Chapter 2

Handling Data

This chapter and the next discuss the fundamentals of any programming language: variables and data types. A variable stores data, which is processed with statements. A program is a list of statements that manipulate variables. To write even simple applications, you need a basic understanding of some fundamental topics, such as the data types (the kind of data you can store in a variable), the scope and lifetime of variables, and how to write procedures and pass arguments to them. In this chapter, we'll explore the basic data types of Visual Basic, and in the following one, you'll learn about procedures and flow-control statements.

If you're new to Visual Basic, you may find some material in this chapter less than exciting. It covers basic concepts and definitions — in general, tedious, but necessary, material. Think of this chapter as a prerequisite for the following ones. If you need information on core features of the language as you go through the examples in the rest of the book, you'll probably find it here.

In this chapter, you'll learn how to do the following:

- Declare and use variables
- Use the native data types
- Create custom data types
- Use arrays

Variables

In Visual Basic, as in any other programming language, variables store values during a program's execution. A variable has a name and a value. The variable *UserName*, for example, might have the value `Joe`, and the variable *Discount* might have the value `0.35`. *UserName* and *Discount* are variable names, and `Joe` and `0.35` are their values. `Joe` is a string (that is, text), and `0.35` is a numeric value. When a variable's value is a string, it must be enclosed in double quotes. In your code, you can refer to the value of a variable by the variable's name.

In addition to a name and a value, variables have a data type, which determines what kind of values you can store to a variable. VB 2010 supports several data types (and they're discussed in detail later in this chapter). It's actually the Common Language Runtime (CLR) that supports the data types, and the data types are common to all languages, not just to Visual Basic. The data type of a variable is specified when the variable is declared, and you should

always declare variables before using them. (I'll tell you more about declaring variables in the next section.)

The various data types are discussed in detail later in this chapter, but let me start with some simple examples to demonstrate the concepts of using variables in an application. One of the available numeric data types is the Decimal data type; it can store both integer and non-integer values. For example, the following statements calculate and display the discount for the amount of $24,500:

```
Dim Amount As Decimal
Dim Discount As Decimal
Dim DiscountedAmount As Decimal
Amount = 24500
Discount = 0.35
DiscountedAmount = Amount * (1 - Discount)
MsgBox("Your price is $" & DiscountedAmount.ToString)
```

If you enter these statements in a button's `Click` event handler to test them, the compiler may underline the statement that assigns the value `0.35` to the *`Discount`* variable and generate an error message. To view the error message, hover the pointer over the underlined segment of the statement in error. This will happen if the Strict option is on. (I discuss the Strict option, along with two more options of the compiler, later in this chapter.) By default, the Strict option is off and the statement won't generate an error.

The compiler treats any numeric value with a fractional part as a Double value and detects that you're attempting to assign a Double value to a Decimal variable. To specify that a numeric value should be treated as a Decimal type, use the following notation:

```
Discount = 0.35D
```

As you will see later, the *D* character at the end of a numeric value indicates that the value should be treated as a Decimal value, and there are a few more type characters (see Table 2.2 later in this chapter). I've used the Decimal data type here because it's commonly used in financial calculations.

The amount displayed on the message box by the last line of code depends on the values of the *`Discount`* and *`Amount`* variables. If you decide to offer a better discount, all you have to do is change the value of the *`Discount`* variable. If you didn't use the *`Discount`* variable, you'd have to make many changes throughout your code. In other words, if you coded the line that calculated the discounted amount as follows, you'd have to look for every line in your code that calculates discounts and change the discount from `0.35` to another value:

```
DiscountedAmount = 24500 * (1 - 0.35)
```

When you change the value of the *`Discount`* variable in a single place in your code, the entire program is up-to-date and it will evaluate the proper discount on any amount.

Declaring Variables

In most programming languages, variables must be declared in advance. Historically, the reason for doing this has been to help the compiler generate the most efficient code. If the compiler

knows all the variables and their types ahead of time, it can produce the most compact and efficient, or optimized, code. For example, when you tell the compiler that the variable *Discount* will hold a number, the compiler sets aside a certain number of bytes for the *Discount* variable to use.

When programming in VB 2010, you should declare your variables because this is the default mode, and Microsoft recommends this practice strongly. If you attempt to use an undeclared variable in your code, VB 2010 will throw an exception. It will actually catch the error as soon as you type in the line that uses the undeclared variable, underlining it with a wiggly line. It is possible to change the default behavior and use undeclared variables the way most people did with earlier versions of VB, but all the examples in this book use explicitly declared variables. In any case, you're strongly encouraged to declare your variables.

You already know how to declare variables with the `Dim` statement and the `As` keyword, which introduces their type:

```
Dim meters As Integer
Dim greetings As String
```

The first variable, *meters*, will store integers, such as 3 or 1,002; the second variable, *greetings*, will store text. You can declare multiple variables of the same or different type in the same line, as follows:

```
Dim Qty As Integer, Amount As Decimal, CardNum As String
```

If you want to declare multiple variables of the same type, you need not repeat the type. Just separate all the variables of the same type with commas and set the type of the last variable:

```
Dim Length, Width, Height As Integer, Volume, Area As Double
```

This statement declares three Integer variables and two Double variables. Double variables hold fractional values (or floating-point values, as they're usually called) that are similar to the Single data type except that they can represent non-integer values with greater accuracy.

An important aspect of variables is their *scope*, a topic that's discussed in more detail later in this chapter. In the meantime, bear in mind that all variables declared with the `Dim` statement exist in the module in which they were declared. If the variable *Count* is declared in a subroutine (an event handler, for example), it exists only in that subroutine. You can't access it from outside the subroutine. Actually, you can have a *Count* variable in multiple procedures. Each variable is stored locally, and they don't interfere with one another.

Variable-Naming Conventions

When declaring variables, you should be aware of a few naming conventions:

- A variable's name must begin with a letter or an underscore character, followed by more letters or digits.
- It can't contain embedded periods or other special punctuation symbols. The only special character that can appear in a variable's name is the underscore character.

- It mustn't exceed 1,024 characters.
- It must be unique within its scope. This means that you can't have two identically named variables in the same subroutine, but you can have a variable named *counter* in many different subroutines.

Variable names are not case sensitive: *myAge*, *myage*, and *MYAGE* all refer to the same variable in your code. Actually, as you enter variable names in your code, the editor converts their casing so that they match their declaration.

Variable Initialization

Visual Basic allows you to initialize variables in the same line that declares them. The following statement declares an Integer variable and immediately places the value 3,045 in it:

```
Dim distance As Integer = 3045
```

This statement is equivalent to the following two:

```
Dim distance As Integer
distance = 3045
```

It is also possible to declare and initialize multiple variables (of the same or different type) on the same line:

```
Dim quantity As Integer = 1, discount As Single = 0.25
```

Types of Variables

You've learned how to declare variables and that all variables should have a type. But what data types are available? Visual Basic recognizes the following five categories of variables:

- Numeric
- String
- Boolean
- Date
- Object

The two major variable categories are numeric and string. Numeric variables store numbers, and string variables store text. Object variables can store any type of data. Why bother to specify the type if one type suits all? On the surface, using object variables might seem like a good idea, but they have their disadvantages. Integer variables are optimized for storing integers, and date variables are optimized for storing dates. Before VB can use an object variable, it must determine its type and perform the necessary conversions. If the variable is declared with a specific type, these conversions are not necessary.

Text is stored in string variables, but numbers can be stored in many formats, depending on the size of the number and its precision. That's why there are many types of numeric variables. The String and Date data types are much richer in terms of the functionality they expose and are discussed in more detail in Chapter 11, ''The Framework at Large.''

Numeric Variables

You'd expect that programming languages would use the same data type for numbers. After all, a number is a number. But this couldn't be further from the truth. All programming languages provide a variety of numeric data types, including the following:

- Integer (there are several Integer data types)
- Decimal
- Single (floating-point numbers with limited precision)
- Double (floating-point numbers with extreme precision)

Decimal, Single, and Double are the three basic data types for storing floating-point numbers (numbers with a fractional part). The Double data type can represent these numbers more accurately than the Single type and is used almost exclusively in scientific calculations. The Integer data types store whole numbers. The data type of your variable can make a difference in the results of the calculations. The proper variable types are determined by the nature of the values they represent, and the choice of data type is frequently a trade-off between precision and speed of execution (less-precise data types are manipulated faster). Visual Basic supports the numeric data types shown in Table 2.1. In the Data Type column, I show the name of each data type and the corresponding keyword in parentheses.

Integer Variables

There are three types of variables for storing integers, and they differ only in the range of numbers each can represent. As you understand, the more bytes a type takes, the larger values it can hold. The type of Integer variable you'll use depends on the task at hand. You should choose the type that can represent the largest values you anticipate will come up in your calculations. You can go for the Long type, to be safe, but Long variables take up four times as much space as Short variables and it takes the computer longer to process them.

Single- and Double-Precision Numbers

The Single and Double data type names come from single-precision and double-precision numbers. Double-precision numbers are stored internally with greater accuracy than single-precision numbers. In scientific calculations, you need all the precision you can get; in those cases, you should use the Double data type.

The Single and Double data types are approximate; you can't represent any numeric value accurately and precisely with these two data types. The problem stems from the fact that computers must store values in a fixed number of bytes, so some accuracy will be lost. Instead of discussing how computers store numeric values, I will demonstrate the side effects of using the wrong data type with a few examples.

The result of the operation $1 \div 3$ is 0.333333 . . . (an infinite number of the digit 3). You could fill 256 MB of RAM with *3s*, and the result would still be truncated. Here's a simple example that demonstrates the effects of truncation.

In a button's `Click` event handler, declare two variables as follows:

```
Dim a As Single, b As Double
```

TABLE 2.1: Visual Basic numeric data types

DATA TYPE	MEMORY REPRESENTATION	STORES
Byte (`Byte`)	1 byte	Integers in the range 0 to 255.
Signed Byte (`SByte`)	1 byte	Integers in the range −128 to 127.
Short (`Int16`)	2 bytes	Integer values in the range −32,768 to 32,767.
Integer (`Int32`)	4 bytes	Integer values in the range −2,147,483,648 to 2,147,483,647.
Long (`Int64`)	8 bytes	Integer values in the range −9,223,372,036,854,755,808 to 9,223,372,036,854,755,807.
Unsigned Short (`UShort`)	2 bytes	Positive integer values in the range 0 to 65,535.
Unsigned Integer (`UInteger`)	4 bytes	Positive integers in the range 0 to 4,294,967,295.
Unsigned Long (`ULong`)	8 bytes	Positive integers in the range 0 to 18,446,744,073,709,551,615.
Single Precision (`Single`)	4 bytes	Single-precision floating-point numbers. A single precision variable can represent negative numbers in the range −3.402823E38 to −1.401298E−45 and positive numbers in the range 1.401298E−45 to 3.402823E38. The value 0 can't be represented precisely (it's a very, very small number, but not exactly 0).
Double Precision (`Double`)	8 bytes	Double-precision floating-point numbers. A double precision variable can represent negative numbers in the range −1.79769313486232E308 to −4.94065645841247E−324 and positive numbers in the range 4.94065645841247E−324 to 1.79769313486232E308.
Decimal (`Decimal`)	16 bytes	Integer and floating-point numbers scaled by a factor in the range from 0 to 28. See the description of the Decimal data type for the range of values you can store in it.

