now contains
                              ' the numeric value 2.
SomeValue = "U" & SomeValue      ' SomeValue now contains
                     ' "U2" (a two- character string).
```

While you can perform operations on Variant variables without much concern for the kind of data they contain, there are some traps you must avoid.

- If you perform arithmetic operations or functions on a Variant, the Variant must contain something that is a number. For details, see the section, "Numeric Values Stored in Variants," in "Advanced Variant Topics."
- If you are concatenating strings, use the & operator instead of the + operator. For details, see the section, "Strings Stored in Variants," in "Advanced Variant Topics."

In addition to being able to act like the other standard data types, Variants can also contain three special values: Empty, Null, and Error.

The Empty Value

Sometimes you need to know if a value has ever been assigned to a created variable. A Variant variable has the Empty value before it is assigned a value. The Empty value is a special value different from 0, a zero-length string (""), or the Null value. You can test for the Empty value with the IsEmpty function:

```
If IsEmpty(Z) Then Z = 0
```

When a Variant contains the Empty value, you can use it in expressions; it is treated as either 0 or a zero-length string, depending on the expression.

The Empty value disappears as soon as any value (including 0, a zero-length string, or Null) is assigned to a Variant variable. You can set a Variant variable back to Empty by assigning the keyword Empty to the Variant.

The Null Value

The Variant data type can contain another special value: Null. Null is commonly used in database applications to indicate unknown or missing data. Because of the way it is used in databases, Null has some unique characteristics:

- Expressions involving Null always result in Null. Thus, Null is said to "propagate" through expressions; if any part of the expression evaluates to Null, the entire expression evaluates to Null.
- Passing Null, a Variant containing Null, or an expression that evaluates to Null as an argument to most functions causes the function to return Null.
- Null values propagate through intrinsic functions that return Variant data types.

You can also assign Null with the Null keyword:

```
Z = Null
```

You can use the IsNull function to test if a Variant variable contains Null:

```
If IsNull(X) And IsNull(Y) Then
   Z = Null
Else
   Z = 0
End If
```

If you assign Null to a variable of any type other than Variant, a trappable error occurs. Assigning Null to a Variant variable doesn't cause an error, and Null will propagate through expressions involving Variant variables (though Null does not propagate through certain functions). You can return Null from any Function procedure with a Variant return value.

Variables are not set to Null unless you explicitly assign Null to them, so if you don't use Null in your application, you don't have to write code that tests for and handles it.

For More Information   For information on how to use Null in expressions, see "Null" in the *Language Reference*.

The Error Value

In a Variant, Error is a special value used to indicate that an error condition has occurred in a procedure. However, unlike for other kinds of errors, normal application-level error handling does not occur. This allows you, or the application itself, to take some alternative based on the error value. Error values are created by converting real numbers to error values using the CVErr function.

For More Information   For information on how to use the Error value in expressions, see "CVErr Function" in the *Language Reference*. For information on error handling, see "Debugging Your Code and Handling Errors." For additional information about the Variant data type, see "Advanced Variant Topics."

Send feedback to MSDN.   Look here for MSDN Online resources.

Advanced Variant Topics

Internal Representation of Values in Variants

Variant variables maintain an internal representation of the values that they store. This representation determines how Visual Basic treats these values when performing comparisons and other operations. When you assign a value to a Variant variable, Visual Basic uses the most compact representation that accurately records the value. Later operations may cause Visual Basic to change the representation it is using for a particular variable. (A Variant variable is not a variable with no type; rather, it is a variable that can freely change its type.) These internal representations correspond to the explicit data types discussed in "Data Types" earlier in this chapter.

Note   A variant always takes up 16 bytes, no matter what you store in it. Objects, strings,

and arrays are not physically stored in the Variant; in these cases, four bytes of the Variant are used to hold either an object reference, or a pointer to the string or array. The actual data are stored elsewhere.

Most of the time, you don't have to be concerned with what internal representation Visual Basic is using for a particular variable; Visual Basic handles conversions automatically. If you want to know what value Visual Basic is using, however, you can use the VarType function.

For example, if you store values with decimal fractions in a Variant variable, Visual Basic always uses the Double internal representation. If you know that your application does not need the high accuracy (and slower speed) that a Double value supplies, you can speed your calculations by converting the values to Single, or even to Currency:

```
If VarType(X) = 5 Then X = CSng(X)   ' Convert to Single.
```

With an array variable, the value of VarType is the sum of the array and data type return values. For example, this array contains Double values:

```
Private Sub Form_Click()
   Dim dblSample(2) As Double
   MsgBox VarType(dblSample)
End Sub
```

Future versions of Visual Basic may add additional Variant representations, so any code you write that makes decisions based on the return value of the VarType function should gracefully handle return values that are not currently defined.

For More Information For information about the VarType function, see "VarType Function" in the *Language Reference*. To read more about arrays, see "Arrays" later in this chapter. For details on converting data types, see "Data Types" earlier in this chapter.

Numeric Values Stored in Variants

When you store whole numbers in Variant variables, Visual Basic uses the most compact representation possible. For example, if you store a small number without a decimal fraction, the Variant uses an Integer representation for the value. If you then assign a larger number, Visual Basic will use a Long value or, if it is very large or has a fractional component, a Double value.

Sometimes you want to use a specific representation for a number. For example, you might want a Variant variable to store a numeric value as Currency to avoid round-off errors in later calculations. Visual Basic provides several conversion functions that you can use to convert values into a specific type (see "Converting Data Types" earlier in this chapter). To convert a value to Currency, for example, you use the CCur function:

```
PayPerWeek = CCur(hours * hourlyPay)
```

An error occurs if you attempt to perform a mathematical operation or function on a Variant that does not contain a number or something that can be interpreted as a number. For example, you cannot perform any arithmetic operations on the value U2 even though it contains a numeric character, because the entire value is not a valid number. Likewise, you cannot perform any calculations on the value 1040EZ; however, you can perform calculations on the values +10 or -1.7E6 because they are valid numbers. For this reason, you often want to determine if a Variant variable contains a value that can be used as a number. The

IsNumeric function performs this task:

```
Do
   anyNumber = InputBox("Enter a number")
Loop Until IsNumeric(anyNumber)
MsgBox "The square root is: " & Sqr(anyNumber)
```

When Visual Basic converts a representation that is not numeric (such as a string containing a number) to a numeric value, it uses the Regional settings (specified in the Windows Control Panel) to interpret the thousands separator, decimal separator, and currency symbol.

Thus, if the country setting in the Windows Control Panel is set to United States, Canada, or Australia, these two statements would return true:

```
IsNumeric("$100")
IsNumeric("1,560.50")
```

While these two statements would return false:

```
IsNumeric("DM100")
IsNumeric("1.560,50")
```

However, the reverse would be the case — the first two would return false and the second two true — if the country setting in the Windows Control Panel was set to Germany.

If you assign a Variant containing a number to a string variable or property, Visual Basic converts the representation of the number to a string automatically. If you want to explicitly convert a number to a string, use the CStr function. You can also use the Format function to convert a number to a string that includes formatting such as currency, thousands separator, and decimal separator symbols. The Format function automatically uses the appropriate symbols according to the Regional Settings Properties dialog box in the Windows Control Panel.

For More Information See "Format Function" and topics about the conversion functions in the *Language Reference*. For information on writing code for applications that will be distributed in foreign markets, see "International Issues."

Strings Stored in Variants

Generally, storing and using strings in Variant variables poses few problems. As mentioned earlier, however, sometimes the result of the + operator can be ambiguous when used with two Variant values. If both of the Variants contain numbers, the + operator performs addition. If both of the Variants contain strings, then the + operator performs string concatenation. But if one of the values is represented as a number and the other is represented as a string, the situation becomes more complicated. Visual Basic first attempts to convert the string into a number. If the conversion is successful, the + operator adds the two values; if unsuccessful, it generates a `Type mismatch` error.

To make sure that concatenation occurs, regardless of the representation of the value in the variables, use the & operator. For example, the following code:

```
Sub Form_Click ()
   Dim X, Y
   X = "6"
   Y = "7"
   Print X + Y, X & Y
```

```
   X = 6
   Print X + Y, X & Y
End Sub
```

produces this result on the form:

```
67      67
13      67
```

Note Visual Basic stores strings internally as Unicode. For more information on Unicode, see "International Issues."

Date/Time Values Stored in Variants

Variant variables can also contain date/time values. Several functions return date/time values. For example, DateSerial returns the number of days left in the year:

```
Private Sub Form_Click ()
   Dim rightnow, daysleft, hoursleft, minutesleft
   rightnow = Now   ' Now returns the current date/time.
   daysleft = Int(DateSerial(Year(rightnow) _
   + 1, 1, 1) - rightnow)
   hoursleft = 24 - Hour(rightnow)
   minutesleft = 60 - Minute(rightnow)
   Print daysleft & " days left in the year."
   Print hoursleft & " hours left in the day."
   Print minutesleft & " minutes left in the hour."
End Sub
```

You can also perform math on date/time values. Adding or subtracting integers adds or subtracts days; adding or subtracting fractions adds or subtracts time. Therefore, adding 20 adds 20 days, while subtracting 1/24 subtracts one hour.

The range for dates stored in Variant variables is January 1, 0100, to December 31, 9999. Calculations on dates don't take into account the calendar revisions prior to the switch to the Gregorian calendar, however, so calculations producing date values earlier than the year in which the Gregorian calendar was adopted (1752 in Britain and its colonies at that time; earlier or later in other countries) will be incorrect.

You can use date/time literals in your code by enclosing them with the number sign (#), in the same way you enclose string literals with double quotation marks (""). For example, you can compare a Variant containing a date/time value with a literal date:

```
If SomeDate > #3/6/93# Then
```

Similarly, you can compare a date/time value with a complete date/time literal:

```
If SomeDate > #3/6/93 1:20pm# Then
```

If you do not include a time in a date/time literal, Visual Basic sets the time part of the value to midnight (the start of the day). If you do not include a date in a date/time literal, Visual Basic sets the date part of the value to December 30, 1899.

Visual Basic accepts a wide variety of date and time formats in literals. These are all valid date/time values:

```
SomeDate = #3-6-93 13:20#
SomeDate = #March 27, 1993 1:20am#
SomeDate = #Apr-2-93#
SomeDate = #4 April 1993#
```

For More Information For information on handling dates in international formats, see "International Issues."

In the same way that you can use the IsNumeric function to determine if a Variant variable contains a value that can be considered a valid numeric value, you can use the IsDate function to determine if a Variant contains a value that can be considered a valid date/time value. You can then use the CDate function to convert the value into a date/time value.

For example, the following code tests the Text property of a text box with IsDate. If the property contains text that can be considered a valid date, Visual Basic converts the text into a date and computes the days left until the end of the year:

```
Dim SomeDate, daysleft
If IsDate(Text1.Text) Then
   SomeDate = CDate(Text1.Text)
   daysleft = DateSerial(Year(SomeDate) + _
   1, 1, 1) - SomeDate
   Text2.Text = daysleft & " days left in the year."
Else
   MsgBox Text1.Text & " is not a valid date."
End If
```

For More Information For information about the various date/time functions, see "Date Function" in the *Language Reference*.

Objects Stored in Variants

Objects can be stored in Variant variables. This can be useful when you need to gracefully handle a variety of data types, including objects. For example, all the elements in an array must have the same data type. Setting the data type of an array to Variant allows you to store objects alongside other data types in an array.

Send feedback to MSDN. Look here for MSDN Online resources.

Arrays

If you have programmed in other languages, you're probably familiar with the concept of arrays. Arrays allow you to refer to a series of variables by the same name and to use a number (an index) to tell them apart. This helps you create smaller and simpler code in many situations, because you can set up loops that deal efficiently with any number of cases by using the index number. Arrays have both upper and lower bounds, and the elements of the array are contiguous within those bounds. Because Visual Basic allocates space for each index number, avoid declaring an array larger than necessary.

Note The arrays discussed in this section are arrays of variables, declared in code. They

are different from the control arrays you specify by setting the Index property of controls at design time. Arrays of variables are always contiguous; unlike control arrays, you cannot load and unload elements from the middle of the array.

All the elements in an array have the same data type. Of course, when the data type is Variant, the individual elements may contain different kinds of data (objects, strings, numbers, and so on). You can declare an array of any of the fundamental data types, including user-defined types (described in the section, "Creating Your Own Data Types," in "More About Programming") and object variables (described in "Programming with Objects").

In Visual Basic there are two types of arrays: a *fixed-size array* which always remains the same size, and a *dynamic array* whose size can change at run-time. Dynamic arrays are discussed in more detail in the section "Dynamic Arrays" later in this chapter.

Declaring Fixed-Size Arrays

There are three ways to declare a fixed-size array, depending on the scope you want the array to have:

- To create a *public array*, use the Public statement in the Declarations section of a module to declare the array.
- To create a *module-level array*, use the Private statement in the Declarations section of a module to declare the array.
- To create a *local array*, use the Private statement in a procedure to declare the array.

Setting Upper and Lower Bounds

When declaring an array, follow the array name by the upper bound in parentheses. The upper bound cannot exceed the range of a Long data type (-2,147,483,648 to 2,147,483,647). For example, these array declarations can appear in the Declarations section of a module:

```
Dim Counters(14) As Integer         ' 15 elements.
Dim Sums(20) As Double             ' 21 elements.
```

To create a public array, you simply use Public in place of Dim:

```
Public Counters(14) As Integer
Public Sums(20) As Double
```

The same declarations within a procedure use Dim:

```
Dim Counters(14) As Integer
Dim Sums(20) As Double
```

The first declaration creates an array with 15 elements, with index numbers running from 0 to 14. The second creates an array with 21 elements, with index numbers running from 0 to 20. The default lower bound is 0.

To specify a lower bound, provide it explicitly (as a Long data type) using the To keyword:

```
Dim Counters(1 To 15) As Integer
```

```
Dim Sums(100 To 120) As String
```

In the preceding declarations, the index numbers of `Counters` range from 1 to 15, and the index numbers of `Sums` range from 100 to 120.

Arrays that Contain Other Arrays

It's possible to create a Variant array, and populate it with other arrays of different data types. The following code creates two arrays, one containing integers and the other strings. It then declares a third Variant array and populates it with the integer and string arrays.

```
Private Sub Command1_Click()
   Dim intX As Integer   ' Declare counter variable.
   ' Declare and populate an integer array.
   Dim countersA(5) As Integer
      For intX = 0 To 4
         countersA(intX) = 5
      Next intX
   ' Declare and populate a string array.
      Dim countersB(5) As String
         For intX = 0 To 4
            countersB(intX) = "hello"
         Next intX
   Dim arrX(2) As Variant   ' Declare a new two-member
                           ' array.
      arrX(1) = countersA()   ' Populate the array with
                              ' other arrays.
      arrX(2) = countersB()
      MsgBox arrX(1)(2)   ' Display a member of each
                        ' array.
      MsgBox arrX(2)(3)
End Sub
```

Multidimensional Arrays

Sometimes you need to keep track of related information in an array. For example, to keep track of each pixel on your computer screen, you need to refer to its X and Y coordinates. This can be done using a multidimensional array to store the values.

With Visual Basic, you can declare arrays of multiple dimensions. For example, the following statement declares a two-dimensional 10-by-10 array within a procedure:

```
Static MatrixA(9, 9) As Double
```

Either or both dimensions can be declared with explicit lower bounds:

```
Static MatrixA(1 To 10, 1 To 10) As Double
```

You can extend this to more than two dimensions. For example:

```
Dim MultiD(3, 1 To 10, 1 To 15)
```

This declaration creates an array that has three dimensions with sizes 4 by 10 by 15. The total number of elements is the product of these three dimensions, or 600.

Note When you start adding dimensions to an array, the total storage needed by the array

increases dramatically, so use multidimensional arrays with care. Be especially careful with Variant arrays, because they are larger than other data types.

Using Loops to Manipulate Arrays

You can efficiently process a multidimensional array by using nested For loops. For example, these statements initialize every element in `MatrixA` to a value based on its location in the array:

```
Dim I As Integer, J As Integer
Static MatrixA(1 To 10, 1 To 10) As Double
For I = 1 To 10
   For J = 1 To 10
      MatrixA(I, J) = I * 10 + J
   Next J
Next I
```

For More Information For information about loops, see "Loop Structures" later in this chapter.

Send feedback to MSDN. Look here for MSDN Online resources.

Dynamic Arrays

Sometimes you may not know exactly how large to make an array. You may want to have the capability of changing the size of the array at run time.

A dynamic array can be resized at any time. Dynamic arrays are among the most flexible and convenient features in Visual Basic, and they help you to manage memory efficiently. For example, you can use a large array for a short time and then free up memory to the system when you're no longer using the array.

The alternative is to declare an array with the largest possible size and then ignore array elements you don't need. However, this approach, if overused, might cause the operating environment to run low on memory.

To create a dynamic array

1. Declare the array with a Public statement (if you want the array to be public) or Dim statement at the module level (if you want the array to be module level), or a Static or Dim statement in a procedure (if you want the array to be local). You declare the array as dynamic by giving it an empty dimension list.

   ```
   Dim DynArray()
   ```

2. Allocate the actual number of elements with a ReDim statement.

   ```
   ReDim DynArray(X + 1)
   ```

The ReDim statement can appear only in a procedure. Unlike the Dim and Static statements, ReDim is an executable statement — it makes the application carry out an action at run time.

The ReDim statement supports the same syntax used for fixed arrays. Each ReDim can change the number of elements, as well as the lower and upper bounds, for each dimension. However, the number of dimensions in the array cannot change.

```
ReDim DynArray(4 to 12)
```

For example, the dynamic array `Matrix1` is created by first declaring it at the module level:

```
Dim Matrix1() As Integer
```

A procedure then allocates space for the array:

```
Sub CalcValuesNow ()
   .
   .
   .
   ReDim Matrix1(19, 29)
End Sub
```

The ReDim statement shown here allocates a matrix of 20 by 30 integers (at a total size of 600 elements). Alternatively, the bounds of a dynamic array can be set using variables:

```
ReDim Matrix1(X, Y)
```

> **Note** You can assign strings to resizable arrays of bytes. An array of bytes can also be assigned to a variable-length string. Be aware that the number of bytes in a string varies among platforms. On Unicode platforms the same string contains twice as many bytes as it does on a non-Unicode platform.

Preserving the Contents of Dynamic Arrays

Each time you execute the ReDim statement, all the values currently stored in the array are lost. Visual Basic resets the values to the Empty value (for Variant arrays), to zero (for numeric arrays), to a zero-length string (for string arrays), or to Nothing (for arrays of objects).

This is useful when you want to prepare the array for new data, or when you want to shrink the size of the array to take up minimal memory. Sometimes you may want to change the size of the array without losing the data in the array. You can do this by using ReDim with the Preserve keyword. For example, you can enlarge an array by one element without losing the values of the existing elements using the UBound function to refer to the upper bound:

```
ReDim Preserve DynArray(UBound(DynArray) + 1)
```

Only the upper bound of the last dimension in a multidimensional array can be changed when you use the Preserve keyword; if you change any of the other dimensions, or the lower bound of the last dimension, a run-time error occurs. Thus, you can use code like this:

```
ReDim Preserve Matrix(10, UBound(Matrix, 2) + 1)
```

But you cannot use this code:

```
ReDim Preserve Matrix(UBound(Matrix, 1) + 1, 10)
```

For More Information For information about dynamic arrays, see "ReDim Statement" in the *Language Reference*. To learn more about object arrays, see "Programming with Objects."

Introduction to Procedures

You can simplify programming tasks by breaking programs into smaller logical components. These components — called *procedures* — can then become building blocks that let you enhance and extend Visual Basic.

Procedures are useful for condensing repeated or shared tasks, such as frequently used calculations, text and control manipulation, and database operations.

There are two major benefits of programming with procedures:

- Procedures allow you to break your programs into discrete logical units, each of which you can debug more easily than an entire program without procedures.
- Procedures used in one program can act as building blocks for other programs, usually with little or no modification.

There are several types of procedures used in Visual Basic:

- Sub procedures do not return a value.
- Function procedures return a value.
- Property procedures can return and assign values, and set references to objects.

For More Information Property procedures are discussed in "Programming with Objects."

To learn more about Sub and Function procedures, see the following topics:

- Sub Procedures A discussion of Sub procedures and how to use them.
- Function Procedures An introduction to Function procedures and how to use them.
- Working with Procedures An introduction to calling procedures from within an application.
- Passing Arguments to Procedures A discussion of passing data to procedures using arguments.

Sub Procedures

A Sub procedure is a block of code that is executed in response to an event. By breaking the code in a module into Sub procedures, it becomes much easier to find or modify the code in your application.

The syntax for a Sub procedure is:

```
[Private|Public][Static]Sub procedurename (arguments)
statements

End Sub
```

Each time the procedure is called, the *statements* between Sub and End Sub are executed. Sub procedures can be placed in standard modules, class modules, and form modules. Sub procedures are by default Public in all modules, which means they can be called from anywhere in the application.

The *arguments* for a procedure are like a variable declaration, declaring values that are passed in from the calling procedure.

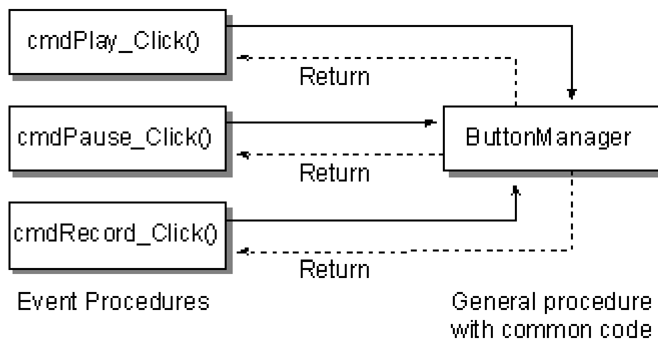
In Visual Basic, it's useful to distinguish between two types of Sub procedures, *general procedures* and *event procedures*.

General Procedures

A general procedure tells the application how to perform a specific task. Once a general procedure is defined, it must be specifically invoked by the application. By contrast, an event procedure remains idle until called upon to respond to events caused by the user or triggered by the system.

Why create general procedures? One reason is that several different event procedures might need the same actions performed. A good programming strategy is to put common statements in a separate procedure (a general procedure) and have your event procedures call it. This eliminates the need to duplicate code and also makes the application easier to maintain. For example, the VCR sample application uses a general procedure called by the click events for several different scroll buttons. Figure 5.7 illustrates the use of a general procedure. Code in the Click events calls the ButtonManager Sub procedure, which runs its own code, and then returns control to the Click event procedure.

Figure 5.7 How general procedures are called by event procedures



Event Procedures

When an object in Visual Basic recognizes that an event has occurred, it automatically invokes the event procedure using the name corresponding to the event. Because the name establishes an association between the object and the code, event procedures are said to be attached to forms and controls.

- An event procedure for a control combines the control's actual name (specified in the Name property), an underscore (_), and the event name. For instance, if you want a command button named cmdPlay to invoke an event procedure when it is clicked, use the procedure cmdPlay_Click.

- An event procedure for a form combines the word "Form," an underscore, and the event name. If you want a form to invoke an event procedure when it is clicked, use the procedure Form_Click. (Like controls, forms do have unique names, but they are not used in the names of event procedures.) If you are using the MDI form, the event procedure combines the word "MDIForm," an underscore, and the event name, as in MDIForm_Load.

All event procedures use the same general syntax.

Syntax for a control event	Syntax for a form event

Private Sub *controlname_eventname* (*arguments*)
  statements
 	
End Sub
	Private Sub Form_*eventname* (*arguments*)
	  statements
	End Sub

Although you can write event procedures from scratch, it's easier to use the code procedures provided by Visual Basic, which automatically include the correct procedure names. You can select a template in the Code Editor window by selecting an object from the Object box and then selecting a procedure from the Procedure box.

It's also a good idea to set the Name property of your controls before you start writing event procedures for them. If you change the name of a control after attaching a procedure to it, you must also change the name of the procedure to match the new name of the control. Otherwise, Visual Basic won't be able to match the control to the procedure. When a procedure name does not match a control name, it becomes a general procedure.

For More Information   Visual Basic recognizes a variety of events for each kind of form and

control. For explanations of all events, see the *Language Reference*.

Send feedback to MSDN. Look here for MSDN Online resources.

Function Procedures

Visual Basic includes built-in, or intrinsic functions, like Sqr, Cos or Chr. In addition, you can use the Function statement to write your own Function procedures.

The syntax for a Function procedure is:

[Private|Public][Static]Function *procedurename* (*arguments*) [**As** *type*]
statements

End Function

Like a Sub procedure, a Function procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. Unlike a Sub procedure, a Function procedure can return a value to the calling procedure. There are three differences between Sub and Function procedures:

- Generally, you call a function by including the function procedure name and arguments on the right side of a larger statement or expression (*returnvalue* = *function()*).
- Function procedures have data types, just as variables do. This determines the type of the return value. (In the absence of an As clause, the type is the default Variant type.)
- You return a value by assigning it to the *procedurename* itself. When the Function procedure returns a value, this value can then become part of a larger expression.

For example, you could write a function that calculates the third side, or hypotenuse, of a right triangle, given the values for the other two sides:

```
Function Hypotenuse (A As Integer, B As Integer) _
As String
   Hypotenuse = Sqr(A ^ 2 + B ^ 2)
End Function
```

You call a Function procedure the same way you call any of the built-in functions in Visual Basic:

```
Label1.Caption = Hypotenuse(CInt(Text1.Text), _
CInt(Text2.Text))
strX = Hypotenuse(Width, Height)
```

For More Information For additional details about the Function procedure, see "Function Statement" in the *Language Reference*. The techniques for calling all types of procedures are discussed in the section, "Calling Procedures," later in this chapter.

Working with Procedures

Creating New Procedures

To create a new general procedure

- Type a procedure heading in the Code window and press ENTER. The procedure heading can be as simple as Sub or Function followed by a name. For example, you can enter either of the following:

  ```
  Sub UpdateForm ()
  Function GetCoord ()
  ```

  Visual Basic responds by completing the template for the new procedure.

Selecting Existing Procedures

To view a procedure in the current module

- To view an existing general procedure, select "(General)" from the Object box in the Code window, and then select the procedure in the Procedure box.

  –or–

  To view an event procedure, select the appropriate object from the Object box in the Code window, and then select the event in the Procedure box.

To view a procedure in another module

1. From the **View** menu, choose **Object Browser**.
2. Select the project from the **Project/Library** box.
3. Select the module from the **Classes** list, and the procedure from the **Members of** list.
4. Choose **View Definition**.

Calling Procedures

The techniques for calling procedures vary, depending on the type of procedure, where it's located, and how it's used in your application. The following sections describe how to call Sub and Function procedures.

Calling Sub Procedures

A Sub procedure differs from a Function procedure in that a Sub procedure cannot be called by using its name within an expression. A call to a Sub is a stand-alone statement. Also, a Sub does not return a value in its name as does a function. However, like a Function, a Sub can modify the values of any variables passed to it.

There are two ways to call a Sub procedure:

```
' Both of these statements call a Sub named MyProc.
Call MyProc (FirstArgument, SecondArgument)
MyProc FirstArgument, SecondArgument
```

Note that when you use the Call syntax, arguments must be enclosed in parentheses. If you omit the Call keyword, you must also omit the parentheses around the arguments.

Calling Function Procedures

Usually, you call a function procedure you've written yourself the same way you call an intrinsic Visual Basic function like Abs; that is, by using its name in an expression:

```
' All of the following statements would call a function
' named ToDec.
Print 10 * ToDec
X = ToDec
If ToDec = 10 Then Debug.Print "Out of Range"
X = AnotherFunction(10 * ToDec)
```

It's also possible to call a function just like you would call a Sub procedure. The following statements both call the same function:

```
Call Year(Now)
Year Now
```

When you call a function this way, Visual Basic throws away the return value.

Calling Procedures in Other Modules

Public procedures in other modules can be called from anywhere in the project. You might need to specify the module that contains the procedure you're calling. The techniques for doing this vary, depending on whether the procedure is located in a form, class, or standard module.

Procedures in Forms

All calls from outside the form module must point to the form module containing the procedure. If a procedure named SomeSub is in a form module called Form1, then you can call the procedure in Form1 by using this statement:

```
Call Form1.SomeSub(arguments)
```

Procedures in Class Modules

Like calling a procedure in a form, calling a procedure in a class module requires that the call to the procedure be qualified with a variable that points to an instance of the class. For example, DemoClass is an instance of a class named Class1:

```
Dim DemoClass as New Class1
DemoClass.SomeSub
```

However, unlike a form, the class name cannot be used as the qualifier when referencing an instance of the class. The instance of the class must be first be declared as an object variable (in this case, DemoClass) and referenced by the variable name.

For More Information You can find details on object variables and class modules in "Programming with Objects."

Procedures in Standard Modules

If a procedure name is unique, you don't need to include the module name in the call. A call from inside or outside the module will refer to that unique procedure. A procedure is unique if it appears only in one place.

If two or more modules contain a procedure with the same name, you may need to qualify it with the module name. A call to a common procedure from the same module runs the procedure in that module. For example, with a procedure named CommonName in Module1 and Module2, a call to CommonName from Module2 will run the CommonName procedure in Module2, not the CommonName procedure in Module1.

A call to a common procedure name from another module must specify the intended module. For example, if you want to call the CommonName procedure in Module2 from Module1, use:

```
Module2.CommonName(arguments)
```

Send feedback to MSDN. Look here for MSDN Online resources.

Passing Arguments to Procedures

Usually the code in a procedure needs some information about the state of the program to do its job. This information consists of variables passed to the procedure when it is called. When a variable is passed to a procedure, it is called an *argument*.

Argument Data Types

The arguments for procedures you write have the Variant data type by default. However, you can declare other data types for arguments. For example, the following function accepts a string and an integer:

```
Function WhatsForLunch(WeekDay As String, Hour _
As Integer) As String
   ' Returns a lunch menu based on the day and time.
   If WeekDay = "Friday" then
      WhatsForLunch = "Fish"
   Else
      WhatsForLunch = "Chicken"
   End If
```

```
   If Hour > 4 Then WhatsForLunch = "Too late"
End Function
```

For More Information Details on Visual Basic data types are presented earlier in this chapter. You can also see the *Language Reference* for specific data types.

Passing Arguments By Value

Only a copy of a variable is passed when an argument is passed by value. If the procedure changes the value, the change affects only the copy and not the variable itself. Use the ByVal keyword to indicate an argument passed by value.

For example:

```
Sub PostAccounts(ByVal intAcctNum as Integer)
   .
   . ' Place statements here.
   .
End Sub
```

Passing Arguments By Reference

Passing arguments by reference gives the procedure access to the actual variable contents in its memory address location. As a result, the variable's value can be permanently changed by the procedure to which it is passed. Passing by reference is the default in Visual Basic.

If you specify a data type for an argument passed by reference, you must pass a value of that type for the argument. You can work around this by passing an expression, rather than a data type, for an argument. Visual Basic evaluates an expression and passes it as the required type if it can.

The simplest way to turn a variable into an expression is to enclose it in parentheses. For example, to pass a variable declared as an integer to a procedure expecting a string as an argument, you would do the following:

```
Sub CallingProcedure()
   Dim intX As Integer
   intX = 12 * 3
   Foo(intX)
End Sub

Sub Foo(Bar As String)
   MsgBox Bar   'The value of Bar is the string "36".
End Sub
```

Using Optional Arguments

You can specify arguments to a procedure as optional by placing the Optional keyword in the argument list. If you specify an optional argument, all subsequent arguments in the argument list must also be optional and declared with the Optional keyword. The two pieces of sample code below assume there is a form with a command button and list box.

For example, this code provides all optional arguments:

```
Dim strName As String
Dim strAddress As String

Sub ListText(Optional x As String, Optional y _
As String)
   List1.AddItem x
   List1.AddItem y
End Sub

Private Sub Command1_Click()
   strName = "yourname"
   strAddress = 12345   ' Both arguments are provided.
   Call ListText(strName, strAddress)
End Sub
```

This code, however, does not provide all optional arguments:

```
Dim strName As String
Dim varAddress As Variant

Sub ListText(x As String, Optional y As Variant)
   List1.AddItem x
   If Not IsMissing(y) Then
      List1.AddItem y
   End If
End Sub

Private Sub Command1_Click()
   strName = "yourname"   ' Second argument is not
                        ' provided.
   Call ListText(strName)
End Sub
```

In the case where an optional argument is not provided, the argument is actually assigned as a variant with the value of Empty. The example above shows how to test for missing optional arguments using the IsMissing function.

Providing a Default for an Optional Argument

It's also possible to specify a default value for an optional argument. The following example returns a default value if the optional argument isn't passed to the function procedure:

```
Sub ListText(x As String, Optional y As _
Integer = 12345)
   List1.AddItem x
   List1.AddItem y
End Sub

Private Sub Command1_Click()
   strName = "yourname"   ' Second argument is not
                        ' provided.
   Call ListText(strName)   ' Adds "yourname" and
                           ' "12345".
End Sub
```

Using an Indefinite Number of Arguments

Generally, the number of arguments in the procedure call must be the same as in the procedure specification. Using the ParamArray keyword allows you to specify that a procedure will accept an arbitrary number of arguments. This allows you to write functions like Sum:

```
Dim x As Integer
Dim y As Integer
Dim intSum As Integer

Sub Sum(ParamArray intNums())
   For Each x In intNums
      y = y + x
   Next x
   intSum = y
End Sub

Private Sub Command1_Click()
   Sum 1, 3, 5, 7, 8
   List1.AddItem intSum
End Sub
```

Creating Simpler Statements with Named Arguments

For many built-in functions, statements, and methods, Visual Basic provides the option of using *named arguments* as a shortcut for typing argument values. With named arguments, you can provide any or all of the arguments, in any order, by assigning a value to the named argument. You do this by typing the argument name plus a colon followed by an equal sign and the value (MyArgument:= "SomeValue") and placing that assignment in any sequence delimited by commas. Notice that the arguments in the following example are in the reverse order of the expected arguments:

```
Function ListText(strName As String, Optional strAddress As String)
   List1.AddItem strName
   List2.AddItem strAddress
End Sub

Private Sub Command1_Click()
   ListText strAddress:="12345", strName:="Your Name"
End Sub
```

This is especially useful if your procedures have several optional arguments that you do not always need to specify.

Determining Support for Named Arguments

To determine which functions, statements, and methods support named arguments, use the AutoQuickInfo feature in the Code window, check the Object Browser, or see the *Language Reference*. Consider the following when working with named arguments:

- Named arguments are not supported by methods on objects in the Visual Basic (VB) object library. They are supported by all language keywords in the Visual Basic for applications (VBA) object library.

- In syntax, named arguments are shown as bold and italic. All other arguments are shown in italic only.

Important You cannot use named arguments to avoid entering required arguments. You can omit only the optional arguments. For Visual Basic (VB) and Visual Basic for applications (VBA) object libraries, the Object Browser encloses optional arguments with square brackets [ ].

For More Information See "ByVal," "ByRef," "Optional," and "ParamArray" in the *Language*

Reference.

Introduction to Control Structures

Control structures allow you to control the flow of your program's execution. If left unchecked by control-flow statements, a program's logic will flow through statements from left to right, and top to bottom. While some very simple programs can be written with only this unidirectional flow, and while some flow can be controlled by using operators to regulate precedence of operations, most of the power and utility of any programming language comes from its ability to change statement order with structures and loops.

To learn more about specific control structures, see the following topics:

- Decision Structures   An introduction to decision structures used for branching.
- Loop Structures   An introduction to loop structures used to repeat processes.
- Working with Control Structures   The basics of working with control structures in your code.

Decision Structures

Visual Basic procedures can test conditions and then, depending on the results of that test, perform different operations. The decision structures that Visual Basic supports include:

- If...Then
- If...Then...Else
- Select Case

If...Then

Use an If...Then structure to execute one or more statements conditionally. You can use either a single-line syntax or a multiple-line *block* syntax:

If *condition* **Then** *statement*

If *condition* **Then**
statements

End If

The *condition* is usually a comparison, but it can be any expression that evaluates to a numeric value. Visual Basic interprets this value as True or False; a zero numeric value is False, and any nonzero numeric value is considered True. If *condition* is True, Visual Basic executes all the *statements* following the Then keyword. You can use either single-line or multiple-line syntax to execute just one statement conditionally (these two examples are equivalent):

```
If anyDate < Now Then anyDate = Now

If anyDate < Now Then
   anyDate = Now
End If
```

Notice that the single-line form of If...Then does not use an End If statement. If you want to execute more than one line of code when *condition* is True, you must use the multiple-line block If...Then...End If syntax.

```
If anyDate < Now Then
   anyDate = Now
   Timer1.Enabled = False      ' Disable timer control.
End If
```

If...Then...Else

Use an If...Then...Else block to define several blocks of statements, one of which will execute:

If *condition1* **Then**
[*statementblock-1*]
[**ElseIf** *condition2* **Then**
[*statementblock-2*]] ...
[**Else**
[*statementblock-n*]]

End If

Visual Basic first tests *condition1*. If it's False, Visual Basic proceeds to test *condition2*, and so on, until it finds a True condition. When it finds a True condition, Visual Basic executes the corresponding statement block and then executes the code following the End If. As an option, you can include an Else statement block, which Visual Basic executes if none of the conditions are True.

If...Then…ElseIf is really just a special case of If...Then...Else. Notice that you can have any number of ElseIf clauses, or none at all. You can include an Else clause regardless of whether you have ElseIf clauses.

For example, your application could perform different actions depending on which control in a menu control array was clicked:

```
Private Sub mnuCut_Click (Index As Integer)
```

```
   If Index = 0 Then            ' Cut command.
      CopyActiveControl         ' Call general procedures.
      ClearActiveControl
   ElseIf Index = 1 Then      ' Copy command.
      CopyActiveControl
   ElseIf Index = 2 Then      ' Clear command.
      ClearActiveControl
   Else                        ' Paste command.
      PasteActiveControl
   End If
End Sub
```

Notice that you can always add more ElseIf parts to your If...Then structure. However, this syntax can get tedious to write when each ElseIf compares the same expression to a different value. For this situation, you can use a Select Case decision structure.

For More Information See "If...Then...Else Statement" in the *Language Reference*.

Select Case

Visual Basic provides the Select Case structure as an alternative to If...Then...Else for selectively executing one block of statements from among multiple blocks of statements. A Select Case statement provides capability similar to the If...Then...Else statement, but it makes code more readable when there are several choices.

A Select Case structure works with a single test expression that is evaluated once, at the top of the structure. Visual Basic then compares the result of this expression with the values for each Case in the structure. If there is a match, it executes the block of statements associated with that Case:

Select Case *testexpression*
[**Case** *expressionlist1*
[*statementblock-1*]]
[**Case** *expressionlist2*
[*statementblock-2*]]
.
.
.
[**Case Else**
[*statementblock-n*]]

End Select

Each *expressionlist* is a list of one or more values. If there is more than one value in a single list, the values are separated by commas. Each *statementblock* contains zero or more statements. If more than one Case matches the test expression, only the statement block associated with the first matching Case will execute. Visual Basic executes statements in the Case Else clause (which is optional) if none of the values in the expression lists matches the test expression.

For example, suppose you added another command to the Edit menu in the If...Then...Else example. You could add another ElseIf clause, or you could write the function with Select Case:

```
Private Sub mnuCut_Click (Index As Integer)
```

```
   Select Case Index
      Case 0                   ' Cut command.
         CopyActiveControl      ' Call general procedures.
         ClearActiveControl
      Case 1                   ' Copy command.
         CopyActiveControl
      Case 2                   ' Clear command.
         ClearActiveControl
      Case 3                   ' Paste command.
         PasteActiveControl
      Case Else
         frmFind.Show          ' Show Find dialog box.
   End Select
End Sub
```

Notice that the Select Case structure evaluates an expression once at the top of the structure. In contrast, the If...Then...Else structure can evaluate a different expression for each ElseIf statement. You can replace an If...Then...Else structure with a Select Case structure only if the If statement and each ElseIf statement evaluates the same expression.

Send feedback to MSDN.   Look here for MSDN Online resources.

Loop Structures

Loop structures allow you to execute one or more lines of code repetitively. The loop structures that Visual Basic supports include:

- Do...Loop
- For...Next
- For Each...Next

Do...Loop

Use a Do loop to execute a block of statements an indefinite number of times. There are several variations of the Do...Loop statement, but each evaluates a numeric condition to determine whether to continue execution. As with If...Then, the *condition* must be a value or expression that evaluates to False (zero) or to True (nonzero).

In the following Do...Loop, the *statements* execute as long as the *condition* is True:

```
Do While condition
statements

Loop
```

When Visual Basic executes this Do loop, it first tests *condition*. If *condition* is False (zero), it skips past all the statements. If it's True (nonzero), Visual Basic executes the statements and

then goes back to the Do While statement and tests the condition again.

Consequently, the loop can execute any number of times, as long as *condition* is nonzero or True. The statements never execute if *condition* is initially False. For example, this procedure counts the occurrences of a target string within another string by looping as long as the target string is found:

```
Function CountStrings (longstring, target)
   Dim position, count
   position = 1
   Do While InStr(position, longstring, target)
      position = InStr(position, longstring, target)_
       + 1
      count = count + 1
   Loop
   CountStrings = count
End Function
```

If the target string doesn't occur in the other string, then InStr returns 0, and the loop doesn't execute.

Another variation of the Do...Loop statement executes the statements first and then tests *condition* after each execution. This variation guarantees at least one execution of *statements*:

Do
statements

Loop While *condition*

Two other variations are analogous to the previous two, except that they loop as long as *condition* is False rather than True.

Loop zero or more times	Loop at least once

Do Until *condition*
  statements
Loop
	Do
	  statements
	Loop Until *condition*

For...Next

Do loops work well when you don't know how many times you need to execute the statements in the loop. When you know you must execute the statements a specific number of times, however, a For…Next loop is a better choice. Unlike a Do loop, a For loop uses a variable called a counter that increases or decreases in value during each repetition of the loop. The syntax is:

For *counter* = *start* **To** *end* [**Step** *increment*]
statements

Next [*counter*]

The arguments *counter*, *start*, *end*, and *increment* are all numeric.

Note The *increment* argument can be either positive or negative. If *increment* is positive,

start must be less than or equal to *end* or the statements in the loop will not execute. If *increment* is negative, *start* must be greater than or equal to *end* for the body of the loop to execute. If Step isn't set, then *increment* defaults to 1.

In executing the For loop, Visual Basic:

1. Sets *counter* equal to *start*.
2. Tests to see if *counter* is greater than *end*. If so, Visual Basic exits the loop.
   (If *increment* is negative, Visual Basic tests to see if *counter* is less than *end*.)
3. Executes the *statements*.
4. Increments *counter* by 1 or by *increment*, if it's specified.
5. Repeats steps 2 through 4.

This code prints the names of all the available Screen fonts:

```
Private Sub Form_Click ()
   Dim I As Integer
   For i = 0 To Screen.FontCount
      Print Screen.Fonts(i)
   Next
End Sub
```

In the VCR sample application, the HighlightButton procedure uses a For...Next loop to step through the controls collection of the VCR form and show the appropriate Shape control:

```
Sub HighlightButton(MyControl As Variant)
   Dim i As Integer
   For i = 0 To frmVCR.Controls.Count - 1
      If TypeOf frmVCR.Controls(i) Is Shape Then
         If frmVCR.Controls(i).Name = MyControl Then
            frmVCR.Controls(i).Visible = True
         Else
            frmVCR.Controls(i).Visible = False
         End If
      End If
   Next
End Sub
```

For Each...Next

A For Each...Next loop is similar to a For...Next loop, but it repeats a group of statements for each element in a collection of objects or in an array instead of repeating the statements a specified number of times. This is especially helpful if you don't know how many elements are in a collection.

Here is the syntax for the For Each...Next loop:

```
For Each element In group
statements

Next element
```

For example, the following Sub procedure opens Biblio.mdb and adds the name of each table to a list box.

```
Sub ListTableDefs()
   Dim objDb As Database
   Dim MyTableDef as TableDef
   Set objDb = OpenDatabase("c:\vb\biblio.mdb", _
   True, False)
   For Each MyTableDef In objDb.TableDefs()
      List1.AddItem MyTableDef.Name
   Next MyTableDef
End Sub
```

Keep the following restrictions in mind when using For Each...Next:

- For collections, *element* can only be a Variant variable, a generic Object variable, or an object listed in the Object Browser.
- For arrays, *element* can only be a Variant variable.
- You cannot use For Each...Next with an array of user-defined types because a Variant cannot contain a user-defined type.

Send feedback to MSDN.   Look here for MSDN Online resources.

Working with Control Structures

Nested Control Structures

You can place control structures inside other control structures (such as an If...Then block within a For...Next loop). A control structure placed inside another control structure is said to be *nested*.

Control structures in Visual Basic can be nested to as many levels as you want. It's common practice to make nested decision structures and loop structures more readable by indenting the body of the decision structure or loop.

For example, this procedure prints all the font names that are common to both the Printer and Screen:

```
Private Sub Form_Click()
   Dim SFont, PFont
   For Each SFont In Screen.Fonts()
      For Each PFont In Printer.Fonts()
         If SFont = PFont Then
            Print SFont
         End If
      Next PFont
   Next SFont
End Sub
```

Notice that the first Next closes the inner For loop and the last For closes the outer For loop. Likewise, in nested If statements, the End If statements automatically apply to the nearest prior If statement. Nested Do...Loop structures work in a similar fashion, with the innermost Loop statement matching the innermost Do statement.

Exiting a Control Structure

The Exit statement allows you to exit directly from a For loop, Do loop, Sub procedure, or Function procedure. The syntax for the Exit statement is simple: Exit For can appear as many times as needed inside a For loop, and Exit Do can appear as many times as needed inside a Do loop:

For *counter* **=** *start* **To** *end* [**Step** *increment*]
[*statementblock*]
[**Exit For**]
[*statementblock*]

Next [*counter*[, *counter*] [,...]]

Do [{**While** | **Until**} *condition*]
[*statementblock*]
[**Exit Do**]
[*statementblock*]

Loop

The Exit Do statement works with all versions of the Do loop syntax.

Exit For and Exit Do are useful because sometimes it's appropriate to quit a loop immediately, without performing any further iterations or statements within the loop. For example, in the previous example that printed the fonts common to both the Screen and Printer, the code continues to compare Printer fonts against a given Screen font even when a match has already been found with an earlier Printer font. A more efficient version of the function would exit the loop as soon as a match is found:

```
Private Sub Form_Click()
   Dim SFont, PFont
   For Each SFont In Screen.Fonts()
      For Each PFont In Printer.Fonts()
         If SFont = PFont Then
            Print Sfont
            Exit For            ' Exit inner loop.
         End If
      Next PFont
   Next SFont
End Sub
```

As this example illustrates, an Exit statement almost always appears inside an If statement or Select Case statement nested inside the loop.

When you use an Exit statement to break out of a loop, the value of the counter variable differs, depending on how you leave the loop:

- When you complete a loop, the counter variable contains the value of the upper bound

plus the step.

- When you exit a loop prematurely, the counter variable retains its value subject to the usual rules on scope.

- When you iterate off the end of a collection, the counter variable contains Nothing if it's an Object data type, and contains Empty if it's a Variant data type.

Exiting a Sub or Function Procedure

You can also exit a procedure from within a control structure. The syntax of Exit Sub and Exit Function is similar to that of Exit For and Exit Do in the previous section, "Exiting a Control Structure." Exit Sub can appear as many times as needed, anywhere within the body of a Sub procedure. Exit Function can appear as many times as needed, anywhere within the body of a Function procedure.

Exit Sub and Exit Function are useful when the procedure has done everything it needs to do and can return immediately. For example, if you want to change the previous example so it prints only the first common Printer and Screen font it finds, you would use Exit Sub:

```
Private Sub Form_Click()
   Dim SFont, PFont
   For Each SFont In Screen.Fonts()
      For Each PFont In Printer.Fonts()
         If SFont = PFont Then
            Print Sfont
            Exit Sub            ' Exit the procedure.
         End If
      Next PFont
   Next SFont
End Sub
```

Send feedback to MSDN.   Look here for MSDN Online resources.

Working with Objects

When you create an application in Visual Basic, you work with objects. You can use objects provided by Visual Basic — such as controls, forms, and data access objects. You can also control other applications' objects from within your Visual Basic application. You can even create your own objects, and define additional properties and methods for them.

The following topics discuss objects in detail:

- What is an Object?   An introduction to the concept of objects.
- What Can You Do with Objects?   A discussion of some ways that objects can be used in an application.
- The Basics of Working with Objects   An introduction to the properties and methods exposed by objects.

- How are Objects Related to Each Other? A discussion of object hierarchies, collections, and containers.
- Creating Objects A discussion of how objects can be created and used at run time.

Send feedback to MSDN. Look here for MSDN Online resources.

What is an Object?

An object is a combination of code and data that can be treated as a unit. An object can be a piece of an application, like a control or a form. An entire application can also be an object. The following table describes examples of the types of objects you can use in Visual Basic.

Example	Description
Command button	Controls on a form, such as command buttons and frames, are objects.
Form	Each form in a Visual Basic project is a separate object.
Database	Databases are objects, and contain other objects, like fields and indexes.
Chart	A chart in Microsoft Excel is an object.

Where do Objects Come From?

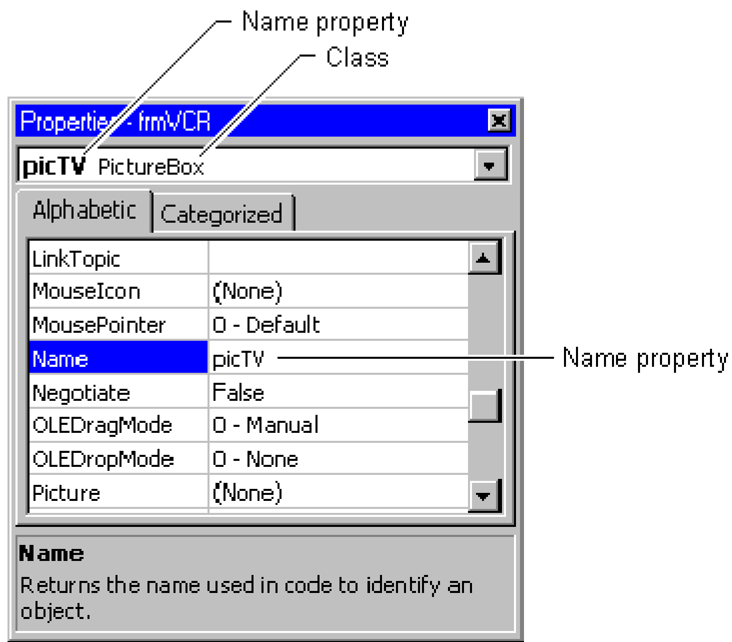
Each object in Visual Basic is defined by a *class*. To understand the relationship between an object and its class, think of cookie cutters and cookies. The cookie cutter is the class. It defines the characteristics of each cookie — for instance, size and shape. The class is used to create objects. The objects are the cookies.

Two examples of the relationship between classes and objects in Visual Basic may make this clearer.

- The controls on the Toolbox in Visual Basic represent classes. The object known as a control doesn't exist until you draw it on a form. When you create a control, you're creating a copy or *instance* of the control class. That instance of the class is the object you refer to in your application.
- The form you work with at design time is a class. At run time, Visual Basic creates an instance of the form's class.

The Properties window displays the class and Name property of objects in your Visual Basic application, as shown in Figure 5.8.

Figure 5.8 Object and class names shown in the Properties window



All objects are created as identical copies of their class. Once they exist as individual objects, their properties can be changed. For example, if you draw three command buttons on a form, each command button object is an instance of the CommandButton class. Each object shares a common set of characteristics and capabilities (properties, methods, and events), defined by the class. However, each has its own name, can be separately enabled and disabled, can be placed in a different location on the form, and so on.

For simplicity, most of the material outside of this chapter won't make many references to an object's class. Just remember that the term "list box control," for example, means "an instance of the ListBox class."

Send feedback to MSDN.   Look here for MSDN Online resources.

What Can You Do with Objects?

An object provides code you don't have to write. For example, you could create your own File Open and File Save dialog boxes, but you don't have to. Instead, you can use the common dialog control (an object) provided by Visual Basic. You could write your own scheduling and resource management code, but you don't have to. Instead, you can use the Calendar, Resources, and Task objects provided by Microsoft Project.

Visual Basic Can Combine Objects from Other Sources

Visual Basic provides the tools to allow you to combine objects from different sources. You can now build custom solutions combining the most powerful features of Visual Basic and applications that support Automation (formerly known as OLE Automation). *Automation* is a feature of the *Component Object Model* (COM), an industry standard used by applications to

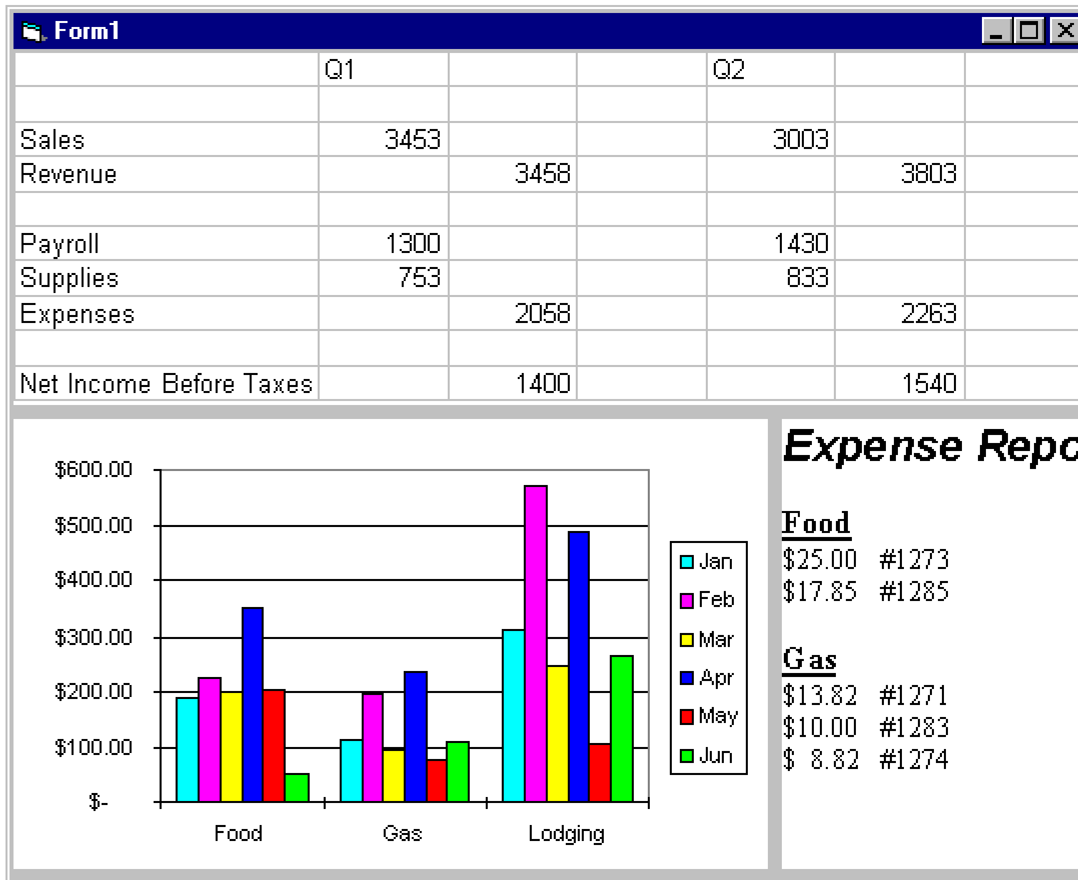
expose objects to development tools and other applications.

You can build applications by tying together intrinsic Visual Basic controls, and you can also use objects provided by other applications. Consider placing these objects on a Visual Basic form:

- A Microsoft Excel Chart object
- A Microsoft Excel Worksheet object
- A Microsoft Word Document object

You could use these objects to create a checkbook application like the one shown in Figure 5.9. This saves you time because you don't have to write the code to reproduce the functionality provided by the Microsoft Excel and Word objects.

Figure 5.9 Using objects from other applications



Send feedback to MSDN. *Look here for MSDN Online resources.*

Visual Basic Concepts

The Basics of Working with Objects

Visual Basic objects support properties, methods, and events. In Visual Basic, an object's data (settings or attributes) are called properties, while the various procedures that can operate on the object are called its methods. An event is an action recognized by an object, such as clicking a mouse or pressing a key, and you can write code to respond to that event.

You can change an object's characteristics by changing its properties. Consider a radio: One property of a radio is its volume. In Visual Basic, you might say that a radio has a "Volume" property that you can adjust by changing its value. Assume you can set the volume of the radio from 0 to 10. If you could control a radio with Visual Basic, you might write code in a procedure that changes the value of the "Volume" property from 3 to 5 to make the radio play louder:

```
Radio.Volume = 5
```

In addition to properties, objects have methods. Methods are a part of objects just as properties are. Generally, methods are actions you want to perform, while properties are the attributes you set or retrieve. For example, you dial a telephone to make a call. You might say that telephones have a "Dial" method, and you could use this syntax to dial the seven-digit number 5551111:

```
Phone.Dial 5551111
```

Objects also have events. Events are triggered when some aspect of the object is changed. For example, a radio might have a "VolumeChange" event. A telephone might have a "Ring" event.

Controlling Objects with Their Properties

Individual properties vary as to when you can set or get their values. Some properties can be set at design time. You can use the Properties window to set the value of these properties without writing any code at all. Some properties are not available at design time; therefore, you must write code to set those properties at run time.

Properties that you can set and get at run time are called *read-write properties*. Properties you can only read at run time are called *read-only properties*.

Setting Property Values

You set the value of a property when you want to change the appearance or behavior of an object. For example, you change the Text property of a text box control to change the contents of the text box.

To set the value of a property, use the following syntax:

object.property = expression

The following statements demonstrate how you set properties:

```
Text1.Top = 200   ' Sets the Top property to 200 twips.
Text1.Visible = True            ' Displays the text box.
Text1.Text = "hello"   ' Displays 'hello' in the text
                     ' box.
```

Getting Property Values

You get the value of a property when you want to find the state of an object before your code performs additional actions (such as assigning the value to another object). For example, you can return the Text property of a text box control to determine the contents of the text box before running code that might change the value.

In most cases, to get the value of a property, you use the following syntax:

variable = object.property

You can also get a property value as part of a more complex expression, without assigning the property to a variable. In the following code example, the Top property of the new member of a control array is calculated as the Top property of the previous member, plus 400:

```
Private Sub cmdAdd_Click()
   ' [statements]
   optButton(n).Top = optButton(n-1).Top + 400
   ' [statements]
End Sub
```

> **Tip** If you're going to use the value of a property more than once, your code will run faster if you store the value in a variable.

Performing Actions with Methods

Methods can affect the values of properties. For example, in the radio analogy, the SetVolume method changes the Volume property. Similarly, in Visual Basic, list boxes have a List property, which can be changed with the Clear and AddItem methods.

Using Methods in Code

When you use a method in code, how you write the statement depends on how many arguments the method requires, and whether the method returns a value. When a method doesn't take arguments, you write the code using the following syntax:

object.method

In this example, the Refresh method repaints the picture box:

```
Picture1.Refresh   ' Forces a repaint of the control.
```

Some methods, such as the Refresh method, don't have arguments and don't return values.

If the method takes more than one argument, you separate the arguments with a comma. For example, the Circle method uses arguments specifying the location, radius, and color of a circle on a form:

```
' Draw a blue circle with a 1200-twip radius.
Form1.Circle (1600, 1800), 1200, vbBlue
```

If you keep the return value of a method, you must enclose the arguments in parentheses. For example, the GetData method returns a picture from the Clipboard:

```
Picture = Clipboard.GetData (vbCFBitmap)
```

If there is no return value, the arguments appear without parentheses. For example, the AddItem method doesn't return a value:

```
List1.AddItem "yourname"      ' Adds the text 'yourname'
                             ' to a list box.
```

For More Information See the *Language Reference* for the syntax and arguments for all methods provided by Visual Basic.

Send feedback to MSDN. Look here for MSDN Online resources.

How are Objects Related to Each Other?

When you put two command buttons on a form, they are separate objects with distinct Name property settings (*Command1* and *Command2*), but they share the same class — CommandButton.

They also share the characteristic that they're on the same form. You've seen earlier in this chapter that a control on a form is also contained by the form. This puts controls in a hierarchy. To reference a control you may have to reference the form first, in the same way you may have to dial a country code or area code before you can reach a particular phone number.

The two command buttons also share the characteristic that they're controls. All controls have common characteristics that make them different from forms and other objects in the Visual Basic environment. The following sections explain how Visual Basic uses collections to group objects that are related.

Object Hierarchies

An object hierarchy provides the organization that determines how objects are related to each other, and how you can access them. In most cases, you don't need to concern yourself with the Visual Basic object hierarchy. However:

- When manipulating another application's objects, you should be familiar with that application's object hierarchy. For information on navigating object hierarchies, see "Programming with Components."
- When working with data access objects, you should be familiar with the Data Access Object hierarchy.

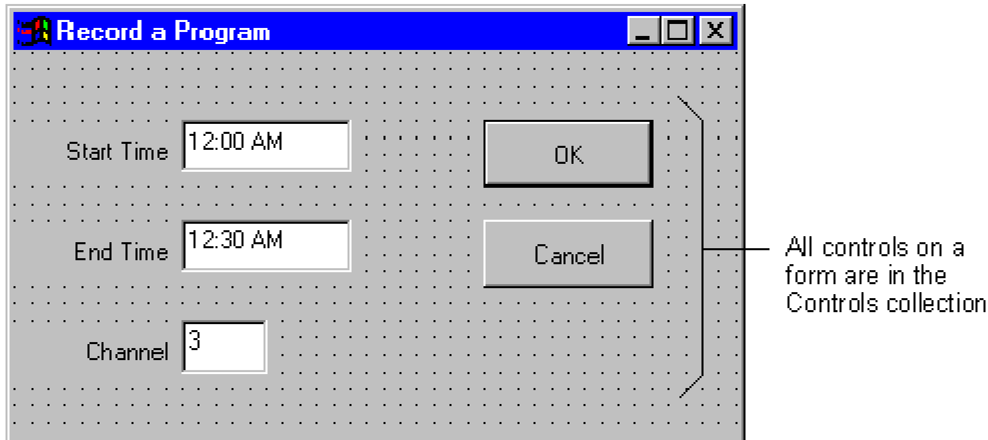
There are some common cases in Visual Basic where one object contains others. These are described in the following sections.

Working with Collections of Objects

Collection objects have their own properties and methods. The objects in a collection object are referred to as *members* of the collection. Each member of the collection is numbered

sequentially beginning at 0; this is the member's *index number*. For example, the Controls collection contains all the controls on a given form, as shown in Figure 5.10. You can use collections to simplify code if you need to perform the same operation on all the objects in a collection.

Figure 5.10 Controls collection



For example, the following code scrolls through the Controls collection and lists each member's name in a list box.

```
Dim MyControl as Control
For Each MyControl In Form1.Controls
   ' For each control, add its name to a list box.
   List1.AddItem MyControl.Name
Next MyControl
```

Applying Properties and Methods to Collection Members

There are two general techniques you can use to address a member of a collection object:

- Specify the name of the member. The following expressions are equivalent:

  ```
  Controls("List1")
  Controls!List1
  ```

- Use the index number of the member:

  ```
  Controls(3)
  ```

Once you're able to address all the members collectively, and single members individually, you can apply properties and methods using either approach:

```
' Set the Top property of the list box control to 200.
Controls!List1.Top = 200
```

–or–

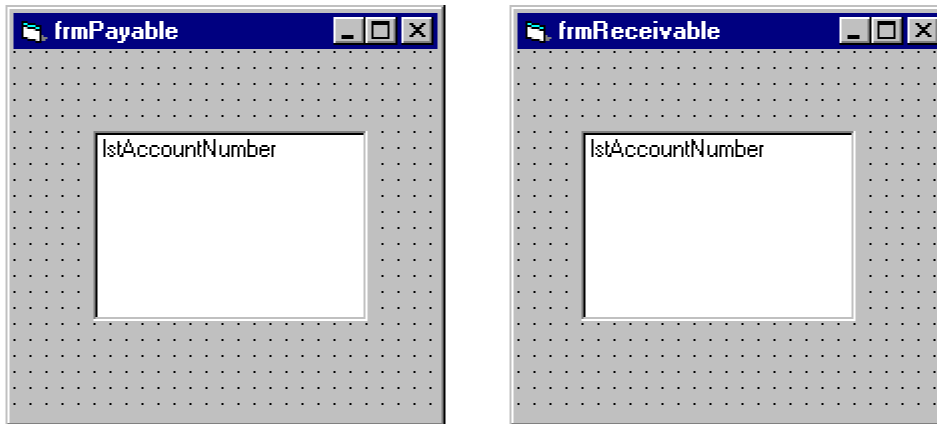
```
Dim MyControl as Control
For Each MyControl In Form1.Controls()
   ' Set the Top property of each member to 200.
   MyControl.Top = 200
```

```
Next MyControl
```

Objects That Contain Other Objects

Some objects in Visual Basic contain other objects. For example, a form usually contains one or more controls. The advantage of having objects as containers for other objects is that you can refer to the container in your code to make it clear which object you want to use. For example, Figure 5.11 illustrates two different forms you could have in an application — one for entering accounts payable transactions, and the other for entering accounts receivable transactions.

Figure 5.11 Two different forms can contain controls that have the same name



Both forms can have a list box named lstAcctNo. You can specify exactly which one you want to use by referring to the form containing the list box:

```
frmReceivable.lstAcctNo.AddItem 1201
```

-or-

```
frmPayable.lstAcctNo.AddItem 1201
```

Common Collections in Visual Basic

There are some common cases in Visual Basic where one object contains other objects. The following table briefly describes the most commonly used collections in Visual Basic.

Collection	Description
Forms	Contains loaded forms.
Controls	Contains controls on a form.
Printers	Contains the available Printer objects.

You can also implement object containment in Visual Basic.

For More Information "For information about object containment, see "Using Collections" in "More About Programming." For information on the Printers collection, see "Working with Text

and Graphics." For details on the forms and controls collections, see the *Language Reference*.

The Container Property

You can use the Container property to change an object's container within a form. The following controls can contain other controls:

- Frame control
- Picture box control
- Toolbar control (Professional and Enterprise editions only)

This example demonstrates moving a command button around from container to container on a form. Open a new project, and draw a frame control, picture box control and a command button on the form.

The following code in the form's click event increments a counter variable, and uses a Select Case loop to rotate the command button from container to container.

