1

Visual Basic 2008 Core Elements

This chapter introduces the core elements that make up Visual Basic 2008. Every software development language has unique elements of syntax and behavior. Visual Basic 2008 has evolved significantly since Visual Basic was introduced in 1991. Although Visual Basic has its origins in traditional procedural-based programming languages, it began the transition to objects back in 1995 with Visual Basic 4.0.

With the release of Visual Basic .NET (that is, Version 7), Visual Basic became a fully object-oriented programming environment. Now with the release of Visual Basic 2008 (that is, Version 9), there are still more new features, but at the core are the same basic types and commands that have been with Visual Basic since its early stages. Object paradigms extend the *core elements* of the language. Therefore, while a very brief introduction to the existence of classes and objects within the language is presented in this chapter, the key concepts of object-oriented development are presented in detail in Chapters 2 and 3.

This chapter focuses on the core elements of the language, including questions about those language elements a new developer not familiar with Visual Basic might ask, such as where semicolons should be placed. The key topics of this chapter include the following:

- Initial syntax and keywords to understand the basic language elements
- Value versus reference types
- Primitive types
- Commands: `If Then Else`, `Select Case`
- Value types (structures)
- Reference types (classes)
- Commands: `For Each`, `For Next`, `Do While`

- Boxing
- Parameter passing `ByVal` and `ByRef`
- Variable scope
- Data type conversions, compiler options, and XML literals

The main goal of this chapter is to familiarize you with Visual Basic. The chapter begins by looking at some of the keywords and language syntax you need. Experienced developers will probably gloss over this information, as this is just a basic introduction to working with Visual Basic. After this, the chapter discusses primitive types and then introduces you to the key branching commands for Visual Basic. After you are able to handle simple conditions, the chapter introduces value and reference types. The code then looks at working with collections and introduces the primary looping control structure syntax for Visual Basic.

After this there is a brief discussion of boxing and value type conversions, conversions which often implicitly occur when values are passed as parameters. Following these topics is a discussion of *variable scope*, which defines the code that can see variables based on where they are defined in relationship to that block of code. Finally, the chapter introduces basic data type conversions, which includes looking at the compiler options for Visual Studio 2008. Visual Studio 2008 includes a new compiler option and a new data type, *XML literals*, which are also introduced in the context of conversions.

Initial Keywords and Syntax

While it would be possible to just add a giant glossary of keywords here, that isn't the focus of this chapter. Instead, a few basic elements of Visual Basic need to be spelled out, so that as you read, you can understand the examples. Chapter 7, for instance, covers working with namespaces, but some examples and other code are introduced in this chapter.

Let's begin with namespace. When .NET was being created, the developers realized that attempting to organize all of these classes required a system. A namespace is an arbitrary system that the .NET developers used to group classes containing common functionality. A namespace can have multiple levels of grouping, each separated by a period (.). Thus, the `System` namespace is the basis for classes that are used throughout .NET, while the `Microsoft.VisualBasic` namespace is used for classes in the underlying .NET Framework but specific to Visual Basic. At its most basic level, a namespace does not imply or indicate anything regarding the relationships between the class implementations in that namespace; it is just a way of managing the complexity of the .NET Framework's thousands of classes. As noted earlier, namespaces are covered in detail in Chapter 7.

Next is the `keyword` class. Chapters 2 and 3 provide details on object-oriented syntax and the related keywords for objects, but a basic definition of this keyword is needed here. The `Class` keyword designates a common set of data and behavior within your application. The class is the definition of an object, in the same way that your source code, when compiled, is the definition of an application. When someone runs your code, it is considered to be an instance of your application. Similarly, when your code creates or instantiates an object from your class definition, it is considered to be an instance of that class, or an instance of that object.

Creating an instance of an object has two parts. The first part is the `New` command, which tells the compiler to create an instance of that class. This command instructs code to call your object definition and

instantiate it. In some cases you might need to run a method and get a return value, but in most cases you use the `New` command to assign that instance of an object to a variable.

To declare a variable in Visual Basic, you use the `Dim` statement. `Dim` is short for ''dimension'' and comes from the ancient past of Basic, which preceded Visual Basic as a language. The idea is that you are telling the system to allocate or dimension a section of memory to hold data. The `Dim` statement is used to declare a variable, to which the system can then assign a value. As discussed in subsequent chapters on objects, the `Dim` statement may be replaced by another keyword such as `Public` or `Private` that not only dimensions the new value but also limits accessibility of that value. Each variable declaration uses a `Dim` statement similar to the example that follows, which declares a new variable, `winForm`:

```
Dim winForm As System.Windows.Forms.Form = New System.Windows.Forms.Form()
```

As a best practice, always set a variable equal to something when it is declared. In the preceding example, the code declares a new variable (`winForm`) of the type `Form`. This variable is then set to an instance of a `Form` object. It might also be assigned to an existing instance of a `Form` object or alternatively to `Nothing`. The `Nothing` keyword is a way of telling the system that the variable does not currently have any value, and as such is not actually using any memory on the heap. Later in this chapter, in the discussion of value and reference types, keep in mind that only reference types can be set to `Nothing`.

What do we mean when we refer to a class consisting of data and behavior? For ''data'' this means that the class specifies certain variables that are within its scope. Embedded in the class definition are zero or more `Dim` statements that create variables used to store the properties of the class. When you create an instance of this class, you create these variables; and in most cases the class contains logic to populate them. The logic used for this, and to carry out other actions, is the *behavior*. This behavior is encapsulated in what, in the object-oriented world, are known as *methods*.

However, Visual Basic doesn't have a ''method'' keyword. Instead, it has two other keywords that are brought forward from VB's days as a procedural language. The first is `Sub`. `Sub` is short for ''subroutine,'' and it defines a block of code that carries out some action. When this block of code completes, it returns control to the code that called it. To declare a function, you write code similar to the following:

```
Private Sub Load(ByVal object As System.Object)

End Sub
```

The preceding example shows the start of a method called `Load`. For now you can ignore the word `Private` at the start of this declaration; this is related to the object and is further explained in the next chapter. This method is implemented as a `Sub` because it doesn't return a value and accepts one parameter when it is called. Thus, in other languages this might be considered and written explicitly as a function that returns `Nothing`.

The preceding method declaration also includes a single parameter, which is declared as being of type `System.Object`. The meaning of the `ByVal` qualifier is explained later in this chapter, but is related to how that value is passed to this method. The code that actually loads the object would be written between the line declaring this method and the `End Sub` line.

In Visual Basic, the only difference between a `Sub` and the second method type, a `Function`, is the return type. Note that the `Function` declaration shown in the following sample code specifies the return type of

the function. A function works just like a `Sub` with the exception that a `Function` returns a value, which can be `Nothing`. This is an important distinction, because when you declare a function you expect it to include a `Return` statement. The `Return` statement is used to indicate that even though additional lines of code may remain within a `Function` or `Sub`, those lines of code should not be executed. Instead, the `Function` or `Sub` should end processing at the current line.

```
Public Function Add(ByVal ParamArray values() As Integer) As Long
    Dim result As Long = 0

    Return result
    'TODO: Implement this function
    'What if user passes in only 1 value, what about 3 or more...
    result = values(0) + values(1)
    Return result
End Function
```

In the preceding example, note that after the function initializes the second line of code, there is a `Return` statement. Because the implementation of this function isn't currently shown (it is shown later in this chapter in the discussion of parameters), the developer wanted the code to exist with a safe value until the code was completed. Moreover, there are *two* `Return` statements in the code. However, as soon as the first `Return` statement is reached, none of the remaining code in this function is executed. The `Return` statement immediately halts execution of a method, even from within a loop.

As shown in the preceding example, the function's return value is assigned to a local variable until returned as part of the `Return` statement. For a `Sub`, there would be no value on the line with the `Return` statement, as a `Sub` does not return a value when it completes. When returned, the return value is usually assigned to something else. This is shown in the next example line of code, which calls a function to retrieve the currently active control on the executing Windows Form:

```
Dim ctrl As System.Windows.Forms.Control = Me.GetContainerControl().ActiveControl()
```

The preceding example demonstrates a call to a function. The value returned by the function `ActiveControl` is of type `Control`, and the code assigns this to the variable `ctrl`. It also demonstrates another keyword that you should be aware of: `Me`. The `Me` keyword is the way, within an object, that you can reference the current instance of that object. For example, in the preceding example, the object being referenced is the current window.

You may have noticed that in all the sample code presented thus far, each line is a complete command. If you're familiar with another programming language, then you may be used to seeing a specific character that indicates the end of a complete set of commands. Several popular languages use a semicolon to indicate the end of a command line. For those who are considering Visual Basic as their first programming language, consider the English language, in which we end each complete thought with a period.

Visual Basic doesn't use visible punctuation to end each line. Instead, it views its source files more like a list, whereby each item on the list is placed on its own line. The result is that Visual Basic ends each command line with the carriage-return linefeed. In some languages, a command such as X = Y can span several lines in the source file until a semicolon or other terminating character is reached. In Visual Basic, that entire statement would be found on a single line unless the user explicitly indicates that it is to continue onto another line.