Then enter the following statements:

```
a = 1 / 3
Debug.WriteLine(a)
```

Run the application, and you should get the following result in the Output window:

```
.3333333
```

There are seven digits to the right of the decimal point. Break the application by pressing Ctrl+Break and append the following lines to the end of the previous code segment:

```
a = a * 100000
Debug.WriteLine(a)
```

This time, the following value will be printed in the Output window:

```
33333.34
```

The result is not as accurate as you might have expected initially — it isn't even rounded properly. If you divide *a* by 100,000, the result will be as follows:

```
0.3333334
```

This number is different from the number we started with (0.3333333). The initial value was rounded when we multiplied it by 100,000 and stored it in a Single variable. This is an important point in numeric calculations, and it's called *error propagation*. In long sequences of numeric calculations, errors propagate. Even if you can tolerate the error introduced by the Single data type in a single operation, the cumulative errors might be significant.

Let's perform the same operations with double-precision numbers, this time using the variable *b*. Add these lines to the button's `Click` event handler:

```
b = 1 / 3
Debug.WriteLine(b)
b = b * 100000
Debug.WriteLine(b)
```

This time, the following numbers are displayed in the Output window:

```
0.333333333333333
33333.3333333333
```

The results produced by the double-precision variables are more accurate.

Why are such errors introduced in our calculations? The reason is that computers store numbers internally with two digits: zero and one. This is very convenient for computers because electronics understand two states: on and off. As a matter of fact, all the statements are translated into bits (zeros and ones) before the computer can understand and execute them. The binary numbering system used by computers is not much different from the decimal system we humans use; computers just use fewer digits. We humans use 10 different digits to represent any number, whole or fractional, because we have 10 fingers (in effect, computers count with just two fingers). Just as with the decimal numbering system, in which some numbers can't be precisely represented, there are numbers that can't be represented precisely in the binary system.

Let me give you a more illuminating example. Create a single-precision variable, *a*, and a double-precision variable, *b*, and assign the same value to them:

```
Dim a As Single, b As Double
```

```
a = 0.03007
b = 0.03007
```

Then print their difference:

```
Debug.WriteLine(a-b)
```

If you execute these lines, the result won't be zero! It will be –6.03199004634014E–10. This is a very small number that can also be written as 0.000000000603199004634014. Because different numeric types are stored differently in memory, they don't quite match. What this means to you is that all variables in a calculation should be of the same type.

Eventually, computers will understand mathematical notation and will not convert all numeric expressions into values as they do today. If you multiply the expression 1/3 by 3, the result should be 1. Computers, however, must convert the expression 1/3 into a value before they can multiply it by 3. Because 1/3 can't be represented precisely, the result of the (1/3) × 3 will not be exactly 1. If the variables *a* and *b* are declared as Single or Double, the following statements will print 1:

```
a = 3
b = 1 / a
Debug.WriteLine(a * b)
```

If the two variables are declared as Decimal, however, the result will be a number very close to 1 but not exactly 1 (it will be 0.99999999999999999999999999 — there will be 28 digits after the decimal point). Fortunately, these errors do not surface with typical business-line applications, but you should be aware of truncation errors and how they may affect your calculations. In business applications, we always round our results to two decimal digits and the value 0.999999 of the preceding example will be rounded to 1.00.

The Decimal Data Type

Variables of the Decimal type are stored internally as integers in 16 bytes and are scaled by a power of 10. The scaling power determines the number of decimal digits to the right of the floating point, and it's an integer value from 0 to 28. When the scaling power is 0, the value is multiplied by 10^0, or 1, and it's represented without decimal digits. When the scaling power is 28, the value is divided by 10^{28}(which is 1 followed by 28 zeros — an enormous value), and it's represented with 28 decimal digits.

The largest possible value you can represent with a Decimal value is an integer: 79,228,162,514,264,337,593,543,950,335. The smallest number you can represent with a Decimal variable is the negative of the same value. These values use a scaling factor of 0. When the scaling factor is 28, the largest value you can represent with a Decimal variable is quite small, actually. It's 7.9228162514264337593543950335 (and the smallest value is the same with a minus sign). This is a very small numeric value (not quite 8), but it's represented with extreme accuracy. The number zero can't be represented precisely with a Decimal variable scaled by a factor of 28. The smallest positive value you can represent with the same scaling factor is 0.00 . . . 01 (there are 27 zeros between the decimal period and the digit 1) — an extremely small value, but still not quite zero. The more accuracy you want to achieve with a Decimal variable, the smaller the range of available values you have at your disposal — just as with everything else in life.

When using decimal numbers, the compiler keeps track of the decimal digits (the digits following the decimal point) and treats all values as integers. The value 235.85 is represented as the integer 23585, but the compiler knows that it must scale down the value by 100 when it finishes using it. Scaling down by 100 (that is, 10^2) corresponds to shifting the decimal point by two places. First, the compiler multiplies this value by 100 to make it an integer. Then, it divides it by 100 to restore the original value. Let's say that you want to multiply the following values:

```
328.558 * 12.4051
```

First, the compiler turns them into integers. The compiler remembers that the first number has three decimal digits and the second number has four decimal digits. The result of the multiplication will have seven decimal digits. So the compiler can multiply the following integer values:

```
328558 * 124051
```

It then treats the last seven digits of the result as decimals. The result of the multiplication is 40,757,948,458. The actual value after taking into consideration the decimal digits is 4,075.7948458. This is how the compiler manipulates the Decimal data type.

Type Characters

As I mentioned earlier, the *D* character at the end of a numeric value specifies that the number should be converted into a Decimal value. By default, every value with a fractional part is treated as a Double value because this type can accommodate fractional values with the greatest possible accuracy. Assigning a Double value to a Decimal variable will produce an error if the Strict option is on, so you must specify explicitly that the two values should be converted to the Decimal type. The *D* character at the end of the value is called a type character. Table 2.2 lists all of the type characters that are available in Visual Basic.

Table 2.2: Type characters

Type Character	Description	Example
C	Converts value to a Char type	`Dim ch As String = "A"c`
D or @	Converts value to a Decimal type	`Dim price As Decimal = 12.99D`
R or #	Converts value to a Double type	`Dim pi As Double = 3.14R`
I or %	Converts value to an Integer type	`Dim count As Integer = 99I`
L or &	Converts value to a Long type	`Dim distance As Long = 1999L`
S	Converts value to a Short type	`Dim age As Short = 1S`
F or !	Converts value to a Single type	`Dim velocity As Single = 74.99F`

If you perform the same calculations with Single variables, the result will be truncated (and rounded) to three decimal digits: 4,075.795. Notice that the Decimal data type didn't introduce any rounding errors. It's capable of representing the result with the exact number of decimal digits provided the Decimal type can accommodate both operands and their result. This is the real advantage of Decimals, which makes them ideal for financial applications. For scientific calculations, you must still use Doubles. Decimal numbers are the best choice for calculations that require a specific precision (such as four or eight decimal digits).

Infinity and Other Oddities

The Framework can represent two very special values, which may not be numeric values themselves but are produced by numeric calculations: `NaN` (not a number) and `Infinity`. If your calculations produce `NaN` or `Infinity`, you should give users a chance to verify their data, or even recode your routines as necessary. For all practical purposes, neither `NaN` nor `Infinity` can be used in everyday business calculations.

Not a Number (NaN)

`NaN` is not new. Packages such as Wolfram Mathematica and Microsoft Excel have been using it for years. The value `NaN` indicates that the result of an operation can't be defined: It's not a regular number, not zero, and not infinity. `NaN` is more of a mathematical concept rather than a value you can use in your calculations. The `Log()` function, for example, calculates the logarithm of positive values. By definition, you can't calculate the logarithm of a negative value. If the argument you pass to the `Log()` function is a negative value, the function will return the value `NaN` to indicate that the calculations produced an invalid result. You may find it annoying that a numeric function returns a non-numeric value, but it's better than if it throws an exception. Even if you don't detect this condition immediately, your calculations will continue and they will all produce `NaN` values.

Some calculations produce undefined results, such as infinity. Mathematically, the result of dividing any number by zero is infinity. Unfortunately, computers can't represent infinity, so they produce an error when you request a division by zero. Visual Basic will report a special value, which isn't a number: the Infinity value. If you call the `ToString` method of this value, however, it will return the string `Infinity`. Let's generate an `Infinity` value. Start by declaring a Double variable, *`dblVar`*:

```
Dim dblVar As Double = 999
```

Then divide this value by zero:

```
Dim infVar as Double
infVar = dblVar / 0
```

And display the variable's value:

```
MsgBox(infVar)
```

The string `Infinity` will appear in a message box. This string is just a description; it tells you that the result is not a valid number (it's a very large number that exceeds the range of numeric values that can be represented with any data type), but it *shouldn't* be used in other calculations. However, you *can* use the Infinity value in arithmetic operations. Certain operations with infinity make sense; others don't. If you add a number to infinity, the result is still infinity (any number, even an arbitrarily large one, can still be increased). If you divide a value by infinity, you'll get the zero value, which also makes sense. If you divide one `Infinity` value by another Infinity value, you'll get the second odd value, `NaN`.

Another calculation that will yield a non-number is the division of a very large number by a very small number (a value that's practically zero, but not quite). If the result exceeds the largest value that can be represented with the Double data type, the result is `Infinity`. Declare three variables as follows:

```
Dim largeVar As Double = 1E299
Dim smallVar As Double = 1E-299
Dim result As Double
```

The notation `1E299` means 10 raised to the power of 299, which is an extremely large number. Likewise, `1E-299` means 10 raised to the power of –299, which is equivalent to dividing 10 by a number as large as `1E299`.

Then divide the large variable by the small variable and display the result:

```
result = largeVar / smallVar
MsgBox(result)
```

The result will be `Infinity`. If you reverse the operands (that is, you divide the very small by the very large variable), the result will be zero. It's not exactly zero, but the Double data type can't accurately represent numeric values that are very, very close (but not equal) to zero.

You can also produce an `Infinity` value by multiplying a very large (or very small) number by itself many times. But clearly, the most absurd method of generating an `Infinity` value is to assign the `Double.PositiveInfinity` or `Double.NegativeInfinity` value to a variable!

The result of the division `0 / 0`, for example, is not a numeric value. If you attempt to enter the statement `0 / 0` in your code, however, VB will catch it even as you type, and you'll get the error message *Division by zero occurs in evaluating this expression*.

To divide zero by zero, set up two variables as follows:

```
Dim var1, var2 As Double
Dim result As Double
var1 = 0
var2 = 0
result = var1 / var2
MsgBox(result)
```

If you execute these statements, the result will be `NaN`. Any calculations that involve the *`result`* variable will also yield `NaN`. The following statements will produce a `NaN` value:

```
result = result + result
result = 10 / result
```

```
result = result + 1E299
MsgBox(result)
```

If you make *var2* a very small number, such as 1E-299, the result will be zero. If you make *var1* a very small number, the result will be Infinity.

For most practical purposes, `Infinity` is handled just like `NaN`. They're both numbers that shouldn't occur in business applications (unless you're projecting the national deficit in the next 50 years), and when they do, it means that you must double-check your code or your data. They are much more likely to surface in scientific calculations, and they must be handled with the statements described in the next section.

Testing for Infinity and NaN

To find out whether the result of an operation is a `NaN` or `Infinity`, use the `IsNaN` and `IsInfinity` methods of the Single and Double data types. The Integer data type doesn't support these methods, even if it's possible to generate `Infinity` and `NaN` results with integers. If the `IsInfinity` method returns True, you can further examine the sign of the Infinity value with the `IsNegativeInfinity` and `IsPositiveInfinity` methods.