```
Private Sub Form_Click()
   Static intX as Integer
   Select Case intX
      Case 0
      Set Command1.Container = Picture1
      Command1.Top= 0
      Command1.Left= 0

      Case 1
      Set Command1.Container = Frame1
      Command1.Top= 0
      Command1.Left= 0

      Case 2
      Set Command1.Container = Form1
      Command1.Top= 0
      Command1.Left= 0

   End Select
   intX = intX + 1
End Sub
```

For More Information See "Container Property" in the *Language Reference*.

Communicating Between Objects

In addition to using and creating objects within Visual Basic, you can communicate with other applications and manipulate their objects from within your application. The ability to share data between applications is one of the key features of the Microsoft Windows operating system. With Visual Basic, you have great flexibility in how you can communicate with other applications.

For More Information For details on using and communicating with other applications' objects, see "Programming with Components."

Creating Objects

The easiest way to create an object is to double-click a control in the Toolbox. However, to realize the full benefit of all the objects available in Visual Basic and from other applications, you can use Visual Basic's programmability features to create objects at run time.

- You can create references to an object with object variables.
- You can create your own objects "from scratch" with class modules.
- You can create your own collections with the Collection object.

For More Information   Other chapters show you how to access objects. The CreateObject and GetObject functions, for example, are discussed in "Programming with Components."

Using Object Variables

In addition to storing values, a variable can refer to an object. You assign an object to a variable for the same reasons you assign any value to a variable:

- Variable names are often shorter and easier to remember than the values they contain (or, in this case, the objects they refer to).
- Variables can be changed to refer to other objects while your code is running.
- Referring to a variable that contains an object is more efficient than repeatedly referring to the object itself.

Using an object variable is similar to using a conventional variable, but with one additional step — assigning an object to the variable:

- First you declare it:

  Dim *variable* As *class*
- Then you assign an object to it:

  Set *variable* = *object*

Declaring Object Variables

You declare an object variable in the same way you declare other variables, with Dim, ReDim, Static, Private, or Public. The only differences are the optional New keyword and the *class* argument; both of these are discussed later in this chapter. The syntax is:

{**Dim** | **ReDim** | **Static** | **Private** | **Public**} *variable* **As** [**New**] *class*

For example, you can declare an object variable that refers to a form in the application called frmMain:

```
Dim FormVar As New frmMain    ' Declare an object
                           ' variable of type frmMain.
```

You can also declare an object variable that can refer to any form in the application:

```
Dim anyForm As Form         ' Generic form variable.
```

Similarly, you can declare an object variable that can refer to any text box in your application:

```
Dim anyText As TextBox      ' Can refer to any text box
                           ' (but only a text box).
```

You can also declare an object variable that can refer to a control of any type:

```
Dim anyControl As Control   ' Generic control variable.
```

Notice that you can declare a form variable that refers to a specific form in the application, but you cannot declare a control variable that refers to a particular control. You can declare a control variable that can refer to a specific type of control (such as TextBox or ListBox), but not to one particular control of that type (such as `txtEntry` or `List1`). However, you can assign a particular control to a variable of that type. For example, for a form with a list box called lstSample, you could write:

```
Dim objDemo As ListBox
Set objDemo = lstSample
```

Assigning Object Variables

You assign an object to an object variable with the Set statement:

`Set` *`variable`* **`=`** *`object`*

Use the Set statement whenever you want an object variable to refer to an object.

Sometimes you may use object variables, and particularly control variables, simply to shorten the code you have to type. For example, you might write code like this:

```
If frmAccountDisplay!txtAccountBalance.Text < 0 Then
   frmAccountDisplay!txtAccountBalance.BackColor = 0   frmAccountDisplay!txtAccountBalance
End If
```

You can shorten this code significantly if you use a control variable:

```
Dim Bal As TextBox
Set Bal = frmAccountDisplay!txtAccountBalance
If Bal.Text < 0 Then
   Bal.BackColor = 0
   Bal.ForeColor = 255
End If
```

Specific and Generic Object Types

Specific object variables must refer to one specific type of object or class. A specific form variable can refer to only one form in the application (though it can refer to one of many instances of that form). Similarly, a specific control variable can refer to only one particular type of control in your application, such as TextBox or ListBox. To see an example, open a new project and place a text box on a form. Add the following code to the form:

```
Private Sub Form_Click()
   Dim anyText As TextBox
   Set anyText = Text1
   anyText.Text = "Hello"
End Sub
```

Run the application, and click the form. The Text property of the text box will be changed to "Hello."

Generic object variables can refer to one of many specific types of objects. A generic form variable, for example, can refer to any form in an application; a generic control variable can refer to any control on any form in an application. To see an example, open a new project and place several frame, label, and command button controls on a form, in any order. Add the following code to the form:

```
Private Sub Form_Click()
   Dim anyControl As Control
   Set anyControl = Form1.Controls(3)
   anyControl.Caption = "Hello"
End Sub
```

Run the application, and click the form. The caption of the control you placed third in sequence on the form will be changed to "Hello."

There are four generic object types in Visual Basic:

Generic Object Type	Object referenced
Form	Any form in the application (including MDI children and the MDI form).
Control	Any control in your application.
MDIForm	The MDI form in the application (if your application has one).
Object	Any object.

Generic object variables are useful when you don't know the specific type of object a variable will refer to at run time. For example, if you want to write code that can operate on any form in the application, you must use a generic form variable.

> **Note**   Because there can be only one MDI form in the application, there is no need to use the generic MDIForm type. Instead, you can use the specific MDIForm type (`MDIForm1`, or whatever you specified for the Name property of the MDI form) whenever you need to declare a form variable that refers to the MDI form. In fact, because Visual Basic can resolve references to properties and methods of specific form types before you run your application, you should always use the specific MDIForm type.

The generic MDIForm type is provided only for completeness; should a future version of Visual Basic allow multiple MDI forms in a single application, it might become useful.

Forms as Objects

Forms are most often used to make up the interface of an application, but they're also objects that can be called by other modules in your application. Forms are closely related to class modules. The major difference between the two is that forms can be visible objects, whereas class modules have no visible interface.

Adding Custom Methods and Properties

You can add custom methods and properties to forms and access them from other modules in your application. To create a new method for a form, add a procedure declared using Public.

```
' Custom method on Form1
Public Sub LateJobsCount()
   .
. ' <statements>
   .
End Sub
```

You can call the LateJobsCount procedure from another module using this statement:

```
Form1.LateJobsCount
```

Creating a new property for a form can be as simple as declaring a public variable in the form module:

```
Public IDNumber As Integer
```

You can set and return the value of IDNumber on Form1 from another module using these two statements:

```
Form1.IDNumber = 3
Text1.Text = Form1.IDNumber
```

You can also use Property procedures to add custom properties to a form.

For More Information Details on Property procedures are provided in "Programming with Objects."

> **Note** You can call a variable, a custom method, or set a custom property on a form without loading the form. This allows you to run code on a form without loading it into memory. Also, referencing a control without referencing one of its properties or methods does not load the form.

Using the New Keyword

Use the New keyword to create a new object as defined by its class. New can be used to create instances of forms, classes defined in class modules, and collections.

Using the New Keyword with Forms

Each form you create at design time is a class. The New keyword can be used to create new instances of that class. To see how this works, draw a command button and several other controls on a form. Set the form's Name property to Sample in the Properties window. Add the following code to your command button's Click event procedure:

```
Dim x As New Sample
x.Show
```

Run the application, and click the command button several times. Move the front-most form aside. Because a form is a class with a visible interface, you can see the additional copies. Each form has the same controls, in the same positions as on the form at design time.

> **Note** To make a form variable and an instance of the loaded form persist, use a Static or Public variable instead of a local variable.

You can also use New with the Set statement. Try the following code in a command button's Click event procedure:

```
Dim f As Form1
Set f = New Form1
f.Caption = "hello"
f.Show
```

Using New with the Set statement is faster and is the recommended method.

Using the New Keyword with Other Objects

The New keyword can be used to create collections and objects from the classes you define in class modules. To see how this works, try the following example.

This example demonstrates how the New keyword creates instances of a class. Open a new project, and draw a command button on Form1. From the Project menu, choose Add Class Module to add a class module to the project. Set the class module's Name property to ShowMe.

The following code in the Form1 module creates a new instance of the class ShowMe, and calls the procedure contained in the class module.

```
Public clsNew As ShowMe
Private Sub Command1_Click()
   Set clsNew = New ShowMe
   clsNew.ShowFrm
End Sub
```

The ShowFrm procedure in the class module creates a new instance of the class Form1, shows the form, and then minimizes it.

```
Sub ShowFrm()
   Dim frmNew As Form1
   Set frmNew = New Form1
   frmNew.Show
   frmNew.WindowState = 1
End Sub
```

To use the example, run the application, and click the command button several times. You'll see a minimized form icon appear on your desktop as each new instance of the ShowMe class is created.

For More Information For information on using New to create objects, see "Programming with Components."

New Keyword Restrictions

The following table describes what you cannot do with the New keyword.

You can't use New to create	Example of code *not* allowed
Variables of fundamental data types.	`Dim X As New Integer`
A variable of any generic object type.	`Dim X As New Control`
A variable of any specific control type.	`Dim X As New ListBox`
A variable of any specific control.	`Dim X As New lstNames`

Freeing References to Objects

Each object uses memory and system resources. It is good programming practice to release these resources when you are no longer using an object.

- Use Unload to unload a form or control from memory.
- Use Nothing to release resources used by an object variable. Assign Nothing to an object variable with the Set statement.

For More Information See "Unload Event" and "Nothing" in the *Language Reference*.

Passing Objects to Procedures

You can pass objects to procedures in Visual Basic. In the following code example, it's assumed that there is a command button on a form:

```
Private Sub Command1_Click()
   ' Calls the Demo sub, and passes the form to it.
   Demo Form1
End Sub

Private Sub Demo(x As Form1)
   ' Centers the form on the screen.
   x.Left = (Screen.Width - x.Width) / 2
End Sub
```

It's also possible to pass an object to an argument by reference and then, inside the procedure, set the argument to a new object. To see how this works, open a project, and insert a second form. Place a picture box control on each form. The following table shows the property settings that need changes:

Object	Property	Setting
Picture box on Form2	Name	Picture2

Picture	c:\vb\icons\arrows\arw01dn.ico

The Form1_Click event procedure calls the GetPicture procedure in Form2, and passes the empty picture box to it.

```
Private Sub Form_Click()
Form2.GetPicture Picture1
End Sub
```

The GetPicture procedure in Form2 assigns the Picture property of the picture box on Form2 to the empty picture box on Form1.

```
Private objX As PictureBox
Public Sub GetPicture(x As PictureBox)
   ' Assign the passed-in picture box to an object
   ' variable.
   Set objX = x
   ' Assign the value of the Picture property to Form1
   ' picture box.
   objX.Picture = picture2.Picture
End Sub
```

To use the example, run the application, and click Form1. You'll see the icon from Form2 appear in the picture box on Form1.

For More Information The previous topics are intended to serve as an introduction to objects. To learn more, see "Programming with Objects" and "Programming with Components."

Send feedback to MSDN. Look here for MSDN Online resources.