When a line ends with the underscore character, this tells Visual Basic that the code on that line does not constitute a completed set of commands. The compiler will then continue onto the next line to find the continuation of the command, and will end as soon as a carriage-return linefeed is found without an accompanying underscore. In other words, Visual Basic enables you to use exceptionally long lines and indicate that the code has been spread across multiple lines to improve readability. The following line demonstrates the use of the underscore to extend a line of code:

```
MessageBox.Show("Hello World", "A First Look at VB.NET",  _
                MessageBoxButtons.OK, MessageBoxIcon.Information)
```

In Visual Basic it is also possible to place multiple different statements on a single line, by separating the statements with colons. However, this is generally considered a poor coding practice because it reduces readability.

Console Applications

The simplest type of application is a *console application*. This application doesn't have much of a user interface; in fact, for those old enough to remember the MS-DOS operating system, a console application looks just like an MS-DOS application. It works in a command window without support for graphics or graphical devices such as a mouse. A console application is a text-based user interface that reads and writes characters from the screen.

The easiest way to create a console application is to use Visual Studio. However, for our purposes let's just look at a sample source file for a console application, as shown in the following example. Notice that the console application contains a single method, a `Sub` called `Main`. However, this `Sub` isn't contained in a class. Instead, the `Sub` is contained in a `Module`:

```
Module Module1
    Sub Main()
         Dim myObject As Object = New Object()
         Console.WriteLine("Hello World")
    End Sub
End Module
```

A module is another item dating to the procedural days of Visual Basic. It isn't a class, but rather a block of code that can contain methods, which are then referenced by classes — or, as in this case, it can represent the execution start for a program. The module in the preceding example contains a single method called `Main`. The `Main` method indicates the starting point for running this application. Once a local variable is declared, the only other action taken by this method is to output a line of text to the console.

Note that in this usage, the `Console` refers to the text-based window, which hosts a command prompt from which this program is run. The console window is best thought of as a window encapsulating the older nongraphical-style user interface whereby literally everything was driven from the command prompt. The `Console` class is automatically created when you start your application, and it supports a variety of `Read` and `Write` methods. In the preceding example, if you were to run the code from within Visual Studio's debugger, then the console window would open and close immediately. To prevent that, you include a final line in the `Main Sub`, which executes a `Read` statement so that the program continues to run while waiting for user input.

Because so many keywords have been covered, a glossary might be useful. The following table briefly summarizes most of the keywords discussed in the preceding section, and provides a short description of their meaning in Visual Basic:

Keyword	Description
`Namespace`	A collection of classes that provide related capabilities. For example, the `System.Drawing` namespace contains classes associated with graphics.
`Class`	A definition of an object. Includes properties (variables) and methods, which can be subs or functions.
`Instance`	When a class is created, the resulting object is an instance of the class's definition. Not a keyword in Visual Basic.
`Method`	A generic name for a named set of commands. In Visual Basic, both subs and functions are types of methods. Not a keyword in Visual Basic.
`Sub`	A method that contains a set of commands, allows data to be transferred as parameters, and provides scope around local variables and commands
`Function`	A method that contains a set of commands, returns a value, allows data to be transferred as parameters, and provides scope around local variables and commands
`Return`	Ends the currently executing sub or function. Combined with a return value for functions.
`Dim`	Declares and defines a new variable
`New`	Creates an instance of an object
`Nothing`	Used to indicate that a variable has no value. Equivalent to null in other languages and databases.
`Me`	A reference to the instance of the object within which a method is executing
`Console`	A type of application that relies on a command-line interface. Console applications are commonly used for simple test frames. Also refers to a command window to and from which applications can read and write text data.
`Module`	A code block that isn't a class but which can contain sub and function methods. Not frequently used for application development, but is part of a console application.

Finally, as an example of code in the sections ahead, we are going to use Visual Studio 2008 to create a simple console application. The sample code for this chapter uses a console application titled ''ProVB_C01_Types.'' To create this, start Visual Studio 2008. From the File menu, select New. In some versions of Visual Studio, you then need to select the Project menu item from within the options for New; otherwise, the New option takes you to the window shown in Figure 1-1.

Select a new console application, name it ProVB_C01_Types, as shown in Figure 1-1, and click OK to continue. Visual Studio 2008 then goes to work for you, generating a series of files to support your new project. These files, which vary by project type, are explained in more detail in subsequent chapters. The only one that matters for this example is the one entitled `Module1.vb`. Fortunately, Visual Studio automatically opens this file for you for editing, as shown in Figure 1-2.

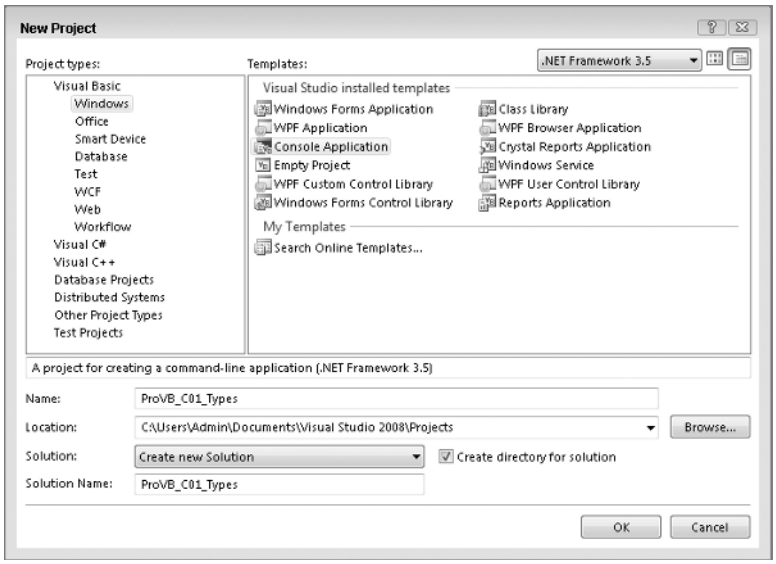

Figure 1-1

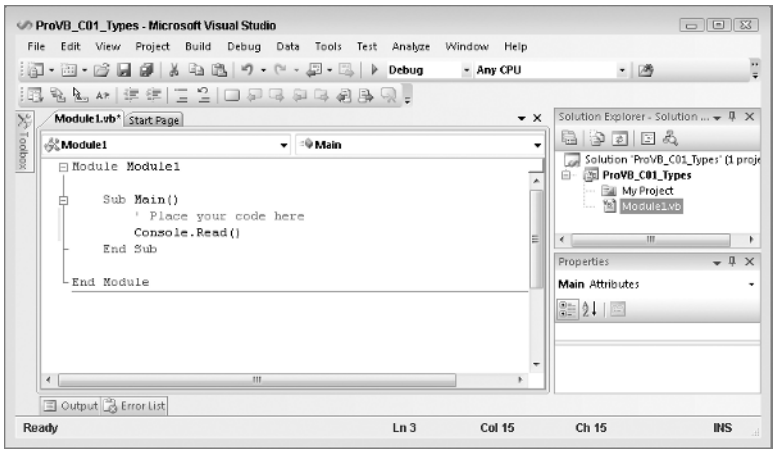

Figure 1-2

The sample shown in Figure 1-2 shows the default code with two lines of text added:

```
' Place your code here
Console.Read()
```

The first line of code represents a comment. A comment is text that is part of your source file but which the language compiler ignores. These lines enable you to describe what your code is doing, or describe what a given variable will contain. Placing a single quote on a line means that everything that follows on that line is considered to be a comment and should not be parsed as part of the Visual Basic language. This means you can have a line of code followed by a comment on the same line. Commenting code

is definitely considered a best practice, and, as discussed in Chapter 13, Visual Studio enables you to structure comments in XML so that they can be referenced for documentation purposes.

The second line of code tells the system to wait for the user to enter something in the Console window before continuing. Without this line of code, if you ran your project, you would see the project quickly run, but the window would immediately close before you could read anything. Adding this line ensures that your code entered prior to the line executes, after which the application waits until you press a key. Because this is the last line in this sample, once you press a key, the application reads what was typed and then closes, as there is no more code to execute.

To test this out, use the F5 key to compile and run your code, or use the play button shown in Figure 1-2 in the top toolbar to run your code. As you look at the sample code in this chapter, feel free to place code into this sample application and run it to see the results. More important, change it to see what happens when you introduce new conditions. As with examples in any book of this kind, the samples in this book are meant to help you get started; you can extend and modify these in any way you like.

With that in mind, now you can take a look at how data is stored and defined within your running application. While you might be thinking about permanent storage such as on your hard drive or within a database, the focus for this chapter is how data is stored in memory while your program is executing. To access data from within your program, you read and assign values to variables, and these variables take different forms or types.

Value and Reference Types

Experienced developers generally consider integers, characters, Booleans, and strings to be the basic building blocks of any language. In .NET, all objects share a logical inheritance from the base `Object` class. One of the advantages of this common heritage is the ability to rely on certain common methods of every variable. Another is that this allows all of .NET to build on a common type system. Visual Basic builds on the common type system shared across .NET languages.

Because all data types are based on the core `Object` class, every variable you dimension can be assured of having a set of common characteristics. However, this logical inheritance does not require a common physical implementation for all variables. This might seem like a conflicting set of statements, but .NET has a base type of `Object` and then allows simple structures to inherit from this base class. While everything in .NET is based on the `Object` class, under the covers .NET has two major variable types: value and reference.