In most situations, you'll display a warning and terminate the calculations. The statements of Listing 2.1 do just that. Place these statements in a button's `Click` event handler and run the application.

LISTING 2.1: Handling NaN and Infinity values

```
Dim var1, var2 As Double
Dim result As Double
var1 = 0
var2 = 0
result = var1 / var2
If Double.IsInfinity(result) Then
    If Double.IsPositiveInfinity(result) Then
        MsgBox("Encountered a very large number. Can't continue")
    Else
        MsgBox("Encountered a very small number. Can't continue")
    End If
Else
    If Double.IsNaN(result) Then
        MsgBox("Unexpected error in calculations")
    Else
        MsgBox("The result is : " & result.ToString)
    End If
End If
```

This listing will generate a `NaN` value. Set the value of the *var1* variable to 1 to generate a positive Infinity value or to –1 to generate a negative Infinity value. As you can see, the `IsInfinity`, `IsPositiveInfinity`, `IsNegativeInfinity`, and `IsNaN` methods require that the variable be passed as an argument.

If you change the values of the *var1* and *var2* variables to the following values and execute the application, you'll get the message *Encountered a very large number*:

```
var1 = 1E+299
var2 = 1E-299
```

If you reverse the values, you'll get the message *Encountered a very small number*. In either case, the program will terminate gracefully and let the user know the type of problem that prevents the completion of the calculations.

Byte Variables

None of the previous numeric types is stored in a single byte. In some situations, however, data are stored as bytes, and you must be able to access individual bytes. The Byte data type holds an integer in the range of 0 to 255. Bytes are frequently used to access binary files, image and sound files, and so on. To declare a variable as a Byte, use the following statement:

```
Dim n As Byte
```

The variable *n* can be used in numeric calculations too, but you must be careful not to assign the result to another Byte variable if its value might exceed the range of the Byte type. If the variables *A* and *B* are initialized as:

```
Dim A As Byte, B As Byte
A = 233
B = 50
```

the following statement will produce an overflow exception:

```
Debug.WriteLine(A + B)
```

The result (283) can't be stored in a single byte. Visual Basic generates the correct answer, but it can't store it into a Byte variable.

Boolean Operations with Bytes

The operators that won't cause overflows are the Boolean operators `And`, `Or`, `Not`, and `Xor`, which are frequently used with Byte variables. These aren't logical operators that return True or False; they combine the matching bits in the two operands and return another byte. If you combine the numbers 199 and 200 with the AND operator, the result is 192. The two values in binary format are 11000111 and 11001000. If you perform a bitwise AND operation on these two values, the result is 11000000, which is the decimal value 192.

In addition to the Byte data type, VB 2010 provides a Signed Byte data type, `SByte`, which can represent signed values in the range from –128 to 127. The bytes starting with the 1 bit represent negative values. The range of positive values is less by one than the range of negative values because the value 0 is considered a positive value (its first bit is 0).

Boolean Variables

The Boolean data type stores True/False values. Boolean variables are, in essence, integers that take the value –1 (for True) and 0 (for False). Actually, any nonzero value is considered True. Boolean variables are declared as

```
Dim failure As Boolean
```

and they are initialized to False. Even so, it's a good practice to initialize your variables explicitly, as in the following code segment. Boolean variables are used in testing conditions, such as the following:

```
Dim failure As Boolean = False
' other statements …
If failure Then MsgBox("Couldn't complete the operation")
```

They are also combined with the logical operators `And`, `Or`, `Not`, and `Xor`. The `Not` operator toggles the value of a Boolean variable. The following statement is a toggle:

```
running = Not running
```

If the variable *`running`* is True, it's reset to False and vice versa. This statement is a shorter way of coding the following:

```
Dim running As Boolean
If running = True Then
   running = False
Else
   running = True
End If
```

Boolean operators operate on Boolean variables and return another Boolean as their result. The following statements will display a message if one (or both) of the variables `ReadOnly` and `Hidden` are True (in the following example, the *`ReadOnly`* and *`Hidden`* variables might represent the corresponding attributes of a file):

```
If ReadOnly Or Hidden Then
   MsgBox("Couldn't open the file")
Else
   ' statements to open and process file…
End If
```

The condition of the `If` statement combines the two Boolean values with the `Or` operator. If one or both of them are True, the final expression is True.

String Variables

The String data type stores only text, and string variables are declared as follows:

```
Dim someText As String
```

You can assign any text to the variable *someText*. You can store nearly 2 GB of text in a string variable (that's 2 billion characters, and it's much more text than you care to read on a computer screen). The following assignments are all valid:

```
Dim aString As String
aString = "Now is the time for all good men to come "
           "to the aid of their country"
aString = ""
aString = "There are approximately 25,000 words in this chapter"
aString = "25,000"
```

The second assignment creates an empty string, and the last one creates a string that just happens to contain numerals, which are also characters. The difference between these two variables is that they hold different values:

```
Dim aNumber As Integer = 25000
Dim aString As String = "25,000"
```

The *aString* variable holds the characters 2, 5, comma, 0, 0, and 0, and *aNumber* holds a single numeric value. However, you can use the variable *aString* in numeric calculations and the variable *aNumber* in string operations. VB will perform the necessary conversions as long as the Strict option is off. In general, you should turn on the Strict option because it will help you catch possible runtime errors, as discussed in the section ''The Strict, Explicit, and Infer Options.'' The recommended practice is to convert strings to numbers and numbers to strings explicitly as needed using the methods discussed in the section ''Converting Variable Types,'' later in this chapter. Even if you prefer to work with the Strict option off, which is the default value, it's recommended that you turn it on temporarily to spot any areas in your code that might cause runtime errors.

Character Variables

Character variables store a single Unicode character in two bytes. In effect, characters are Unsigned Short integers (`UInt16`), but the Framework provides all the tools you need to work with characters without having to resort to their numeric values (a very common practice for the older among us).

To declare a Character variable, use the `Char` data type:

```
Dim char1, char2 As Char
```

You can initialize a Char variable by assigning either a character or a string to it. In the latter case, only the first character of the string is assigned to the variable. The following statements will print the characters *a* and *A* to the Output window:

```
Dim char1 As Char = "a", char2 As Char = "ABC"
Debug.WriteLine(char1)
Debug.WriteLine(char2)
```

These statements will work only if the Strict option is off. If it's on, the values assigned to the *char1* and *char2* variables will be marked in error and the code will not compile. To fix the error, change the `Dim` statement as follows:

```
Dim char1 As Char = ″a″c, char2 As Char = ″A″c
```

(This tells the compiler to treat the values of the variables as characters, not strings.) When the Strict option is on, you can't assign a string to a Char variable and expect that only the first character of the string will be used.

Unicode or ANSI

The Integer values that correspond to the English characters are the ANSI (American National Standards Institute) codes of the equivalent characters. The following statement will print the value 65:

```
Debug.WriteLine(Convert.ToInt32("a"))
```

If you convert the Greek character alpha (α) to an integer, its value is 945. The Unicode value of the famous character π is 960. Unicode and ANSI values for English characters are the same, but all ''foreign'' characters have a unique Unicode value.

Character variables are used in conjunction with strings. You'll rarely save real data as characters. However, you might have to process the individual characters in a string, one at a time. Let's say the string variable `password` holds a user's new password, and you require that passwords contain at least one special symbol. The code segment of Listing 2.2 scans the password and rejects it if it contains letters and digits only.

Listing 2.2: Processing individual characters

```
Dim password As String, ch As Char
Dim i As Integer
Dim valid As Boolean = False
While Not valid
   password = InputBox(″Please enter your password″)
   For i = 0 To password.Length - 1
      ch = password.Chars(i)
      If Not Char.IsLetterOrDigit(ch) Then
         valid = True
         Exit For
      End If
   Next
   If valid Then
      MsgBox(″You new password will be activated immediately! ″)
   Else
      MsgBox(″Your password must contain at least one special symbol! ″)
   End If
End While
```

If you are not familiar with the `If…Then`, `For…Next`, or `While…End While` structures, you can read their descriptions in the following chapter.

The code prompts the user with an input box to enter a password. The *valid* variable is Boolean and it's initialized to False. (You don't have to initialize a Boolean variable to False because this is its default initial value, but it does make the code easier to read.) It's set to True from within the body of the loop only if the password contains a character that is not a letter or a digit. We set it to False initially, so the `While…End While` loop will be executed at least once. This loop will keep prompting the user until a valid password is entered.

The `For…Next` loop scans the string variable *password*, one letter at a time. At each iteration, the next letter is copied into the *ch* variable. The `Chars` property of the String data type is an array that holds the individual characters in the string (another example of the functionality built into the data types).

Then the program examines the current character. The `IsLetterOrDigit` method of the Char data type returns True if a character is either a letter or a digit. If the current character is a symbol, the program sets the *valid* variable to True so that the outer loop won't be executed again, and it exits the `For…Next` loop. Finally, it prints the appropriate message and either prompts for another password or quits.

Date Variables

Date variables store date values that may include a time part (or not), and they are declared with the Date data type:

```
Dim expiration As Date
```

The following are all valid assignments:

```
expiration = #01/01/2010#
expiration = #8/27/1998 6:29:11 PM#
expiration = "July 2, 2011"
expiration = Today()
```

Now and Today

By the way, the `Today()` function returns the current date and time, while the `Now()` function returns the current date. You can also retrieve the current date by calling the `Today` property of the Date data type: `Date.Today`.

The pound sign tells Visual Basic to store a date value to the *expiration* variable, just as the quotes tell Visual Basic that the value is a string. You can store a date as a string to a Date variable, but it will be converted to the appropriate format.

The format of the date inside the pound characters is determined by the regional settings (found in Control Panel). In the United States, the format is *mm/dd/yy*. (In other countries, the format is *dd/mm/yy*.) If you assign an invalid date to a Date variable, such as 23/04/2012, the statement will be underlined and an error message will appear in the Task List window. The description of the error is *Date constant is not valid*.

You can also perform arithmetic operations with date values. VB recognizes your intention to subtract dates and it properly evaluates their difference. The result is a TimeSpan

object, which represents a time interval. If you execute the following statements, the value 638.08:49:51.4970000 will appear in the Output window:

```
Dim d1, d2 As Date
d1 = Now
d2 = #1/1/2004#Debug.WriteLine(d1 - d2)
```

The value of the TimeSpan object represents an interval of 638 days, 8 hours, 49 minutes, and 51.497 seconds.

Converting between Locales

In a global environment like ours, handling dates has gotten a bit complicated. If you live in the United States and you receive a data file that includes dates from a company in the United Kingdom, you should take into consideration the locale of the computer that generated the file. To specify the locale of a date value, use the `Parse` method of the DateTime class, which accepts two arguments: the date to be parsed and a `CultureInfo` object that represents the date's locale. (If you find this tip too advanced on first reading, please make a note and look it up when you have to deal with dates in different cultures).