For example, what most programmers see as some of the basic underlying types, such as `Integer`, `Long`, `Character`, and even `Byte`, are not implemented as classes. Thus, on the one hand, all types inherit from the `Object` class, and on the other hand, there are two core types. This is important, as you'll see when we discuss boxing and the cost of transitioning between value types and reference types. The key point is that every type, whether it is a built-in structure such as an integer or string, or a custom class such as `WroxEmployee`, does, in fact, inherit from the `Object` class. The difference between value and reference types is an underlying implementation difference:

- Value types represent simple data storage located on the stack. The stack is used for items of a known size, so items on the stack can be retrieved faster than those on the managed heap.
- Reference types are based on complex classes with implementation inheritance from their parent classes, and custom storage on the managed heap. The managed heap is optimized to support dynamic allocation of differently sized objects.

Note that the two implementations are stored in different portions of memory. As a result, value and reference types are treated differently within assignment statements, and their memory management is handled differently. It is important to understand how these differences affect the software you will write in Visual Basic. Understanding the foundations of how data is manipulated in the .NET Framework will enable you to build more reliable and better-performing applications.

Consider the difference between the stack and the heap. The stack is a comparatively small memory area in which processes and threads store data of fixed size. An integer or decimal value needs the same number of bytes to store data, regardless of the actual value. This means that the location of such variables on the stack can be efficiently determined. (When a process needs to retrieve a variable, it has to search the stack. If the stack contained variables that had dynamic memory sizes, then such a search could take a long time.)

Reference types do not have a fixed size — a string can vary in size from two bytes to nearly all the memory available on a system. The dynamic size of reference types means that the data they contain is stored on the heap, rather than the stack. However, the address of the reference type (that is, the location of the data on the heap) does have a fixed size, and thus can be (and, in fact, is) stored on the stack. By storing a reference only to a custom allocation on the stack, the program as a whole runs much more quickly, as the process can rapidly locate the data associated with a variable.

Storing the data contained in fixed and dynamically sized variables in different places results in differences in the way variables behave. Rather than limit this discussion to the most basic of types in .NET, this difference can be illustrated by comparing the behavior of the `System.Drawing.Point` structure (a value type) and the `System.Text.StringBuilder` class (a reference type).

The `Point` structure is used as part of the .NET graphics library, which is part of the `System.Drawing` namespace. The `StringBuilder` class is part of the `System.Text` namespace and is used to improve performance when you're editing strings.

First, here is an example of how the `System.Drawing.Point` structure is used:

```
Dim ptX As New System.Drawing.Point(10, 20)
Dim ptY As New System.Drawing.Point

ptY = ptX
ptX.X = 200

Console.WriteLine(ptY.ToString())
```

The output from this operation will be $\{X = 10, Y = 20\}$, which seems logical. When the code copies `ptX` into `ptY`, the data contained in `ptX` is copied into the location on the stack associated with `ptY`. Later, when the value of `ptX` changes, only the memory on the stack associated with `ptX` is altered. Altering the value of `ptX` has no effect on `ptY`. This is not the case with reference types. Consider the following code, which uses the `System.Text.StringBuilder` class:

```
Dim objX As New System.Text.StringBuilder("Hello World")
Dim objY As System.Text.StringBuilder

objY = objX
objX.Replace("World", "Test")

Console.WriteLine(objY.ToString())
```

The output from this operation will be ''Hello Test,'' not ''Hello World.'' The previous example using points demonstrated that when one value type is assigned to another, the data stored on the stack is copied. Similarly, this example demonstrates that when `objY` is assigned to `objX`, the data associated with `objX` on the stack is copied to the data associated with `objY` on the stack. However, what is copied in this case isn't the actual data, but rather the address on the managed heap where the data is actually located. This means that `objY` and `objX` now reference the same data. When the data on the heap is changed, the data associated with every variable that holds a reference to that memory is changed. This is the default behavior of reference types, and is known as a *shallow copy*. Later in this chapter, you'll see how this behavior has been overridden for strings (which perform a *deep copy*).

The differences between value types and reference types go beyond how they behave when copied, and later in this chapter you'll encounter some of the other features provided by objects. First, though, let's take a closer look at some of the most commonly used value types and learn how .NET works with them.

Primitive Types

Visual Basic, in common with other development languages, has a group of elements such as integers and strings that are termed *primitive types*. These primitive types are identified by keywords such as `String`, `Long`, and `Integer`, which are aliases for types defined by the .NET class library. This means that the line

```
Dim i As Long
```

is equivalent to the line

```
Dim i As System.Int64
```

The reason why these two different declarations are available has to do with long-term planning for your application. In most cases (such as when Visual Basic transitioned to .NET), you want to use the `Short`, `Integer`, and `Long` designations. When Visual Basic moved to .NET, the `Integer` type went from 16 bits to 32 bits. Code written with this `Integer` type would automatically use the larger value if you rewrote the code in .NET. Interestingly enough, however, the Visual Basic Migration Wizard actually recast Visual Basic 6 `Integer` values to Visual Basic .NET `Short` values.

This is the same reason why `Int16`, `Int32`, and `Int64` exist. These types specify a physical implementation; therefore, if your code is someday migrated to a version of .NET that maps the `Integer` value to `Int64`, then those values defined as `Integer` will reflect the new larger capacity, while those declared as `Int32` will not. This could be important if your code was manipulating part of an interface where changing the physical size of the value could break the interface.

The following table lists the primitive types that Visual Basic 2008 defines, and the structures or classes to which they map:

Primitive Type	.NET Class or Structure
`Byte`	`System.Byte` (structure)
`Short`	`System.Int16` (structure)
`Integer`	`System.Int32` (structure)

Primitive Type	.NET Class or Structure
`Long`	`System.Int64` (structure)
`Single`	`System.Single` (structure)
`Double`	`System.Double` (structure)
`Decimal`	`System.Decimal` (structure)
`Boolean`	`System.Boolean` (structure)
`Date`	`System.DateTime` (structure)
`Char`	`System.Char` (structure)
`String`	`System.String` (class)

The `String` *primitive type stands out from the other primitives. Strings are implemented as a class, not a structure. More important, strings are the one primitive type that is a reference type.*

You can perform certain operations on primitive types that you can't on other types. For example, you can assign a value to a primitive type using a literal:

```
Dim i As Integer = 32
Dim str As String = "Hello"
```

It's also possible to declare a primitive type as a constant using the `Const` keyword, as shown here:

```
Dim Const str As String = "Hello"
```

The value of the variable `str` in the preceding line of code cannot be changed elsewhere in the application containing this code at runtime. These two simple examples illustrate the key properties of primitive types. As noted, most primitive types are, in fact, value types. The next step is to take a look at core language commands that enable you to operate on these variables.

Commands: Conditional

Unlike many programming languages, Visual Basic has been designed to focus on readability and clarity. Many languages are willing to sacrifice these attributes to enable developers to type as little as possible. Visual Basic, conversely, is designed under the paradigm that the readability of code matters more than saving a few keystrokes, so commands in Visual Basic tend to spell out the exact context of what is being done.

Literally dozens of commands make up the Visual Basic language, so there isn't nearly enough space here to address all of them. Moreover, many of the more specialized commands are covered later in this book. However, if you are not familiar with Visual Basic or are relatively new to programming, a few would be helpful to look at here. These fall into two basic areas: *conditional statements* and *looping statements*. This chapter addresses two statements within each of these categories, starting with the conditional statements and later, after collections and arrays have been introduced, covering looping statements.

Each of these statements has the ability not only to call another method, the preferred way to manage blocks of code, but also to literally encapsulate a block of code. Note that the variables declared within the context of a conditional statement (between the `If` and `End If` lines) are only visible up until the `End If` statement. After that these variables go out of scope. The concept of scoping is discussed in more detail later in this chapter.

If Then

The conditional is one of two primary programming constructs (the other being the loop) that is present in almost every programming language. After all, even in those rare cases where the computer is just repeatedly adding values or doing some other repetitive activity, at some point a decision is needed and a condition evaluated, even if the question is only ''is it time to stop?'' Visual Basic supports the `If-Then` statement as well as the `Else` statement; and unlike some languages, the concept of an `ElseIf` statement. The `ElseIf` and `Else` statements are totally optional, and it is not only acceptable but common to encounter conditionals that do not utilize either of these code blocks. The following example illustrates a simple pair of conditions that have been set up serially:

```
If i > 1 Then
    'Code A1
ElseIf i < 1 Then
    'Code B2
Else
    'Code C3
End If
```

If the first condition is true, then code placed at marker `A1` is executed. The flow would then proceed to the `End If`, and the program would not evaluate any of the other conditions. Note that for best performance, it makes the most sense to have your most common condition first in this structure, because if it is successful, none of the other conditions need to be tested.

If the initial comparison in the preceding example code were false, then control would move to the first `Else` statement, which in this case happens to be an `ElseIf` statement. The code would therefore test the next conditional to determine whether the value of `i` were less than 1. If this were the case, then the code associated with block `B2` would be executed.

However, if the second condition were also false, then the code would proceed to the `Else` statement, which isn't concerned with any remaining condition and just executes the code in block `C3`. Not only is the `Else` optional, but even if an `ElseIf` is used, the `Else` condition is still optional. It is acceptable for the `Else` and `C3` block to be omitted from the preceding example.

Comparison Operators

There are several ways to discuss what is evaluated in an `If` statement. Essentially, the code between the `If` and `Then` portion of the statement must eventually evaluate out to a Boolean. At the most basic level, this means you can write `If True Then`, which results in a valid statement, although the code would always execute the associated block of code with that `If` statement. The idea, however, is that for a basic comparison, you take two values and place between them a comparison operator. Comparison operators include the following symbols: =, >, <, >=, <=.

Additionally, certain keywords can be used with a comparison operator. For example, the keyword `Not` can be used to indicate that the statement should consider the failure of a given comparison as a reason to execute the code encapsulated by its condition. An example of this is shown in the next example:

```
If Not i = 1 Then
    'Code A1
End If
```

It is therefore possible to compare two values and then take the resulting Boolean from this comparison and reevaluate the result. In this case, the result is only reversed, but the `If` statement supports more complex comparisons using statements such as `And` and `Or`. These statements enable you to create a complex condition based on several comparisons, as shown here:

```
If Not i = 1 Or i < 0 And str = "Hello" Then
    'Code A1
Else
    'Code B2
End If
```

The `And` and `Or` conditions are applied to determine whether the first comparison's results are true or false along with the second value's results. The `And` conditional means that both comparisons must evaluate to true in order for the `If` statement to execute the code in block `A1`, and the `Or` statement means that if the condition on either side is true, then the `If` statement can evaluate code block `A1`. However, in looking at this statement, your first reaction should be to pause and attempt to determine in exactly what order all of the associated comparisons occur.

There is a precedence. First, any numeric style comparisons are applied, followed by any unary operators such as `Not`. Finally, proceeding from left to right, each Boolean comparison of `And` and `Or` is applied. However, a much better way to write the preceding statement is to use parentheses to identify in what order you want these comparisons to occur. The first `If` statement in the following example illustrates the default order, while the second and third use parentheses to force a different priority on the evaluation of the conditions:

```
If ((Not i = 1) Or i < 0) And (str = "Hello") Then
If (Not i = 1) Or (i < 0 And str = "Hello") Then
If Not ((i = 1 Or i < 0) And str = "Hello") Then
```

All three of the preceding `If` statements are evaluating the same set of criteria, yet their results are potentially very different. It is always best practice to enclose complex conditionals within parentheses to illustrate the desired order of evaluation. Of course, these comparisons have been rather simple; you could replace the variable value in the preceding examples with a function call that might include a call to a database. In such a situation, if the desired behavior were to execute this expensive call only when necessary, then you might want to use one of the shortcut comparison operators.

Since you know that for an `And` statement both sides of the `If` statement must be true, there are times when knowing that the first condition is false could save processing time; you would not bother executing the second condition. Similarly, if the comparison involves an `Or` statement, then once the first part of the condition is true, there is no reason to evaluate the second condition because you know that the net result is success. In this case, the `AndAlso` and `OrElse` statements allow for performance optimization.

```
If ((Not i = 1) Or i < 0) AndAlso (MyFunction() = "Success") Then
If Not i = 1 OrElse (i < 0 And MyFunction() = "Success") Then
```

The preceding code illustrates that instead of using a variable like `str` as used in the preceding samples, your condition might call a function you've written that returns a value. In this case, `MyFunction` would return a string that would then be used in the comparison. Each of these conditions has therefore been optimized so that there are situations where the code associated with `MyFunction` won't be executed.

This is potentially important, not only from a performance standpoint, but also in a scenario where given the first condition your code might throw an error. For example, it's not uncommon to first determine whether a variable has been assigned a value and then to test that value. This introduces yet another pair of conditional elements: the `Is` and `IsNot` conditionals.

Using `Is` enables you to determine whether a variable has been given a value, or to determine its type. In the past it was common to see nested `If` statements as a developer first determined whether the value was null, followed by a separate `If` statement to determine whether the value was valid. Starting with .NET 2.0, the short-circuit conditionals enable you to check for a value and then check whether that value meets the desired criteria. The short-circuit operator prevents the check for a value from occurring and causing an error if the variable is undefined, so both checks can be done with a single `If` statement:

```
Dim mystring as string = Nothing
If mystring IsNot Nothing AndAlso mystring.Length > 100 Then
    'Code A1
ElseIf mystring.GetType Is GetType(Integer) Then
    'Code B2
End If
```

The preceding code would fail on the first comparison because `mystring` has only been initialized to `Nothing`, meaning that by definition it doesn't have a length. Note also that the second condition will fail because you know that `myString` isn't of type `Integer`.

Select Case

The preceding section makes it clear that the `If` statement is the king of conditionals. However, in another scenario you may have a simple condition that needs to be tested repeatedly. For example, suppose a user selects a value from a drop-down list and different code executes depending on that value. This is a relatively simple comparison, but if you have 20 values, then you would potentially need to string together 20 different `If Then` and `ElseIf` statements to account for all of the possibilities.

A cleaner way of evaluating such a condition is to leverage a `Select Case` statement. This statement was designed to test a condition, but instead of returning a Boolean value, it returns a value that is then used to determine which block of code, each defined by a `Case` statement, should be executed:

```
Select Case i
    Case 1
       'Code A1
    Case 2
       'Code B2
    Case Else
       'Code C3
End Select
```

The preceding sample code shows how the `Select` portion of the statement determines the value represented by the variable `i`. Depending on the value of this variable, the `Case` statement executes the appropriate code block. For a value of 1, the code in block `A1` is executed; similarly, a 2 results in code block `B2` executing. For any other value, because this case statement includes an `Else` block, the case statement executes the code represented by `C3`. Finally, the next example illustrates that the cases do not need to be integer values and can, in fact, even be strings:

```
Dim mystring As String = "Intro"
Select Case mystring
    Case "Intro"
        'Code A1
    Case "Exit"
        'Code A2
    Case Else
        'Code A3
End Select
```

Now that you have been introduced to these two control elements that enable you to control what happens in your code, your next step is to review details of the different variable types that are available within Visual Basic 2008, starting with the value types.

Value Types (Structures)

Value types aren't as versatile as reference types, but they can provide better performance in many circumstances. The core value types (which include the majority of primitive types) are `Boolean`, `Byte`, `Char`, `DateTime`, `Decimal`, `Double`, `Guid`, `Int16`, `Int32`, `Int64`, `SByte`, `Single`, and `TimeSpan`. These are not the only value types, but rather the subset with which most Visual Basic developers consistently work. As you've seen, value types by definition store data on the stack.

Value types can also be referred to by their proper name: structures. The underlying principles and syntax of creating custom structures mirrors that of creating classes, covered in the next chapter. This section focuses on some of the built-in types provided by the .NET Framework — in particular, the built-in types known as *primitives*.

Boolean

The .NET `Boolean` type represents true or false. Variables of this type work well with the conditional statements that were just discussed. When you declare a variable of type `Boolean`, you can use it within a conditional statement directly:

```
Dim blnTrue As Boolean = True
Dim blnFalse As Boolean = False
If blnTrue Then
   Console.WriteLine(blnTrue)
   Console.WriteLine(blnFalse.ToString)
End If
```

Always use the `True` *and* `False` *constants when working with Boolean variables.*

Unfortunately, in the past developers had a tendency to tell the system to interpret a variable created as a `Boolean` as an `Integer`. This is referred to as *implicit conversion* and is discussed later in this chapter. It is not the best practice, and when .NET was introduced, it caused issues for Visual Basic because the underlying representation of `True` in other languages wasn't going to match those of Visual Basic. The result was that Visual Basic represents `True` differently for implicit conversions than other .NET languages.

`True` has been implemented in such a way that when converted to an integer, Visual Basic converts a value of `True` to `-1` (negative one). This is one of the few (but not the only) legacy carryovers from older versions of Visual Basic and is different from other languages, which typically use the value integer value 1. Generically, all languages tend to implicitly convert `False` to 0 and `True` to a nonzero value.

However, Visual Basic works as part of a multilanguage environment, with metadata-defining interfaces, so the external value of `True` is as important as its internal value. Fortunately, Microsoft implemented Visual Basic in such a way that while `-1` is supported, the .NET standard of `1` is exposed from Visual Basic methods to other languages.

To create reusable code, it is always better to avoid implicit conversions. In the case of `Boolean`s, if the code needs to check for an integer value, then you should explicitly evaluate the `Boolean` and create an appropriate integer. The code will be far more maintainable and prone to fewer unexpected results.

Integer Types

Now that `Boolean`s have been covered in depth, the next step is to examine the `Integer` types that are part of Visual Basic. Visual Basic 6.0 included two types of integer values: The `Integer` type was limited to a maximum value of 32767, and the `Long` type supported a maximum value of 2147483647. The .NET Framework added a new integer type, the `Short`. The `Short` is the equivalent of the `Integer` value from Visual Basic 6.0; the `Integer` has been promoted to support the range previously supported by the Visual Basic 6.0 `Long` type, and the Visual Basic .NET `Long` type is an eight-byte value. The new `Long` type provides support for 64-bit values, such as those used by current 64-bit processors. In addition, each of these types also has two alternative types. In all, Visual Basic supports nine `Integer` types:

Type	Allocated Memory	Minimum Value	Maximum Value
`Short`	2 bytes	−32768	32767
`Int16`	2 bytes	−32768	32767
`UInt16`	2 bytes	0	65535
`Integer`	4 bytes	−2147483648	2147483647
`Int32`	4 bytes	−2147483648	2147483647
`UInt32`	4 bytes	0	4294967295
`Long`	8 bytes	−9223372036854775808	9223372036854775807
`Int64`	8 bytes	−9223372036854775808	9223372036854775807
`UInt64`	8 bytes	0	18446744073709551615

Short

A `Short` value is limited to the maximum value that can be stored in two bytes. This means there are 16 bits and the value can range between −32768 and 32767. This limitation may or may not be based on the amount of memory physically associated with the value; it is a definition of what must occur in the .NET Framework. This is important, because there is no guarantee that the implementation will actually use less memory than when using an `Integer` value. It is possible that in order to optimize memory or processing, the operating system will allocate the same amount of physical memory used for an `Integer` type and then just limit the possible values.