The date 25/12/2011 is a valid UK date, but if you attempt to assign this value to a Date variable (assuming that your computer's locale is English-US), the statement will generate an error. To convert the date to US format, create a `CultureInfo` that represents the locale of the original date:

```
Dim UK As New CultureInfo("en-GB")
```

Then call the `DateTime.Parse` method, as follows, to convert the date value to a valid date:

```
Dim D1 As Date
D1 = DateTime.Parse("25/12/2011", UK)
```

The following code segment compares two dates with different locales to one another and prints an appropriate message that indicates whether the two dates are equal (in this example, they are):

```
Dim D1, D2 As Date
Dim UK As New CultureInfo("en-GB")
Dim US As New CultureInfo("en-US")
D1 = DateTime.Parse("27/8/2010", UK)
D2 = DateTime.Parse("8/27/2010", US)
If D1 = D2 Then
    MsgBox("Same date")
Else
    MsgBox("Different dates")
End If
```

Dates like 3/4/2025 or 4/3/2025 are valid in any culture, but they may not be correct unless you interpret them with the proper locale, so be careful when importing dates. You can look

up the locales of other countries in the documentation. For example, fr-FR is France's French locale, fr-BE is Belgium's French locale, and fr-CH is Switzerland's French locale. For Switzerland, a culturally diverse place, there's also a German locale, the de-CH locale. The problem of locales is also addressed by XML, which is the definitive standard for data exchange, and it's discussed later in this book in Chapter 13, ''XML in Modern Programming,'' and Chapter 14, ''Introduction to LINQ.''

You'll face a similar issue with formatted numeric values because some locales use the period as the decimal separator while others use it as a separator for thousands. The two formatted values 19,000.99 and 19.000,99 are valid in different cultures, but they're not the same at once. To properly convert these formatted numbers, use the `Parse` method of the Decimal or Double class, passing as argument the string to be parsed and the locale of the original value (the US locale for 19,999.99 and the UK locale for 19,999.99). Again, examine the following statements that convert these two formatted numeric strings into numeric values, taking into consideration the proper locale. The statements are equivalent to the ones I showed you earlier for handling dates. For this example, I'll use the Italian language locale; that locale uses the period as the thousands separator and the coma as the decimal separator.

```
Dim val1, val2 As Decimal
Dim IT As New CultureInfo("it-IT")
Dim US As New CultureInfo("en-US")
val1 = System.Decimal.Parse("19,999.99", IT)
val2 = System.Decimal.Parse("19,999.99", US)
If val1 = val2 Then
    MsgBox("Same values")
Else
    MsgBox("Different values")
End If
```

Many developers try to remove the thousands separator(s) from the formatted number and then replace the period with a coma (or vice versa). Use the technique shown here; it will work regardless of the current locale and it's so much easier to read and so much safer.

The Strict, Explicit, and Infer Options

The Visual Basic compiler provides three options that determine how it handles variables:

- The Explicit option indicates whether you will declare all variables.
- The Strict option indicates whether all variables will be of a specific type.
- The Infer option indicates whether the compiler should determine the type of a variable from its value.

These options have a profound effect on the way you declare and use variables, and you should understand what they do. By exploring these settings, you will also understand a little better how the compiler handles variables. It's recommended that you turn on all three, but old VB developers may not want to follow this advice.

VB 2010 doesn't *require* that you declare your variables, but the default behavior is to throw an exception if you attempt to use a variable that hasn't been previously declared. If an

undeclared variable's name appears in your code, the editor will underline the variable's name with a wiggly line, indicating that it caught an error. The description of the error will appear in the Task List window below the code window. If you rest the cursor over the segment in question, you will see the description of the error in a ToolTip box.

To change the default behavior, you must insert the following statement at the beginning of the file:

```
Option Explicit Off
```

The `Option Explicit` statement must appear at the very beginning of the file. This setting affects the code in the current module, not in all files of your project or solution. You can turn on the Strict (as well as the Explicit) option for an entire solution. Open the project's properties page (right-click the project's name in Solution Explorer and select Properties), select the Compile tab, and set the Strict and Explicit options accordingly, as shown in Figure 2.1.

FIGURE 2.1
Setting the variable-related options on the project's properties pages

You can also set default values for the Explicit option (as well as for Strict and Infer) for all projects through the Options dialog box of the IDE (Integrated Development Environment). To open this dialog box, choose the Options command from the Tools menu. When the dialog box appears, select the VB Defaults tab under Projects And Solutions, as shown in Figure 2.2. Here you can set the default values for all four options. You can still change the default values for specific projects through the project's properties pages.

The way undeclared variables are handled by VB 2010 is determined by the Explicit and Strict options, which can be either on or off. The Explicit option requires that all variables used in the code are declared before they're used. The Strict option requires that variables are declared with a specific type. In other words, the Strict option disallows the use of generic variables that can store any data type.

The default value of the Explicit statement is On. This is also the recommended value, and you should not make a habit of changing this setting. By setting the Explicit option to Off, you're telling VB that you intend to use variables without declaring them. As a consequence, VB can't make any assumption about the variable's type, so it uses a generic type of variable that can hold any type of information. These variables are called Object variables, and they're equivalent to the old variants.

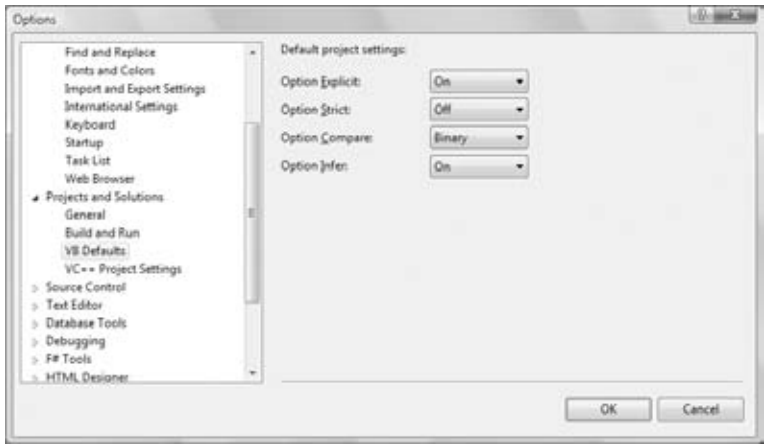

FIGURE 2.2
Setting the variable-related options in the Visual Studio Options dialog box

While the option Explicit is set to Off, every time Visual Basic runs into an undeclared variable name, it creates a new variable on the spot and uses it. Visual Basic adjusts the variable's type according to the value you assign to it. With Explicit turned off, create two variables, *var1* and *var2*, by referencing them in your code with statements like the following ones:

```
var1 = "Thank you for using Fabulous Software"
var2 = 49.99
```

The *var1* variable is a string variable, and *var2* is a numeric one. You can verify this with the `GetType` method, which returns a variable's type. The following statements print the highlighted types shown below each statement:

```
Debug.WriteLine "Variable var1 is " & var1.GetType().ToString
Variable var1 is System.String
Debug.WriteLine "Variable var2 is " & var2.GetType().ToString
Variable var2 is System.Double
```

Later in the same program, you can reverse the assignments:

```
var1 = 49.99
var2 = "Thank you for using Fabulous Software"
```

If you execute the preceding type-checking statements again, you'll see that the types of the variables have changed. The *var1* variable is now a Double, and *var2* is a String. The type of a generic variable is determined by the variable's contents, and it can change in the course of the application. Of course, changing a variable's type at runtime doesn't come without a performance penalty (a small one, but nevertheless some additional statements must be executed).

Another related option is the Strict option, which is off by default. The Strict option tells the compiler whether the variables should be *strictly typed*. A strictly typed (or strongly typed) variable must be declared with a specific type and it can accept values of the same type only.

With the Strict option set to Off, you can use a string variable that holds a number in a numeric calculation:

```
Dim a As String = "25000"
Debug.WriteLine a / 2
```

The last statement will print the value 12500 in the Immediate window. Likewise, you can use numeric variables in string calculations:

```
Dim a As Double = 31.03
a = a + "1"
```

After the execution of the preceding statements, the *a* variable will still be a Double and will have the value 32.03. If you turn the Strict option on by inserting the following statement at the beginning of the file, you won't be able to mix and match variable types:

```
Option Strict On
```

If you attempt to execute any of the last two code segments while the Strict option is on, the editor will underline a segment of the statement to indicate an error. If you rest the cursor over the underlined segment of the code, the following error message will appear in a tip box:

```
Option strict disallows implicit conversions from String to Double
```

or any type conversion is implied by your code.

When the Strict option is set to On, the compiler will allow some implicit conversions between data types, but not always. For example, it will allow you to assign the value of an integer to a Long, but not the opposite. The Long value might exceed the range of values that can be represented by an Integer variable.

Type Inference

One of the trademark features of BASIC, including earlier versions of Visual Basic, was the ability to use variables without declaring them. It has never been a recommended practice, yet VB developers loved it. This feature is coming back to the language, only in a safer manner. VB 2010 allows you to declare variables by assigning values to them. The compiler will infer the type of the variable from its value and will create a variable of the specific type behind the scenes. The following statement creates an Integer variable:

```
Dim count = 2999
```

Behind the scenes, the compiler will create a typed variable with the following statement:

```
Dim count As Integer = 2999
```

To request the variable's type, use the `GetType` method. This method returns a Type object, which represents the variable's type. The name of the type is given by the `ToString` property. The following statement will print the highlighted string in the Immediate window:

```
Debug.WriteLine(count.GetType.ToString)
System.Int32
```

The *count* variable is of the Integer type (the 32-bit integer variety, to be precise). If you attempt to assign a value of a different type, such as a date, to this variable later in your code, the editor will underline the value and generate a warning like this: *Value of type 'Date' cannot be converted to Integer*. The compiler has inferred the type of the value assigned initially to the variable and created a variable of the same type. That's why subsequent statements can't change the variable's type. Behind the scenes, the compiler will actually insert a Dim statement, as if you had declared the variable explicitly.

If the Infer option is off, the compiler will handle variables declared without a specific type depending on the Strict option. If the Strict option is off, the compiler will create an Object variable, which can store any value, even values of different types in the course of the application. If the Strict option is on, the compiler will reject the declaration; it will underline the variable's name with a wiggly line and generate the following warning: *Option Strict On requires all variable declarations to have an As clause*.

Object Variables

Variants — variables without a fixed data type — were the bread and butter of VB programmers up to version 6. Variants are the opposite of strictly typed variables: They can store all types of values, such as integers, strings, characters, you name it. If you're starting with VB 2010, you should use strongly typed variables. However, variants are a major part of the history of VB, and most applications out there (the ones you may be called to maintain) use them. I will discuss variants briefly in this section and show you what was so good (and bad) about them.

Variants, or object variables, are the most flexible data type because they can accommodate all other types. A variable declared as Object (or a variable that hasn't been declared at all) is handled by Visual Basic according to the variable's current contents. If you assign an integer value to an object variable, Visual Basic treats it as an integer. If you assign a string to an object variable, Visual Basic treats it as a string. Variants can also hold different data types in the course of the same program. Visual Basic performs the necessary conversions for you.

To declare a variant, you can turn off the Strict option and use the `Dim` statement without specifying a type, as follows:

```
Dim myVar
```

You can use object variables in both numeric and string calculations. Suppose that the variable *modemSpeed* has been declared as Object with one of the following statements:

```
Dim modemSpeed             ' with Option Strict = Off
Dim modemSpeed As Object   ' with Option Strict = On
```

Later in your code, you assign the following value to it:

```
modemSpeed = "28.8"
```

You can treat the *modemSpeed* variable as a string and use it in statements such as the following:

```
MsgBox "We suggest a " & modemSpeed & " modem."
```

This statement displays the following message:

```
"We suggest a 28.8 modem."
```

You can also treat the *modemSpeed* variable as a numeric value, as in the following statement:

```
Debug.WriteLine "A " & modemSpeed & " modem can transfer " &
            modemSpeed * 1024 / 8 & " bytes per second."
```

This statement displays the following message:

```
"A 28.8 modem can transfer 3686.4 bytes per second."
```

The first instance of the *modemSpeed* variable in the preceding statement is treated as a string because this is the variant's type according to the assignment statement (we assigned a string to it). The second instance, however, is treated as a number (a single-precision number). The compiler sees that it's used in a numeric calculation and converts it to a double value before using it.

Another example of this behavior of variants can be seen in the following statements:

```
Dim I, S
I = 10
S = "11"
Debug.WriteLine(I + S)
Debug.WriteLine(I & S)
```

The first `WriteLine` statement will display the numeric value **21**, whereas the second statement will print the string **1011**. The plus operator (+) tells VB to add two values. In doing so, VB must convert the two strings into numeric values and then add them. The concatenation operator (&) tells VB to concatenate the two strings.

Visual Basic knows how to handle object variables in a way that makes sense. The result may not be what you had in mind, but it certainly is dictated by common sense. If you really want to concatenate the strings 10 and 11, you should use the concatenation operator (&), which tells Visual Basic exactly what to do. Quite impressive, but for many programmers, this is a strange behavior that can lead to subtle errors — and they avoid it. Keep in mind that if the value of the *S* variable were the string `A1`, then the code would compile fine but would crash at runtime. And this is what we want to avoid at all costs: an application that compiles without warnings but crashes at runtime. Using strongly typed variables is one of the precautions you can take to avoid runtime errors. Keep in mind that a program that prompts users for data, or reads it from a file, may work for quite a while, just because it's reading valid data, and crash when it encounters invalid data. It's up to you to decide whether to use variants and how far you will go with them. Sure, you can perform tricks with variants, but you shouldn't overuse them to the point that others can't read your code.

Variables as Objects

Variables in Visual Basic are more than just names or placeholders for values. They're intelligent entities that can not only store but also process their values. I don't mean to scare you, but I think you should be told: VB variables are objects. And here's why: A variable that holds dates is declared as such with the following statement:

```
Dim expiration As Date
```

To assign a date value to the *expiration* variable, use a statement like this:

```
expiration = #1/1/2003#
```

So far, nothing out of the ordinary; this is how we always used variables, in most languages. In addition to holding a date, however, the *expiration* variable can manipulate dates. The following expression will return a new date that's three years ahead of the date stored in the expiration variable:

```
expiration.AddYears(3)
```

The `AddYears` method returns a new date, which you can assign to another date variable:

```
Dim newExpiration As Date
newExpiration = expiration.AddYears(3)
```

`AddYears` is a method that knows how to add a number of years to a Date variable. By adding a number of years (or months, or days) to a date, we get back another date. The method will take into consideration the number of days in each month and the leap years, which is a totally nontrivial task if we had to code it ourselves. There are similarly named methods for adding months, days, and so on. In addition to methods, the Date type exposes properties, such as the `Month` and `Day` properties, which return the date's month and day number, respectively. The keywords following the period after the variable's name are called *methods* and *properties*, just like the properties and methods of the controls you place on a form to create your application's visual interface. The methods and properties (or the *members*) of a variable expose the functionality that's built into the class representing the variable itself. Without this built-in functionality, you'd have to write some serious code to extract the month from a date variable, to add a number of days to a given date, to figure out whether a character is a letter or a digit or a punctuation symbol, and so on. Much of the functionality that you'll need in an application that manipulates dates, numbers, or text has already been built into the variables themselves.

Don't let the terminology scare you. Think of variables as placeholders for values and access their functionality with expressions like the ones shown earlier. Start using variables to store values, and if you need to process them, enter a variable's name followed by a period to see a list of the members it exposes. In most cases, you'll be able to figure out what these members do by just reading their names. I'll come back to the concept of variables as objects, but I wanted to hit it right off the bat. A more detailed discussion of the notion of variables as objects can be found in Chapter 8, ''Working with Objects,'' which discusses objects in detail.

Basic Data Types versus Objects

Programming languages can treat simple variables much more efficiently than they treat objects. An integer takes two bytes in memory, and the compiler will generate very efficient code to manipulate an integer variable (add it to another numeric value, compare it to another integer, and so on). If you declare an integer variable and use it in your code as such, Visual Basic doesn't create an object to represent this value. It creates a new variable for storing integers, like good old BASIC. After you call one of the variable's methods, the compiler emits code to create the actual object. This process is called *boxing*, and it introduces a small delay, which is truly insignificant compared to the convenience of manipulating a variable through its methods.

As you've seen by now, variables are objects. This shouldn't come as a surprise, but it's an odd concept for programmers with no experience in object-oriented programming. We haven't covered objects and classes formally yet, but you have a good idea of what an object is. It's an entity that exposes some functionality by means of properties and methods. The TextBox control is an object and it exposes the `Text` property, which allows you to read or set the text on the control. Any name followed by a period and another name signifies an object. The name after the period is a property or method of the object.

Converting Variable Types

In many situations, you will need to convert variables from one type into another. Table 2.3 shows the methods of the Convert class that perform data-type conversions.

TABLE 2.3: The data-type conversion methods of the Convert class

METHOD	CONVERTS ITS ARGUMENT TO
`ToBoolean`	Boolean
`ToByte`	Byte
`ToChar`	Unicode character
`ToDateTime`	Date
`ToDecimal`	Decimal
`ToDouble`	Double
`ToInt16`	Short Integer (2-byte integer, `Int16`)
`ToInt32`	Integer (4-byte integer, `Int32`)
`ToInt64`	Long (8-byte integer, `Int64`)
`ToSByte`	Signed Byte
`CShort`	Short (2-byte integer, `Int16`)
`ToSingle`	Single
`ToString`	String
`ToUInt16`	Unsigned Integer (2-byte integer, `Int16`)
`ToUInt32`	Unsigned Integer (4-byte integer, `Int32`)
`ToUInt64`	Unsigned Long (8-byte integer, `Int64`)

In addition to the methods of the Convert class, you can still use the data-conversion functions of VB (`CInt()` to convert a numeric value to an Integer, `CDbl()` to convert a numeric value to a Double, `CSng()` to convert a numeric value to a Single, and so on), which you can look up in the documentation. If you're writing new applications in VB 2010, use the new Convert class to convert between data types.

To convert the variable initialized as

```
Dim A As Integer
```

to a Double, use the `ToDouble` method of the Convert class:

```
Dim B As Double
B = Convert.ToDouble(A)
```

Suppose you have declared two integers, as follows:

```
Dim A As Integer, B As Integer
A = 23
B = 7
```

The result of the operation `A / B` will be a Double value. The statement

```
Debug.Write(A / B)
```

displays the value 3.28571428571429. The result is a Double value, which provides the greatest possible accuracy. If you attempt to assign the result to a variable that hasn't been declared as Double and the Strict option is on, the editor will generate an error message. No other data type can accept this value without loss of accuracy. To store the result to a Single variable, you must convert it explicitly with a statement like the following:

```
Dim C As Single = Convert.ToSingle(A / B)
```

You can also use the `DirectCast()` function to convert a variable or expression from one type to another. The `DirectCast()` function is identical to the `CType()` function. Let's say the variable *A* has been declared as String and holds the value 34.56. The following statement converts the value of the *A* variable to a Decimal value and uses it in a calculation:

```
Dim A As String = "34.56"
Dim B As Double
B = DirectCast(A, Double) / 1.14
```

The conversion is necessary only if the Strict option is on, but it's a good practice to perform your conversions explicitly. The following section explains what might happen if your code relies on implicit conversions.

Widening and Narrowing Conversions

In some situations, VB 2010 will convert data types automatically, but not always. Let's say you have declared and initialized two variables, an Integer and a Double, with the following statements:

```
Dim count As Integer = 99
Dim pi As Double = 3.1415926535897931
```

If the Strict option is off and you assign the variable `pi` to the `count` variable, the `count` variable's new value will be 3. (The Double value will be rounded to an Integer value, according to the variable's type.) Although this may be what you want, in most cases it's an oversight that will lead to incorrect results.

If the Strict option is on and you attempt to perform the same assignment, the compiler will generate an error message to the effect that you can't convert a Double to an Integer. The exact message is *Option Strict disallows implicit conversions from Double to Integer*.

When the Strict option is on, VB 2010 will allow conversions that do not result in loss of accuracy (precision) or magnitude. These conversions are called *widening conversions*. When you assign an Integer value to a Double variable, no accuracy or magnitude is lost. This is a widening conversion because it goes from a narrower to a wider type and will therefore be allowed when Strict is on.

On the other hand, when you assign a Double value to an Integer variable, some accuracy could be lost (the decimal digits may be truncated). This is a *narrowing conversion* because we go from a data type that can represent a wider range of values to a data type that can represent a narrower range of values. With the Strict option on, such a conversion will not be allowed.

Because you, the programmer, are in control, you might want to give up the accuracy — presumably, it's no longer needed. Table 2.4 summarizes the widening conversions that VB 2010 will perform for you automatically.

Table 2.4: VB 2010 widening conversions

Original Data Type	Wider Data Type
Any type	Object
Byte	Short, Integer, Long, Decimal, Single, Double
Short	Integer, Long, Decimal, Single, Double
Integer	Long, Decimal, Single, Double
Long	Decimal, Single, Double
Decimal	Single, Double
Single	Double
Double	None
Char	String

If the Strict option is on, the compiler will point out all the statements that may cause runtime errors and you can reevaluate your choice of variable types. Even if you're working with the Strict option off, you can turn it on momentarily to see the compiler's warnings and then turn it off again.

Formatting Numbers

So far, you've seen how to use the basic data types. Let me digress here for a moment and mention that the basic data types are no longer part of the language (Visual Basic or C#). They're actually part of the Common Language Runtime (CLR), which is a basic component of Visual Studio (actually, it's the core of Visual Studio and it's shared by all languages that can be used with Visual Studio). You can treat this note as fine print for now, but don't be surprised when you read in the documentation that the basic data types are part of the CLR. All data types expose a `ToString` method, which returns the variable's value (a number or date) as a string so that it can be used with other strings in your code. The `ToString` method formats numbers and dates in many ways, and it's probably one of the most commonly used methods. You can call the `ToString` method without any arguments, as we have done so far, to convert any value to a string. With many types, the `ToString` method, however, accepts an optional argument, which determines how the value will be formatted as a string. For example, you can format a number as currency by prefixing it with the appropriate symbol (such as the dollar symbol) and displaying it with two decimal digits, and you can display dates in many formats. Some reports require that negative amounts are enclosed in parentheses. The `ToString` method allows you to display numbers and dates, and any other type, in any way you wish.

Notice that `ToString` is a method, not a property. It returns a value that you can assign to a string variable or pass as arguments to a function such as `MsgBox()`, but the original value is not affected. The `ToString` method can also format a value if called with an optional argument:

```
ToString(formatString)
```

The *`formatString`* argument is a format specifier (a string that specifies the exact format to be applied to the variable). This argument can be a specific character that corresponds to a predetermined format (a standard format string, as it's called) or a string of characters that have special meaning in formatting numeric values (a picture format string). Use standard format strings for the most common formatting options, and use picture strings to specify unusual formatting requirements. To format the value 9959.95 as a dollar amount, you can use the `C` format specifier, which stands for *Currency*:

```
Dim Amnt As Single = 9959.95
Dim strAmnt As String
strAmnt = Amnt.ToString("C")
```

Or use the following picture numeric format string:

```
strAmnt = Amnt.ToString("$#,###.00")
```

Both statements will format the value as $9,959.95. If you're using a non-U.S. version of Windows, the currency symbol will change accordingly. If you're in the United States, use the Regional And Language Options tool in Control Panel to temporarily change the current culture to a European one and the amount will be formatted with the Euro sign.