The `Short` (or `Int16`) value type can be used to map SQL `smallint` values.

Integer

An `Integer` is defined as a value that can be safely stored and transported in four bytes (not as a four-byte implementation). This gives the `Integer` and `Int32` value types a range from −2147483648 to 2147483647. This range is more than adequate to handle most tasks.

The main reason to use an `Int32` in place of an integer value is to ensure future portability with interfaces. For example, the `Integer` value in Visual Basic 6.0 was limited to a two-byte value, but is now a four-byte value. In future 64-bit platforms, the `Integer` value might be an eight-byte value. Problems could occur if an interface used a 64-bit `Integer` with an interface that expected a 32-bit `Integer` value, or, conversely, if code using the `Integer` type is suddenly passed to a variable explicitly declared as `Int32`.

The solution is to be consistent. Use `Int32`, which would remain a 32-bit value, even on a 64-bit platform, if that is what you need. In addition, as a best practice, use `Integer` so your code can be unconcerned with the underlying implementation.

The Visual Basic .NET `Integer` value type matches the size of an integer value in SQL Server, which means that you can easily align the column type of a table with the variable type in your programs.

Long

The `Long` type is aligned with the `Int64` value. `Long`s have an eight-byte range, which means that their value can range from −9223372036854775808 to 9223372036854775807. This is a big range, but if you need to add or multiply `Integer` values, then you need a large value to contain the result. It's common while doing math operations on one type of integer to use a larger type to capture the result if there's a chance that the result could exceed the limit of the types being manipulated.

The `Long` value type matches the `bigint` type in SQL.

Unsigned Types

Another way to gain additional range on the positive side of an `Integer` type is to use one of the unsigned types. The unsigned types provide a useful buffer for holding a result that might exceed an operation by a small amount, but this isn't the main reason they exist. The `UInt16` type happens to have the same characteristics as the `Character` type, while the `UInt32` type has the same characteristics as a system memory pointer on a 32-bit system.

However, never write code that attempts to leverage this relationship. Such code isn't portable, as on a 64-bit system the system memory pointer changes and uses the `UInt64` type. However, when larger

integers are needed and all values are known to be positive, these values are of use. As for the low-level uses of these types, certain low-level drivers use this type of knowledge to interface with software that expects these values and they are the underlying implementation for other value types. This is why, when you move from a 32-bit system to a 64-bit system, you need new drivers for your devices, and why applications shouldn't leverage this same type of logic.

Decimal Types

Just as there are several types to store integer values, there are three implementations of value types to store real number values. The `Single` and `Double` types work the same way in Visual Basic .NET as they did in Visual Basic 6.0. The difference is the Visual Basic 6.0 `Currency` type (which was a specialized version of a `Double` type), which is now obsolete; it was replaced by the `Decimal` value type for very large real numbers.

Type	Allocated Memory	Negative Range	Positive Range
`Single`	4 bytes	−3.402823E38 to −1.401298E-45	1.401298E-45 to 3.402823E38
`Double`	8 bytes	−1.79769313486231E308 to −4.94065645841247E-324	4.94065645841247E-324 to 1.79769313486232E308
`Currency`	Obsolete	—	—
`Decimal`	16 bytes	−79228162514264 337593543950335 to 0.00000000000000 00000000000001	0.00000000000000 00000000000001 to 79228162514643 37593543950335

Single

The `Single` type contains four bytes of data, and its precision can range anywhere from 1.401298E-45 to 3.402823E38 for positive values and from −3.402823E38 to −1.401298E-45 for negative values.

It can seem strange that a value stored using four bytes (the same as the `Integer` type) can store a number that is larger than even the `Long` type. This is possible because of the way in which numbers are stored; a real number can be stored with different levels of precision. Note that there are six digits after the decimal point in the definition of the `Single` type. When a real number gets very large or very small, the stored value is limited by its significant places.

Because real values contain fewer significant places than their maximum value, when working near the extremes it is possible to lose precision. For example, while it is possible to represent a `Long` with the value of 9223372036854775805, the `Single` type rounds this value to 9.223372E18. This seems like a reasonable action to take, but it isn't a reversible action. The following code demonstrates how this loss of precision and data can result in errors:

```
Dim l As Long
Dim s As Single

l = Long.MaxValue
Console.WriteLine(l)
```

```
s = Convert.ToSingle(l)
Console.WriteLine(s)
s -= 1000000000000
l = Convert.ToInt64(s)

Console.WriteLine(l)
Console.WriteLine(Long.MaxValue - l)
```

I placed this code into the simple console application created earlier in this chapter and ran it. The code creates a `Long` that has the maximum value possible, and outputs this value. Then it converts this value to a `Single` and outputs it in that format. Next, the value 1000000000000 is subtracted to the `Single` using the `-=` syntax, which is similar to writing `s = s – 1000000000000`. Finally, the code assigns the `Single` value back into the `Long` and then outputs both the `Long` and the difference between the original value and the new value. The results are shown in Figure 1-3. The results probably aren't consistent with what you might expect.

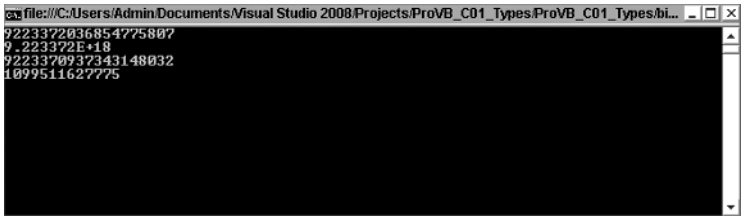

Figure 1-3

The first thing to notice is how the values are represented in the output based on type. The `Single` value actually uses an exponential display instead of displaying all of the significant digits. More important, as you can see, the result of what is stored in the `Single` after the math operation actually occurs is not accurate in relation to what is computed using the `Long` value. Therefore, both the `Single` and `Double` types have limitations in accuracy when you are doing math operations. These accuracy issues are because of limitations in what is stored and how binary numbers represent decimal numbers. To better address these issues for large numbers, .NET provides the `decimal` type.

Double

The behavior of the previous example changes if you replace the value type of `Single` with `Double`. A `Double` uses eight bytes to store values, and as a result has greater precision and range. The range for a `Double` is from 4.94065645841247E-324 to 1.79769313486232E308 for positive values and from −1.79769313486231E308 to −4.94065645841247E-324 for negative values. The precision has increased such that a number can contain 15 digits before the rounding begins. This greater level of precision makes the `Double` value type a much more reliable variable for use in math operations. It's possible to represent most operations with complete accuracy with this value. To test this, change the sample code from the previous section so that instead of declaring the variable `s` as a `Single` you declare it as a `Double` and rerun the code. Don't forget to also change the conversion line from `ToSingle` to `ToDouble`. Did you get an accurate difference? Or was yours like mine — the difference in value off by 1?

Decimal

The `Decimal` type is a hybrid that consists of a 12-byte integer value combined with two additional 16-bit values that control the location of the decimal point and the sign of the overall value. A `Decimal` value consumes 16 bytes in total and can store a maximum value of 79228162514264337593543950335.

This value can then be manipulated by adjusting where the decimal place is located. For example, the maximum value while accounting for four decimal places is 792281625142643375935439 5.0335. This is because a `Decimal` isn't stored as a traditional number, but as a 12-byte integer value, with the location of the decimal in relation to the available 28 digits. This means that a `Decimal` does not inherently round numbers the way a `Double` does.

As a result of the way values are stored, the closest precision to zero that a `Decimal` supports is 0.0000000000000000000000000001. The location of the decimal point is stored separately; and the `Decimal` type stores a value that indicates whether its value is positive or negative separately from the actual value. This means that the positive and negative ranges are exactly the same, regardless of the number of decimal places.

Thus, the system makes a trade-off whereby the need to store a larger number of decimal places reduces the maximum value that can be kept at that level of precision. This trade-off makes a lot of sense. After all, it's not often that you need to store a number with 15 digits on both sides of the decimal point, and for those cases you can create a custom class that manages the logic and leverages one or more decimal values as its properties. You'll find that if you again modify and rerun the sample code you've been using in the last couple of sections that converts to and from `Long` values by using `Decimal`s for the interim value and conversion, now your results are completely accurate.

Char and Byte

The default character set under Visual Basic is Unicode. Therefore, when a variable is declared as type `Char`, Visual Basic creates a two-byte value, since, by default, all characters in the Unicode character set require two bytes. Visual Basic supports the declaration of a character value in three ways. Placing a `$c$` following a literal string informs the compiler that the value should be treated as a character, or the `Chr` and `ChrW` methods can be used. The following code snippet shows that all three of these options work similarly, with the difference between the `Chr` and `ChrW` methods being the range of available valid input values. The `ChrW` method allows for a broader range of values based on wide character input.

```
Dim chrLtr_a As Char = "a"c
Dim chrAsc_a As Char = Chr(97)
Dim chrAsc_b as Char = ChrW(98)
```

To convert characters into a string suitable for an ASCII interface, the runtime library needs to validate each character's value to ensure that it is within a valid range. This could have a performance impact for certain serial arrays. Fortunately, Visual Basic supports the `Byte` value type. This type contains a value between `0` and `255` that exactly matches the range of the ASCII character set. When interfacing with a system that uses ASCII, it is best to use a `Byte` array. The runtime knows there is no need to perform a Unicode-to-ASCII conversion for a `Byte` array, so the interface between the systems operates significantly faster.

In Visual Basic, the `Byte` value type expects a numeric value. Thus, to assign the letter ''a'' to a `Byte`, you must use the appropriate character code. One option to get the numeric value of a letter is to use the `Asc` method, as shown here:

```
Dim bytLtrA as Byte = Asc("a")
```

DateTime

The Visual Basic `Date` keyword has always supported a structure of both date and time. You can, in fact, declare date values using both the `DateTime` and `Date` types. Note that internally Visual Basic no longer stores a date value as a `Double`; however, it provides key methods for converting the current internal date representation to the legacy `Double` type. The `ToOADate` and `FromOADate` methods support backward compatibility during migration from previous versions of Visual Basic.

Visual Basic also provides a set of shared methods that provides some common dates. The concept of shared methods is described in more detail in the next chapter, which covers object syntax, but, in short, shared methods are available even when you don't create an instance of a class. For the `DateTime` structure, the `Now` method returns a `Date` value with the local date and time. This method has not been changed from Visual Basic 6.0, but the `Today` and `UtcNow` methods have been added. These methods can be used to initialize a `Date` object with the current local date, or the date and time based on Universal Coordinated Time (also known as Greenwich Mean Time), respectively. You can use these shared methods to initialize your classes, as shown in the following code sample:

```
Dim dteNow as Date = Now()
Dim dteToday as Date = Today()
Dim dteGMT as DateTime = DateTime.UtcNow()
Dim dteString As Date = #4/16/1965#
Console.WriteLine(dteString.ToLongDateString)
```

The last two lines in the preceding example just demonstrate the use of `Date` as a primitive. As noted earlier, primitive values enable you to assign them directly within your code, but many developers seem unaware of the format for doing this with dates.

Reference Types (Classes)

A lot of the power of Visual Basic is harnessed in objects. An object is defined by its class, which describes what data, methods, and other attributes an instance of that class supports. Thousands of classes are provided in the .NET Framework class library.

When code instantiates an object from a class, the object created is a reference type. Recall that the data contained in value and reference types is stored in different locations, but this is not the only difference between them. A class (which is the typical way to refer to a reference type) has additional capabilities, such as support for protected methods and properties, enhanced event-handling capabilities, constructors, and finalizers, and can be extended with a custom base class via inheritance. Classes can also be used to define how operators such as ''='' and ''+'' work on an instance of the class.

The intention of this chapter is to introduce you to some commonly used classes, and to complement your knowledge of the common value types already covered. Chapters 2 and 3 contain a detailed look at object orientation in Visual Basic. This chapter examines the features of the `Object`, `String`, `DBNull`, and `Array` classes, as well as the `Collection` classes found in the `System.Collections` namespace.

The Object Class

The `Object` class is the base class for every type in .NET, both value and reference types. At its core, every variable is an object and can be treated as such. You can think of the `Object` class (in some ways) as the

replacement for the `Variant` type found in COM and COM-based versions of Visual Basic, but take care. COM, a `Variant` type, represents a variant memory location; in Visual Basic, an `Object` type represents a reference to an instance of the `Object` class. In COM, a `Variant` is implemented to provide a reference to a memory area on the heap, but its definition doesn't define any specific ways of accessing this data area. As you'll see during this look at objects, an instance of an ''object'' includes all the information needed to define the actual type of that object.

Because the `Object` class is the basis of all types, you can assign any variable to an object. Reference types maintain their current reference and implementation but are generically handled, whereas value types are packaged into a box and placed into the memory location associated with the `Object`. For example, there are instance methods that are available on `Object`, such as `ToString`. This method, if implemented, returns a string representation of an instance value. Because the `Object` class defines it, it can be called on any object:

```
Dim objMyClass as New MyClass("Hello World")

Console.WriteLine(objMyClass.ToString)
```

This brings up the question of how the `Object` class knows how to convert custom classes to `String` objects. The answer is that it doesn't. In order for this method to actually return the data in an instance of a `String`, a class must override this method. Otherwise, when this code is run, the default version of this method defined at the `Object` level returns the name of the current class (`MyClass`) as its string representation. This section will be clearer after you read Chapter 2. The key point is that if you create an implementation of `ToString` in your class definition, then even when an instance of your object is cast to the type `Object`, your custom method will still be called. The following snippet shows how to create a generic object under the `Option Strict` syntax:

```
Dim objVar as Object

objVar = Me

CType(objVar, Form).Text = "New Dialog Title Text"
```

That `Object` is then assigned a copy of the current instance of a Visual Basic form. In order to access the `Text` property of the original `Form` class, the `Object` must be cast from its declared type of `Object` to its actual type (`Form`), which supports the `Text` property. The `CType` command (covered later) accepts the object as its first parameter, and the class name (without quotes) as its second parameter. In this case, the current instance variable is of type `Form`; and by casting this variable, the code can reference the `Text` property of the current form.

The String Class

Another class that plays a large role in most development projects is the `String` class. Having `String`s defined as a class is more powerful than the Visual Basic 6.0 data type of `String` that you may be more familiar with. The `String` class is a special class within .NET because it is the one primitive type that is not a value type. To make `String` objects compatible with some of the underlying behavior in .NET, they have some interesting characteristics.

These methods are shared, which means that the methods are not specific to any instance of a `String`. The `String` class also contains several other methods that are called based on an instance of a specific

`String` object. The methods on the `String` class replace the functions that Visual Basic 6.0 had as part of the language for string manipulation, and they perform operations such as inserting strings, splitting strings, and searching strings.

String()

The `String` class has several different constructors for those situations in which you aren't simply assigning an existing value to a new string. The term *constructor* is expanded upon in Chapter 2. Constructors are methods that are used to construct an instance of a class. `String()` would be the default constructor for the `String` class, but the `String` class does not expose this constructor publicly. The following example shows the most common method of creating a `String`:

```
Dim strConstant as String = "ABC"
Dim strRepeat as New String("A"c, 20)
```

A variable is declared of type `String` and as a primitive is assigned the value `'ABC'`. The second declaration uses one of the parameterized versions of the `String` constructor. This constructor accepts two parameters: The first is a character and the second is the number of times that character should be repeated in the string.

In addition to creating an instance of a string and then calling methods on your variable, the `String` class has several shared methods. A shared method refers to a method on a class that does not require an instance of that class. Shared methods are covered in more detail in relation to objects in Chapter 2; for the purpose of this chapter, the point is that you can reference the class `String` followed by a ''.'' and see a list of shared methods for that class. For strings, this list includes the following:

Shared Methods	Description
`Empty`	This is actually a property. It can be used when an empty `String` is required. It can be used for comparison or initialization of a `String`.
`Compare`	Compares two objects of type `String`
`CompareOrdinal`	Compares two `String`s, without considering the local national language or culture
`Concat`	Concatenates one or more `String`s
`Copy`	Creates a new `String` with the same value as an instance provided
`Equals`	Determines whether two `String`s have the same value
`IsNullorEmpty`	This shared method is a very efficient way of determining whether a given variable has been set to the empty string or `Nothing`.

Not only have creation methods been encapsulated, but other string-specific methods, such as character and substring searching, and case changes, are now available from `String` objects instances.

The SubString Method

The .NET `String` class has a method called `SubString`. Thanks to overloading, covered in Chapter 2, there are two versions of this method: The first accepts a starting position and the number of characters

to retrieve, while the second accepts simply the starting location. The following code shows examples of using both of these methods on an instance of a `String`:

```
Dim strMyString as String = "Hello World"

Console.WriteLine(strMystring.SubString(0,5))
Console.WriteLine(strMyString.SubString(6))
```

The PadLeft and PadRight Methods

These methods enable you to justify a `String` so that it is left- or right-justified. As with `SubString`, the `PadLeft` and `PadRight` methods are overloaded. The first version of these methods requires only a maximum length of the `String`, and then uses spaces to pad the `String`. The other version requires two parameters: the length of the returned `String` and the character that should be used to pad the original `String`. An example of working with the `PadLeft` method is as follows:

```
Dim strMyString as String = "Hello World"

Console.WriteLine(strMyString.PadLeft(30))
Console.WriteLine(strMyString.PadLeft(20,"."c))
```

The String.Split Method

This instance method on a string is designed to enable you to take a string and separate out that string into an array of components. For example, if you want to quickly find each of the different elements in a comma-delimited string, you could use the `Split` method to turn the string into an array of smaller strings, each of which contains one field of data. In the following example, the `stringarray` variable will contain an array of three elements:

```
Dim strMyString As String = "Column1, Col2, Col3"
Dim stringarray As String() = strMyString.Split(","c)
Console.WriteLine(stringarray.Count.ToString())
```

The String Class Is Immutable

The Visual Basic `String` class isn't entirely different from the `String` type that VB programmers have used for years. The majority of string behaviors remain unchanged, and the majority of methods are now available as classes. However, to support the default behavior that people associate with the `String` primitive type, the `String` class isn't declared in the same way as several other classes. Strings in .NET do not allow editing of their data. When a portion of a string is changed or copied, the operating system allocates a new memory location and copies the resulting string to this new location. This ensures that when a string is copied to a second variable, the new variable references its own copy.