The picture format string is made up of literals and characters that have special meaning in formatting. The dollar sign has no special meaning and will appear as is. The # symbol is a digit placeholder; all # symbols will be replaced by numeric digits, starting from the right. If the number has fewer digits than specified in the string, the extra symbols to the left will be ignored. The comma tells the `ToString` method to insert a comma between thousands. The period is the decimal point, which is followed by two more digit placeholders. Unlike the # sign, the 0 is a special placeholder: If there are not enough digits in the number for all the zeros you've specified, a 0 will appear in the place of the missing decimal digits. If the original value had been 9959.9, for example, the last statement would have formatted it as $9,959.90. If you used the # placeholder instead, the string returned by the `ToString` method would have a single decimal digit.

Standard Numeric Format Strings

The `ToString` method of the numeric data types recognizes the standard numeric format strings shown in Table 2.5.

The format character can be followed by an integer. If present, the integer value specifies the number of decimal places that are displayed. The default accuracy is two decimal digits.

Table 2.5: Standard numeric format strings

Format Character	Description	Example
C or c	Currency	`(12345.67).ToString("C")` returns $12,345.67.
D or d	Decimal	`(123456789).ToString("D")` returns 123456789. It works with integer values only.
E or e	Scientific format	`(12345.67).ToString("E")` returns 1.234567E + 004.
F or f	Fixed-point format	`(12345.67).ToString("F")` returns 12345.67.
G or g	General format	Returns a value either in fixed-point or scientific format.
N or n	Number format	`(12345.67).ToString("N")` returns 12,345.67.
P or p	Percentage	`(0.12345).ToString("N")` returns 12.35%.
R or r	Round-trip	`(1 / 3).ToString("R")` returns 0.33333333333333331 (where the G specifier would return a value with fewer decimal digits: 0.333333333333333).
X or x	Hexadecimal format	`250.ToString("X")` returns FA.

The C format string causes the ToString method to return a string representing the number as a currency value. An integer following the C determines the number of decimal digits that are displayed. If no number is provided, two digits are shown after the decimal separator. Assuming that the variable *value* has been declared as Decimal and its value is 5596, then the expression value.ToString("C") will return the string $5,596.00. If the value of the variable were 5596.4499, then the expression value.ToString("C3") would return the string $5,596.450. Also note that the C format string formats negative amounts in a pair of parentheses, as is customary in business applications.

Notice that not all format strings apply to all data types. For example, only integer values can be converted to hexadecimal format, and the D format string works with integer values only.

Picture Numeric Format Strings

If the format characters listed in Table 2.5 are not adequate for the control you need over the appearance of numeric values, you can provide your own picture format strings. Picture format strings contain special characters that allow you to format your values exactly as you like. Table 2.6 lists the picture formatting characters.

Table 2.6: Picture numeric format strings

Format Character	Description	Effect
0	Display zero placeholder	Results in a nonsignificant zero if a number has fewer digits than there are zeros in the format
#	Display digit placeholder	Replaces the symbol with only significant digits
.	Decimal point	Displays a period (.) character
,	Group separator	Separates number groups — for example, 1,000
%	Percent notation	Displays a % character
E + 0, E − 0, e + 0, e − 0	Exponent notation	Formats the output of exponent notation
\	Literal character	Used with traditional formatting sequences such as \n (newline)
""	Literal string	Displays any string within single or double quotation marks literally
;	Section separator	Specifies different output if the numeric value to be formatted is positive, negative, or zero

The following statements will print the highlighted values:

```
Dim Amount As Decimal = 42492.45
Debug.WriteLine(Amount.ToString("$#,###.00"))
$42,492.45
Amount = 0.2678
Debug.WriteLine(Amount.ToString("0.000"))
0.268
Amount = -24.95
Debug.WriteLine(Amount.ToString("$#,###.00;($#,###.00)"))
($24.95)
```

User-Defined Data Types

In the previous sections, we used variables to store individual values (or scalar values, as they're called). As a matter of fact, most programs store sets of data of different types. For example, a program for balancing your checkbook must store several pieces of information for each check: the check's number, amount, date, and so on. All these pieces of information are necessary to process the checks, and ideally, they should be stored together.

What we need is a variable that can hold multiple related values of the same or different type. You can create custom data types that are made up of multiple values using Structures. A Structure allows you to combine multiple values of the basic data types and handle them as a whole. For example, each check in a checkbook-balancing application is stored in a separate Structure (or record), as shown in Figure 2.3. When you recall a given check, you need all the information stored in the Structure.

FIGURE 2.3
Pictorial representation of a structure

Record Structure

Check Number	Check Date	Check Amount	Check Paid To

Array of Records

275	11/04/2010	104.25	Gas Co.
276	11/09/2010	48.76	Books
277	11/12/2010	200.00	VISA
278	11/21/2010	631.50	Rent

To define a Structure in VB 2010, use the `Structure` statement, which has the following syntax:

```
Structure structureName
   Dim variable1 As varType
   Dim variable2 As varType
   …
   Dim variablen As varType
End Structure
```

varType can be any of the data types supported by the CLR or the name of another Structure that has been defined already. The `Dim` statement can be replaced by the `Private` or `Public` access modifiers. For Structures, `Dim` is equivalent to `Public`.

After this declaration, you have in essence created a new data type that you can use in your application. *structureName* can be used anywhere you'd use any of the base types (Integers, Doubles, and so on). You can declare variables of this type and manipulate them as you manipulate all other variables (with a little extra typing). The declaration for the CheckRecord Structure shown in Figure 2.3 is as follows:

```
Structure CheckRecord
   Dim CheckNumber As Integer
   Dim CheckDate As Date
   Dim CheckAmount As Single
   Dim CheckPaidTo As String
End Structure
```

This declaration must appear outside any procedure; you can't declare a Structure in a subroutine or function. Once declared, the CheckRecord Structure becomes a new data type for your application.

To declare variables of this new type, use a statement such as this one:

```
Dim check1 As CheckRecord, check2 As CheckRecord
```

To assign a value to one of these variables, you must separately assign a value to each one of its components (they are called fields), which can be accessed by combining the name of the variable and the name of a field separated by a period, as follows:

```
check1.CheckNumber = 275
```

Actually, as soon as you type the period following the variable's name, a list of all members to the CheckRecord Structure will appear, as shown in Figure 2.4. Notice that the Structure supports a few members on its own. You didn't write any code for the Equals, GetType, and ToString members, but they're standard members of any Structure object, and you can use them in your code. Both the GetType and ToString methods will return a string like ProjectName.FormName + CheckRecord. You can provide your own implementation of the ToString method, which will return a more meaningful string:

```
Public Overrides Function ToString() As String
    Return "CHECK # " & CheckNumber & " FOR " & CheckAmount.ToString("C")
End Function
```

I haven't discusses the Overrides keyword yet; it tells the compiler to override the default implementation of the ToString method. For the time being, use it as shown here to create your custom ToString method. This, as well as other object-related terms, are discussed in detail in Chapter 8.

As you understand, Structures are a lot like objects that expose their fields as properties and then expose a few members of their own. The following statements initialize a variable of the CheckRecord type:

```
check2.CheckNumber = 275
check2.CheckDate = #09/12/2010#
```

```
check2.CheckAmount = 104.25
check2.CheckPaidTo = "Gas Co."
```

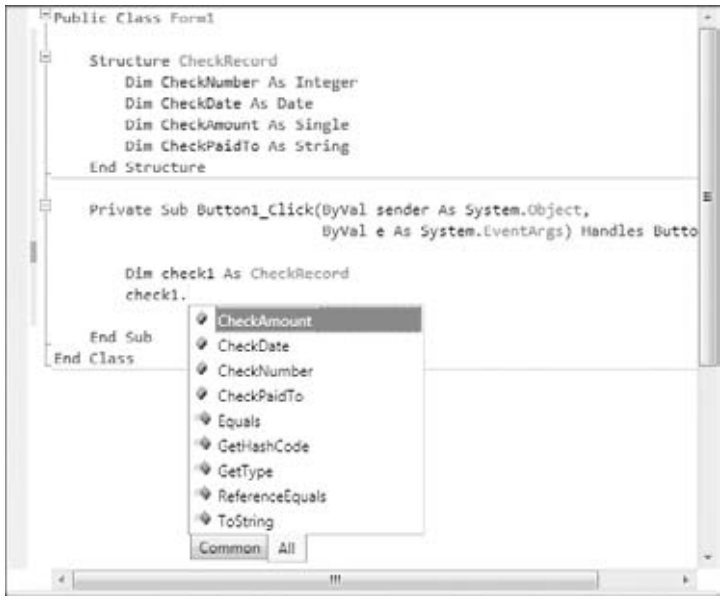

FIGURE 2.4
Variables of custom types expose their members as properties.

Examining Variable Types

Besides setting the types of variables and the functions for converting between types, Visual Basic provides the `GetType` method. `GetType` returns a string containing the name of the variable type (`Int32`, `Decimal`, and so on). All variables expose this method automatically, and you can call it like this:

```
Dim var As Double
Debug.WriteLine "The variable's type is " & var.GetType.ToString
```

There's also a `GetType` operator, which accepts as an argument a type and returns a Type object for the specific data type. The `GetType` method and `GetType` operator are used mostly in `If` structures, like the following one:

```
If var.GetType() Is GetType(Double) Then
   '  code to handle a Double value
End If
```

Notice that the code doesn't reference data type names directly. Instead, it uses the value returned by the `GetType` operator to retrieve the type of the class System.Double and then compares this value to the variable's type with the `Is` (or the `IsNot`) keyword. If you attempt to express this comparison with the equals operator (=), the editor will detect the error and

suggest that you use the `Is` operator. This syntax is a bit arcane for BASIC developers; just make a note, and when you need to find out a variable's type in your application, use it as is.

Is It a Number, String, or Date?

Another set of Visual Basic functions returns variable data types, but not the exact type. They return a True/False value indicating whether a variable holds a numeric value, a date, or an array. The following functions are used to validate user input, as well as data stored in files, before you process them.

`IsNumeric()` Returns True if its argument is a number (Short, Integer, Long, Single, Double, Decimal). Use this function to determine whether a variable holds a numeric value before passing it to a procedure that expects a numeric value or before processing it as a number. The following statements keep prompting the user with an InputBox for a numeric value. The user must enter a numeric value or click the Cancel button to exit. As long as the user enters non-numeric values, the InputBox keeps popping up and prompting for a numeric value:

```
Dim strAge as String = ""
Dim Age As Integer
While Not IsNumeric(strAge)
    strAge = InputBox("lease enter your age")
End While
Age = Convert.ToInt16(strAge)
```

The variable *`strAge`* is initialized to a non-numeric value so that the `While`…`End While` loop will be executed at least once.

`IsDate()` Returns True if its argument is a valid date (or time). The following expressions return True because they all represent valid dates:

```
IsDate(#10/12/2010#)
IsDate("10/12/2010")
IsDate("October 12, 2010")
```

`IsArray()` Returns True if its argument is an array.