To support this behavior in .NET, the `String` class is defined as an *immutable class*. This means that each time a change is made to the data associated with a string, a new instance is created, and the original referenced memory is released for garbage collection. This is an expensive operation, but the result is that the `String` class behaves as people expect a primitive type to behave. Additionally, when a copy of a string is made, the `String` class forces a new version of the data into the referenced memory. This ensures that each instance of a string references only its own memory. Consider the following code:

```
Dim strMyString as String
Dim intLoop as Integer
```

```
For intLoop = 1 to 1000
   strMyString = strMyString & "A very long string "
Next
Console.WriteLine(strMyString)
```

This code does not perform well. For each assignment operation on the `strMyString` variable, the system allocates a new memory buffer based on the size of the new string, and copies both the current value of `strMyString` and the new text that is to be appended. The system then frees the previous memory that must be reclaimed by the garbage collector. As this loop continues, the new memory allocation requires a larger chunk of memory. Therefore, operations such as this can take a long time. However, .NET offers an alternative in the `System.Text.StringBuilder` object, shown in the following example:

```
Dim objMyStrBldr as New System.Text.StringBuilder()
Dim intLoop as Integer

For intLoop = 1 to 1000
   ObjMyStrBldr.Append("A very long string ")
Next
Console.WriteLine(objMyStrBldr.ToString())
```

The preceding code works with strings but does not use the `String` class. The .NET class library contains a class called `System.Text.StringBuilder`, which performs better when strings will be edited repeatedly. This class does not store strings in the conventional manner; it stores them as individual characters, with code in place to manage the ordering of those characters. Thus, editing or appending more characters does not involve allocating new memory for the entire string. Because the preceding code snippet does not need to reallocate the memory used for the entire string, each time another set of characters is appended it performs significantly faster. Ultimately, an instance of the `String` class is never explicitly needed because the `StringBuilder` class implements the `ToString` method to roll up all of the characters into a string. While the concept of the `StringBuilder` class isn't new, because it is now available as part of the Visual Basic implementation, developers no longer need to create their own string memory managers.

String Constants

If you ever have to produce output based on a string you'll quickly find yourself needing to embed certain constant values. For example, it's always useful to be able to add a carriage-return linefeed combination to trigger a new line in a message box. One way to do this is to learn the underlying ASCII codes and then embed these control characters directly into your string or string-builder object.

Visual Basic provides an easier solution for working with these: the `Microsoft.VisualBasic.Constants` class. The `Constants` class, which you can tell by its namespace is specific to Visual Basic, contains definitions for several standard string values that you might want to embed. The most common, of course, is `Constants.VbCrLf`, which represents the carriage-return linefeed combination. Feel free to explore this class for additional constants that you might need to manipulate string output.

The DBNull Class and IsDBNull Function

When working with a database, a value for a given column may not be defined. For a reference type this isn't a problem, as it is possible to set reference types to `Nothing`. However, for value types, it is necessary to determine whether a given column from the database or other source has an actual value.

The first way to manage this task is to leverage the `DBNull` call and the `IsDBNull` function. The `IsDBNull` function accepts an object as its parameter and returns a `Boolean` that indicates whether the variable has been initialized.

In addition to this method, Visual Basic has access to the `DBNull` class. This class is part of the `System` namespace, and to use it you declare a local variable with the `DBNull` type. This variable is then used with an `is` comparison operator to determine whether a given variable has been initialized:

```
Dim sysNull As System.DBNull = System.DBNull.Value
Dim strMyString As String = Nothing

If strMyString Is sysNull Then
  strMyString = "Initialize my String"
End If
If Not IsDBNull(strMyString) Then
  Console.WriteLine(strMyString)
End If
```

In this code, the `strMyString` variable is declared and initialized to `Nothing`. The first conditional is evaluated to `True`, and as a result the string is initialized. The second conditional then ensures that the declared variable has been initialized. Because this was accomplished in the preceding code, this condition is also `True`. In both cases, the `sysNull` value is used not to verify the type of the object, but to verify that it has not yet been instantiated with a value.

Nullable Types

In addition to having the option to explicitly check for the `DBNull` value, Visual Basic 2005 introduced the capability to create a nullable value type. In the background, when this syntax is used, the system creates a reference type containing the same data that would be used by the value type. Your code can then check the value of the nullable type before attempting to set this into a value type variable. Nullable types are built using generics, discussed in Chapter 6.

For consistency, however, let's take a look at how nullable types work. The key, of course, is that value types can't be set to null. This is why nullable types aren't value types. The following statement shows how to declare a nullable integer:

```
Dim intValue as Nullable(Of Integer)
```

The `intValue` variable acts like an integer, but isn't actually an integer. As noted, the syntax is based on generics, but essentially you have just declared an object of type `Nullable` and declared that this object will, in fact, hold integer data. Thus, both of the following assignment statements are valid:

```
intValue = 123
intValue = Nothing
```

However, at some point you are going to need to pass `intValue` to a method as a parameter, or set some property on an object that is looking for an object of type `Integer`. Because `intValue` is actually of type `Nullable`, it has the properties of a nullable object. The `nullable` class has two properties of interest when you want to get the underlying value. The first is the property `value`. This represents the underlying value type associated with this object. In an ideal scenario, you would just use the `value` property of the nullable object in order to assign to your actual value a type of `integer` and everything

would work. If the `intValue.value` wasn't assigned, you would get the same value as if you had just declared a new `Integer` without assigning it a value.

Unfortunately, that's not how the nullable type works. If the `intValue.value` property contains `Nothing` and you attempt to assign it, then it throws an exception. To avoid getting this exception, you always need to check the other property of the nullable type: `HasValue`. The `HasValue` property is a Boolean that indicates whether a value exists; if one does not, then you shouldn't reference the underlying value. The following code example shows how to safely use a nullable type:

```
Dim int as Integer
If intValue.HasValue Then
    int = intValue.Value
End If
```

Of course, you could add an `Else` statement to the preceding and use either `Integer.MinValue` or `Integer.MaxValue` as an indicator that the original value was `Nothing`. The key point here is that nullable types enable you to easily work with nullable columns in your database, but you must still verify whether an actual value or null was returned.

Arrays

It is possible to declare any type as an array of that type. Because an array is a modifier of another type, the basic `Array` class is never explicitly declared for a variable's type. The `System.Array` class that serves as the base for all arrays is defined such that it cannot be created, but must be inherited. As a result, to create an `Integer` array, a set of parentheses is added to the declaration of the variable. These parentheses indicate that the system should create an array of the type specified. The parentheses used in the declaration may be empty or may contain the size of the array. An array can be defined as having a single dimension using a single number, or as having multiple dimensions.

All .NET arrays at an index of zero have a defined number of elements. However, the way an array is declared in Visual Basic varies slightly from other .NET languages such as C#. Back when the first .NET version of Visual Basic was announced, it was also announced that arrays would always begin at `0` and that they would be defined based on the number of elements in the array. In other words, Visual Basic would work the same way as the other initial .NET languages. However, in older versions of Visual Basic, it is possible to specify that an array should start at 1 instead of 0. This meant that a lot of existing code didn't define arrays based on their upper limit. To resolve this issue, the engineers at Microsoft decided on a compromise: All arrays in .NET begin at `0`, but when an array is declared in Visual Basic, the definition is based on the upper limit of the array, not the number of elements.

The main result of this upper-limit declaration is that arrays defined in Visual Basic have one more entry by definition than those defined with other .NET languages. Note that it's still possible to declare an array in Visual Basic and reference it in C# or another .NET language. The following code illustrates some simple examples that demonstrate five different ways to create arrays, using a simple integer array as the basis for the comparison:

```
Dim arrMyIntArray1(20) as Integer
Dim arrMyIntArray2() as Integer = {1, 2, 3, 4}
Dim arrMyIntArray3(4,2) as Integer
Dim arrMyIntArray4( , ) as Integer = _
    { {1, 2, 3},{4, 5, 6}, {7, 8, 9},{10, 11, 12},{13, 14 , 15} }
Dim arrMyIntArray5() as Integer
```

In the first case, the code defines an array of integers that spans from `arrMyIntArray1(0)` to `arrMyIntArray1(20)`. This is a 21-element array, because all arrays start at `0` and end with the value defined in the declaration as the upper bound. The second statement creates an array with four elements numbered `0` through `3`, containing the values `1` to `4`. The third statement creates a multidimensional array containing five elements at the first level, with each of those elements containing three child elements. The challenge is to remember that all subscripts go from `0` to the upper bound, meaning that each array contains one more element than its upper bound. The result is an array with 15 elements. The next line of code, the fourth, shows an alternative way of creating the same array, but in this case there are four elements, each containing four elements, with subscripts from `0` to `3` at each level. Finally, the last line demonstrates that it is possible to simply declare a variable and indicate that the variable is an array, without specifying the number of elements in the array.