A Variable's Scope

In addition to a type, a variable has a scope. The scope (or visibility) of a variable is the section of the application that can see and manipulate the variable. If a variable is declared within a procedure, only the code in the specific procedure has access to that variable; the variable doesn't exist for the rest of the application. When the variable's scope is limited to a procedure, it's called local.

Suppose that you're coding the handler for the `Click` event of a button to calculate the sum of all even numbers in the range 0 to 100. One possible implementation is shown in Listing 2.3.

LISTING 2.3: Summing even numbers

```
Private Sub Button1_Click(ByVal sender As Object, _
              ByVal e As System.EventArguments) _
              Handles Button1.Click
   Dim i As Integer
   Dim Sum As Integer = 0
   For i = 0 to 100 Step 2
      Sum = Sum + i
   Next
   MsgBox "The sum is " & Sum.ToString
End Sub
```

The variables *i* and *Sum* are local to the `Button1_Click()` procedure. If you attempt to set the value of the *Sum* variable from within another procedure, Visual Basic will complain that the variable hasn't been declared. (Or, if you have turned off the Explicit option, it will create another *Sum* variable, initialize it to zero, and then use it. But this won't affect the variable *Sum* in the `Button1_Click()` subroutine.) The *Sum* variable is said to have procedure-level scope; it's visible within the procedure and invisible outside the procedure.

Sometimes, however, you'll need to use a variable with a broader scope: a variable that's available to all procedures within the same file. This variable, which must be declared outside any procedure, is said to have a module-level scope. In principle, you could declare all variables outside the procedures that use them, but this would lead to problems. Every procedure in the file would have access to any variable, and you would need to be extremely careful not to change the value of a variable without good reason. Variables that are needed by a single procedure (such as loop counters) should be declared in that procedure.

Another type of scope is the block-level scope. Variables introduced in a block of code, such as an `If` statement or a loop, are local to the block but invisible outside the block. Let's revise the previous code segment so that it calculates the sum of squares. To carry out the calculation, we first compute the square of each value and then sum the squares. The square of each value is stored to a variable that won't be used outside the loop, so we can define the *sqrValue* variable in the loop's block and make it local to this specific loop, as shown in Listing 2.4.

LISTING 2.4: A variable scoped in its own block

```
Private Sub Button1_Click(ByVal sender As Object, _
              ByVal e As System.EventArguments) _
              Handles Button1.Click
   Dim i, Sum As Integer
   For i = 0 to 100 Step 2
      Dim sqrValue As Integer
      sqrValue = i * i
      Sum = Sum + sqrValue
   Next
   MsgBox "The sum of the squares is " & Sum
End Sub
```

The *sqrValue* variable is not visible outside the block of the For…Next loop. If you attempt to use it before the For statement or after the Next statement, the code won't compile.

The *sqrValue* variable maintains its value between iterations. The block-level variable is not initialized at each iteration, even though there's a Dim statement in the loop.

Finally, in some situations, the entire application must access a certain variable. In this case, the variable must be declared as Public. Public variables have a global scope; they are visible from any part of the application. To declare a public variable, use a Public statement in place of a Dim statement. Moreover, you can't declare public variables in a procedure. If you have multiple forms in your application and you want the code in one form to see a certain variable in another form, you can use the Public modifier.

So, why do we need so many types of scope? You'll develop a better understanding of scope and which type of scope to use for each variable as you get involved in larger projects. In general, you should try to limit the scope of your variables as much as possible. If all variables were declared within procedures, you could use the same name for storing a temporary value in each procedure and be sure that one procedure's variables wouldn't interfere with those of another procedure, even if you use the same name.

A Variable's Lifetime

In addition to type and scope, variables have a lifetime, which is the period for which they retain their value. Variables declared as Public exist for the lifetime of the application. Local variables, declared within procedures with the Dim or Private statement, live as long as the procedure. When the procedure finishes, the local variables cease to exist, and the allocated memory is returned to the system. Of course, the same procedure can be called again, and then the local variables are re-created and initialized again. If a procedure calls another, its local variables retain their values while the called procedure is running.

You also can force a local variable to preserve its value between procedure calls by using the Static keyword. Suppose that the user of your application can enter numeric values at any time. One of the tasks performed by the application is to track the average of the numeric values. Instead of adding all the values each time the user adds a new value and dividing by the count, you can keep a running total with the function RunningAvg(), which is shown in Listing 2.5.

LISTING 2.5: Calculations with global variables

```
Function RunningAvg(ByVal newValue As Double) As Double
   CurrentTotal = CurrentTotal + newValue
   TotalItems = TotalItems + 1
   RunningAvg = CurrentTotal / TotalItems
End Function
```

You must declare the variables *CurrentTotal* and *TotalItems* outside the function so that their values are preserved between calls. Alternatively, you can declare them in the function with the Static keyword, as shown in Listing 2.6.

LISTING 2.6: Calculations with local Static variables

```
Function RunningAvg(ByVal newValue As Double) As Double
   Static CurrentTotal As Double
```

```
    Static TotalItems As Integer
    CurrentTotal = CurrentTotal + newValue
    TotalItems = TotalItems + 1
    RunningAvg = CurrentTotal / TotalItems
End Function
```

The advantage of using static variables is that they help you minimize the number of total variables in the application. All you need is the running average, which the `RunningAvg()` function provides without making its variables visible to the rest of the application. Therefore, you don't risk changing the variable values from within other procedures.

Variables declared in a form module outside any procedure take effect when the form is loaded and cease to exist when the form is unloaded. If the form is loaded again, its variables are initialized as if it's being loaded for the first time.

Variables are initialized when they're declared, according to their type. Numeric variables are initialized to zero, string variables are initialized to a blank string, and object variables are initialized to Nothing.

Constants

Some variables don't change value during the execution of a program. These variables are constants that appear many times in your code. For instance, if your program does math calculations, the value of pi (3.14159...) might appear many times. Instead of typing the value 3.14159 over and over again, you can define a constant, name it `pi`, and use the name of the constant in your code. The statement

```
circumference = 2 * pi * radius
```

is much easier to understand than the equivalent

```
circumference = 2 * 3.14159 * radius
```

The manner in which you declare constants is similar to the manner in which you declare variables except that you use the `Const` keyword, and in addition to supplying the constant's name, you must also supply a value, as follows:

```
Const constantname As type = value
```

Constants also have a scope and can be `Public` or `Private`. The constant *`pi`*, for instance, is usually declared in a module as `Public` so that every procedure can access it:

```
Public Const pi As Double = 3.14159265358979
```

The rules for naming variables also apply to naming constants. The constant's value is a literal value or a simple expression composed of numeric or string constants and operators. You can't use functions in declaring constants. By the way, the specific value I used for this example need not be stored in a constant. Use the `pi` member of the Math class instead (`Math.pi`).

Arrays

A standard structure for storing data in any programming language is the array. Whereas individual variables can hold single entities, such as one number, one date, or one string, arrays can hold sets of data of the same type (a set of numbers, a series of dates, and so on). An array has a name, as does a variable, and the values stored in it can be accessed by a number or index.

For example, you could use the variable *Salary* to store a person's salary:

```
Salary = 34000
```

But what if you wanted to store the salaries of 16 employees? You could either declare 16 variables — *Salary1*, *Salary2*, and so on up to *Salary16* — or declare an array with 16 elements. An array is similar to a variable: It has a name and multiple values. Each value is identified by an index (an integer value) that follows the array's name in parentheses. Each different value is an element of the array. If the array *Salaries* holds the salaries of 16 employees, the element *Salaries(0)* holds the salary of the first employee, the element *Salaries(1)* holds the salary of the second employee, and so on up to the element *Salaries(15)*. Yes, the default indexing of arrays starts at zero, as odd as it may be for traditional BASIC developers.

Declaring Arrays

Arrays must be declared with the `Dim` (or `Public`) statement followed by the name of the array and the index of the last element in the array in parentheses, as in this example:

```
Dim Salary(15) As Integer
```

Salary is the name of an array that holds 16 values (the salaries of the 16 employees) with indices ranging from 0 to 15. *Salary(0)* is the first person's salary, *Salary(1)* the second person's salary, and so on. All you have to do is remember who corresponds to each salary, but even this data can be handled by another array. To do this, you'd declare another array of 16 elements:

```
Dim Names(15) As String
```

Then assign values to the elements of both arrays:

```
Names(0) = "Joe Doe"
Salary(0) = 34000
Names(1) = "Beth York"
Salary(1) = 62000
…
Names(15) = "Peter Smack"
Salary(15) = 10300
```

This structure is more compact and more convenient than having to hard-code the names of employees and their salaries in variables.

All elements in an array have the same data type. Of course, when the data type is Object, the individual elements can contain different kinds of data (objects, strings, numbers, and so on).

Arrays, like variables, are not limited to the basic data types. You can declare arrays that hold any type of data, including objects. The following array holds colors, which can be used later in the code as arguments to the various functions that draw shapes:

```
Dim colors(2) As Color
colors(0) = Color.BurlyWood
colors(1) = Color.AliceBlue
colors(2) = Color.Sienna
```

The Color class represents colors, and among the properties it exposes are the names of the colors it recognizes.

A better technique for storing names and salaries is to create a structure and then declare an array of this type. The following structure holds names and salaries:

```
Structure Employee
   Dim Name As String
   Dim Salary As Decimal
End Structure
```

Insert this declaration in a form's code file, outside any procedure. Then create an array of the Employee type:

```
Dim Emps(15) As Employee
```

Each element in the *Emps* array exposes two fields, and you can assign values to them by using statements such as the following:

```
Emps(2).Name = "Beth York"
Emps(2).Salary = 62000
```

The advantage of using an array of structures instead of multiple arrays is that the related information will always be located under the same index. The code is more compact, and you need not maintain multiple arrays.

Initializing Arrays

Just as you can initialize variables in the same line in which you declare them, you can initialize arrays, too, with the following constructor (an array initializer, as it's called):

```
Dim nameArray() As type = {entry0, entry1, … entryN}
```

Here's an example that initializes an array of strings:

```
Dim Names() As String = {"Joe Doe", "Peter Smack"}
```

This statement is equivalent to the following statements, which declare an array with two elements and then set their values:

```
Dim Names(1) As String
```

```
Names(0) = "Joe Doe"
Names(1) = "Peter Smack"
```

The number of elements in the curly brackets following the array's declaration determines the dimensions of the array, and you can't add new elements to the array without resizing it. If you need to resize the array in your code dynamically, you must use the `ReDim` statement and supply the new size of the array in parentheses.

Array Limits

The first element of an array has index 0. The number that appears in parentheses in the `Dim` statement is one fewer than the array's total capacity and is the array's upper limit (or upper bound). The index of the last element of an array (its upper bound) is given by the method `GetUpperBound`, which accepts as an argument the dimension of the array and returns the upper bound for this dimension. The arrays we have examined so far are one-dimensional, and the argument to be passed to the `GetUpperBound` method is the value 0. The total number of elements in the array is given by the method `GetLength`, which also accepts a dimension as an argument. The upper bound of the following array is 19, and the capacity of the array is 20 elements:

```
Dim Names(19) As Integer
```

The first element is *`Names(0)`*, and the last is *`Names(19)`*. If you execute the following statements, the highlighted values will appear in the Output window:

```
Debug.WriteLine(Names.GetLowerBound(0))
0
Debug.WriteLine(Names.GetUpperBound(0))
19
```

To assign a value to the first and last element of the *`Names`* array, use the following statements:

```
Names(0) = "First entry"
Names(19) = "Last entry"
```

To iterate through the array elements, use a loop like the following one:

```
Dim i As Integer, myArray(19) As Integer
For i = 0 To myArray.GetUpperBound(0)
 myArray(i) = i * 1000
Next
```

The number of elements in an array is given by the expression `myArray.GetUpperBound(0) + 1`. You can also use the array's `Length` property to retrieve the count of elements. The following statement will print the number of elements in the array *`myArray`* in the Output window:

```
Debug.WriteLine(myArray.Length)
```

Still confused with the zero-indexing scheme, the count of elements, and the index of the last element in the array? You can make the array a little larger than it needs to be and ignore the first element. Just make sure that you never use the zero element in your code — don't store a value in the element `Array(0)`, and you can then ignore this element. To get 20 elements, declare an array with 21 elements as `Dim MyArray(20) As type` and then ignore the first element.

Multidimensional Arrays

One-dimensional arrays, such as those presented so far, are good for storing long sequences of one-dimensional data (such as names or temperatures). But how would you store a list of cities *and* their average temperatures in an array? Or names and scores, years and profits, or data with more than two dimensions, such as products, prices, and units in stock? In some situations, you will want to store sequences of multidimensional data. You can store the same data more conveniently in an array of as many dimensions as needed.

Figure 2.5 shows two one-dimensional arrays — one of them with city names, the other with temperatures. The name of the third city would be *`City(2)`*, and its temperature would be *`Temperature(2)`*.

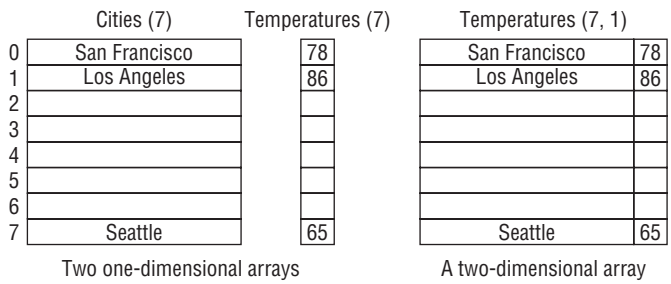

FIGURE 2.5
Two one-dimensional arrays and the equivalent two-dimensional array

A two-dimensional array has two indices: The first identifies the row (the order of the city in the array), and the second identifies the column (city or temperature). To access the name and temperature of the third city in the two-dimensional array, use the following indices:

```
Temperatures(2, 0)     ' is the third city's name
Temperatures(2, 1)     ' is the third city's average temperature
```

The benefit of using multidimensional arrays is that they're conceptually easier to manage. Suppose you're writing a game and want to track the positions of certain pieces on a board. Each square on the board is identified by two numbers: its horizontal and vertical coordinates. The obvious structure for tracking the board's squares is a two-dimensional array, in which the first index corresponds to the row number and the second corresponds to the column number. The array could be declared as follows:

```
Dim Board(9, 9) As Integer
```

When a piece is moved from the square in the first row and first column to the square in the third row and fifth column, you assign the value 0 to the element that corresponds to the initial position:

```
Board(0, 0) = 0
```

And you assign 1 to the square to which it was moved to indicate the new state of the board:

```
Board(2, 4) = 1
```

To find out whether a piece is on the top-left square, you'd use the following statement:

```
If Board(0, 0) = 1 Then
   '  piece found
Else
   '  empty square
End If
```

This notation can be extended to more than two dimensions. The following statement creates an array with 1,000 elements (10 by 10 by 10):

```
Dim Matrix(9, 9, 9)
```

You can think of a three-dimensional array as a cube made up of overlaid two-dimensional arrays, such as the one shown in Figure 2.6.

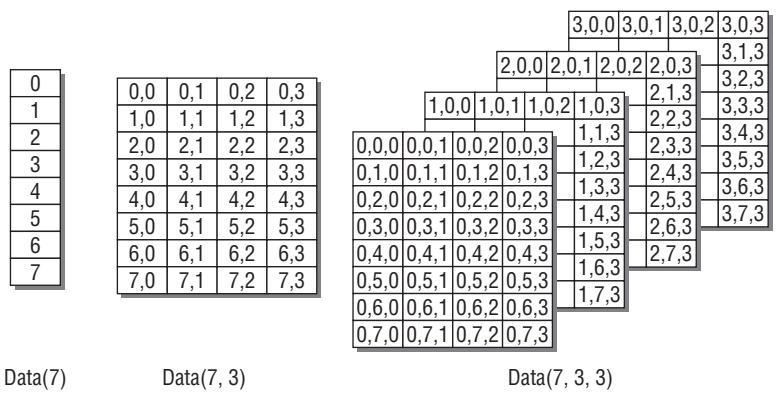

FIGURE 2.6
Pictorial representations of one-, two-, and three-dimensional arrays

It is possible to initialize a multidimensional array with a single statement, just as you do with a one-dimensional array. You must insert enough commas in the parentheses following the array name to indicate the array's rank. The following statements initialize a two-dimensional array and then print a couple of its elements:

```
Dim a(,) As Integer = {{10, 20, 30}, {11, 21, 31}, {12, 22, 32}}
Console.WriteLine(a(0, 1))     ' will print 20
Console.WriteLine(a(2, 2))     ' will print 32
```

You should break the line that initializes the dimensions of the array into multiple lines to make your code easier to read:

```
Dim a(,) As Integer = {{10, 20, 30},
                       {11, 21, 31},
                       {12, 22, 32}}
```

If the array has more than one dimension, you can find out the number of dimensions with the `Array.Rank` property. Let's say you have declared an array for storing names and salaries by using the following statements:

```
Dim Employees(1,99) As Employee
```

To find out the number of dimensions, use the following statement:

```
Employees.Rank
```

When using the `Length` property to find out the number of elements in a multidimensional array, you will get back the total number of elements in the array (2 × 100 for our example). To find out the number of elements in a specific dimension, use the `GetLength` method, passing as an argument a specific dimension. The following expressions will return the number of elements in the two dimensions of the array:

```
Debug.WriteLine(Employees.GetLength(0))
2
Debug.WriteLine(Employees.GetLength(1))
100
```

Because the index of the first array element is zero, the index of the last element is the length of the array minus 1. Let's say you have declared an array with the following statement to store player statistics for 15 players and there are five values per player:

```
Dim Statistics(14, 4) As Integer
```

The following statements will return the highlighted values shown beneath them:

```
Debug.WriteLine(Statistics.Rank)
2                          ' dimensions in array
Debug.WriteLine(Statistics.Length)
75                         ' total elements in array
Debug.WriteLine(Statistics.GetLength(0))
15                         ' elements in first dimension
Debug.WriteLine(Statistics.GetLength(1))
5                          ' elements in second dimension
Debug.WriteLine(Statistics.GetUpperBound(0))
14                         ' last index in the first dimension
Debug.WriteLine(Statistics.GetUpperBound(1))
4                          ' last index in the second dimension
```

Multidimensional arrays are becoming obsolete because arrays (and other collections) of custom structures and objects are more flexible and convenient.

Collections

Historically, arrays are the primary structures for storing sets of data, and for years they were the primary storage mechanism for in-memory data manipulation. In this field, however, where technologies grow in and out of style overnight, arrays are being replaced by other, more flexible and more powerful structures, the collections. Collections are discussed in detail in Chapter 12, but I should mention them briefly in this chapter, not only for completeness, but also because collections are used a lot in programming and you will find many examples of collections in this book's chapters.

A collection is a dynamic data storage structure: You don't have to declare the size of a collection ahead of time. Moreover, the position of the items in a collection is not nearly as important as the position of the items in an array. New items are appended to a collection with the `Add` method, while existing items are removed with the `Remove` method. (Note that there's no simple method of removing an array element, short of copying the original array to a new one and skipping the element to be removed.) The collection I just described is the List collection, which is very similar to an array. To declare a List collection, use the `New` keyword:

```
Dim names As New List(Of String)
```

The `New` keyword is literally new to you; use it to create variables that are true objects (any variable that's not of a basic data type or structure). The `New` keyword tells the compiler to create a variable of the specified type and initialize it. The List collection must be declared with a specific data type, which is specified with the `Of` keyword in parentheses. All items stored in the example *`names`* list must be strings. A related collection is the ArrayList collection, which is identical to the List collection but you don't have to declare the type of variables you intend to store in it because you can add objects of any type to an ArrayList collection.

To create a collection of color values, use the following declaration:

```
Dim colors As New List(Of Color)
```

The following statements add a few items to the two collections:

```
names.Add("Richard")
names.Add("Nancy")
colors.Add(Color.Red)
colors.Add(TextBox1.BackColor)
```

Another collection is the Dictionary collection, which allows you to identify each element by a key instead of an index value. The following statement creates a new Dictionary collection for storing names and birth dates:

```
Dim BDays As New Dictionary(Of String, Date)
```

The first data type following the `Of` keyword is the data type of the keys, while the following argument is the data type of the values you want to store to the collection. Here's how you add data to a Dictionary collection:

```
BDays.Add("Manfred", #3/24/1972#)
BDays.Add("Alfred", #11/24/1959#)
```

To retrieve the birth date of Manfred, use the following statement:

```
BDays("Manfred")
```

Finally, you can use collections to store custom objects too. Let's say you have three variables that represent checks (they're of the `CheckRecord` custom type presented earlier in this chapter in the section ''User-Defined Data Types''). You can add them to a List collection just as you would add integers or strings to a collection:

```
Dim Checks As New List(Of CheckRecord)
Checks.Add(check1)
Checks.Add(check2)
Checks.Add(check3)
```

A seasoned developer would store the same data to a Dictionary collection using the check number as an index value:

```
Dim Checks As New Dictionary(Of Integer, CheckRecord)
Checks.Add(check1.CheckNumber, check1)
```

An application that uses this structure can prompt the user for a specific check number, retrieve it by its index from the Checks collection and display it to the user. As you will see in Chapter 12, a big advantage of collections over arrays is that collections allow you to remove elements with the `Remove` method.

The Bottom Line

Declare and use variables. Programs use variables to store information during their execution, and different types of information are stored in variables of different types. Dates, for example, are stored in variables of the Date type, while text is stored in variables of the String type. The various data types expose a lot of functionality that's specific to a data type; the methods provided by each data type are listed in the IntelliSense box.

Master It How would you declare and initialize a few variables?

Master It Explain briefly the Explicit, Strict, and Infer options.

Use the native data types. The CLR recognized the following data types, which you can use in your code to declare variables: String, numeric data types (Integer, Double, and so on), Date, Char and Boolean types.

All other variables, or variables that are declared without a type, are Object variables and can store any data type or any object.

Master It How will the compiler treat the following statement?

```
Dim amount = 32
```

Create custom data types. Practical applications need to store and manipulate multiple data items, not just integers and strings. To maintain information about people, we need to store each person's name, date of birth, address, and so on. Products have a name, a description, a price, and other related items. To represent such entities in our code, we use structures, which hold many pieces of information about a specific entity together.

Master It Create a structure for storing products and populate it with data.

Use arrays. Arrays are structures for storing sets of data as opposed to single-valued variables.

Master It How would you declare an array for storing 12 names and another one for storing 100 names and Social Security numbers?