Multidimensional Arrays

As shown earlier in the sample array declarations, the definition of `arrMyIntArray3` is a multi-dimensional array. This declaration creates an array with 15 elements (five in the first range, each containing three elements) ranging from `arrMyIntArray3(0,0)` through `arrMyIntArray3(2,1)` to `arrMyIntArray3(4,2)`. As with all elements of an array, when it is created without specific values, the value of each of these elements is created with the default value for that type. This case also demonstrates that the size of the different dimensions can vary. It is possible to nest deeper than two levels, but this should be done with care because such code is difficult to maintain.

The fourth declaration shown previously creates `arrMyIntArray4(, )` with predefined values. The values are mapped based on the outer set being the first dimension and the inner values being associated with the next inner dimension. For example, the value of `arrMyIntArray4(0,1)` is `2`, while the value of `arrMyIntArray4(2,3)` is `12`. The following code snippet illustrates this using a set of nested loops to traverse the array. It also provides an example of calling the `UBound` method with a second parameter to specify that you are interested in the upper bound for the second dimension of the array:

```
Dim intLoop1 as Integer
Dim intLoop2 as Integer
For intLoop1 = 0 to UBound(arrMyIntArray4)
  For intLoop2 = 0 to UBound(arrMyIntArray4, 2)
    Console.WriteLine arrMyIntArray4(intLoop1, intLoop2).ToString
  Next
Next
```

The UBound Function

Continuing to reference the arrays defined earlier, the declaration of `arrMyIntArray2` actually defined an array that spans from `arrMyIntArray2(0)` to `arrMyIntArray2(3)`. This is the case because when you declare an array by specifying the set of values, it still starts at `0`. However, in this case you are not specifying the upper bound, but rather initializing the array with a set of values. If this set of values came from a database or other source, then it might not be clear what the upper limit on the array was. To verify the upper bound of an array, a call can be made to the `UBound` function:

```
Console.Writeline CStr(UBound(ArrMyIntArray2))
```

The preceding line of code retrieves the upper bound of the first dimension of the array. However, as noted in the preceding section, you can specify an array with several different dimensions. Thus, this old-style method of retrieving the upper bound carries the potential for an error of omission. The better

way to retrieve the upper bound is to use the `GetUpperBound` method. In this case, you need to tell the array which upper-bound value you want, as shown here:

```
ArrMyIntArray2.GetUpperBound(0)
```

This is the preferred method of getting an array's upper bound because it explicitly indicates which upper bound is wanted when using multidimensional arrays.

The `UBound` function has a companion called `LBound`. The `LBound` function computes the lower bound for a given array. However, as all arrays and collections in Visual Basic are `0`-based, it doesn't have much value anymore.

The ReDim Statement

The final declaration demonstrated previously is for `arrMyIntArray5()`. This is an example of an array that has not yet been instantiated. If an attempt were made to assign a value to this array, it would trigger an exception. The solution to this is to use the `ReDim` keyword. Although `ReDim` was part of Visual Basic 6.0, it has changed slightly. The first change is that code must first `Dim` an instance of the variable; it is not acceptable to declare an array using the `ReDim` statement. The second change is that code cannot change the number of dimensions in an array. For example, an array with three dimensions cannot grow to an array of four dimensions, nor can it be reduced to only two dimensions. To further extend the example code associated with arrays, consider the following, which manipulates some of the arrays previously declared. Note that the `arrMyIntArray5` declaration was repeated for this example because this variable isn't actually usable until after it is redimensioned in the following code:

```
Dim arrMyIntArray5() as Integer

' The commented statement below would compile but would cause a runtime exception.
'arrMyIntArray5(0) = 1

ReDim arrMyIntArray5(2)
ReDim arrMyIntArray3(5,4)
ReDim Preserve arrMyIntArray4(UBound(arrMyIntArray4),2)
```

The `ReDim` of `arrMyIntArray5` instantiates the elements of the array so that values can be assigned to each element. The second statement redimensions the `arrMyIntArray3` variable defined earlier. Note that it is changing the size of both the first dimension and the second dimension. While it is not possible to change the number of dimensions in an array, it is possible to resize any of an array's dimensions. This capability is required if declarations such as `Dim arrMyIntArray6( , , ,) As Integer` are to be legal.

By the way, while it is possible to repeatedly `ReDim` a variable, this type of action should ideally be done only rarely, and never within a loop. If you intend to loop through a set of entries and add entries to an array, try to determine the number of entries you'll need before entering the loop, or at a minimum `ReDim` the size of your array in chunks to improve performance.

The Preserve Keyword

The last item in the code snippet in the preceding section illustrates an additional keyword associated with redimensioning. The `Preserve` keyword indicates that the data stored in the array prior to redimensioning should be transferred to the newly created array. If this keyword is not used, then the data stored in an array is lost. Additionally, in the preceding example, the `ReDim` statement actually reduces

the second dimension of the array. While this is a perfectly legal statement, this means that even though you have specified preserving the data, the data values `4`, `8`, `12`, and `16` that were assigned in the original definition of this array will be discarded. These are lost because they were assigned in the highest index of the second array. Because `arrMyIntArray4(1,3)` is no longer valid, the value that resided at this location has been lost.

Arrays continue to be very powerful in Visual Basic, but the basic `Array` class is just that, basic. It provides a powerful framework, but it does not provide a lot of other features that would allow for more robust logic to be built into the array. To achieve more advanced features, such as sorting and dynamic allocation, the base `Array` class has been inherited by the classes that make up the `Collections` namespace.

Collections

The `Collections` namespace is part of the `System` namespace and provides a series of classes that implement advanced array features. While the capability to make an array of existing types is powerful, sometimes more power is needed in the array itself. The capability to inherently sort or dynamically add dissimilar objects in an array is provided by the classes of the `Collections` namespace. This namespace contains a specialized set of objects that can be instantiated for additional features when working with a collection of similar objects. The following table defines several of the objects that are available as part of the `System.Collections` namespace:

Class	Description
`ArrayList`	Implements an array whose size increases automatically as elements are added
`BitArray`	Manages an array of Booleans that are stored as bit values
`Hashtable`	Implements a collection of values organized by key. Sorting is done based on a hash of the key.
`Queue`	Implements a first in, first out collection
`SortedList`	Implements a collection of values with associated keys. The values are sorted by key and are accessible by key or index.
`Stack`	Implements a last in, first out collection

Each of the objects listed focuses on storing a collection of objects. This means that in addition to the special capabilities each provides, it also provides one additional capability not available to objects created based on the `Array` class. In short, because every variable in .NET is based on the `Object` class, it is possible to have a collection defined, because one of these objects contains elements that are defined with different types. This is true because each of these collection types stores an array of objects, and because all classes are of type `Object`, a string could be stored alongside an integer value. As a result, it's possible within the collection classes for the actual objects being stored to be different. Consider the following example code:

```
Dim objMyArrList As New System.Collections.ArrayList()
Dim objItem As Object
Dim intLine As Integer = 1
Dim strHello As String = "Hello"
```

```
Dim objWorld As New System.Text.StringBuilder("World")

' Add an integer value to the array list.
objMyArrList.Add(intLine)

' Add an instance of a string object
objMyArrList.Add(strHello)

' Add a single character cast as a character.
objMyArrList.Add(" "c)

' Add an object that isn't a primitive type.
objMyArrList.Add(objWorld)

' To balance the string, insert a break between the line
' and the string "Hello", by inserting a string constant.
objMyArrList.Insert(1, ". ")

For Each objItem In objMyArrList
  ' Output the values...
  Console.Write(objItem.ToString())
Next
Console.WriteLine()
For Each objItem In objMyArrList
  ' Output the types...
  Console.Write(objItem.GetType.ToString() + " : ")
Next
```

The preceding code is an example of implementing the `ArrayList` collection class. The collection classes, as this example shows, are versatile. The preceding code creates a new instance of an `ArrayList`, along with some related variables to support the demonstration. The code then shows four different types of variables being inserted into the same `ArrayList`. Next, the code inserts another value into the middle of the list. At no time has the size of the array been declared, nor has a redefinition of the array size been required.

Part of the reason for this is that the `Add` and `Insert` methods on the `ArrayList` class are defined to accept a parameter of type `Object`. This means that the `ArrayList` object can literally accept any value in .NET.

Specialized and Generic Collections

Visual Basic has additional classes available as part of the `System.Collections.Specialized` namespace. These classes tend to be oriented around a specific problem. For example, the `ListDictionary` class is designed to take advantage of the fact that while a hash table is very good at storing and retrieving a large number of items, it can be costly when there are only a few items. Similarly, the `StringCollection` and `StringDictionary` classes are defined so that when working with strings, the time spent interpreting the type of object is reduced and overall performance is improved. Each class defined in this namespace represents a specialized implementation that has been optimized for handling specific data types.

This specialization is different from the specialization provided by one of the features introduced with Visual Studio 2005 and .NET 2.0, generics. The `System.Collections.Generics` namespace contains versions of the collection classes that have been defined to support generics. The basic idea of generics is that because performance costs and reliability concerns are associated with casting to and from the object type, collections should allow you to specify what specific type they will contain. Generics not only

prevent you from paying the cost of boxing for value types but, more important, add to the capability to create type-safe code at compile time. Generics are a powerful extension to the .NET environment and are covered in detail in Chapter 6.

Commands: Looping Statements

Just as with conditional statements, it is possible in Visual Basic to loop or cycle through all of the elements in an array. The preceding examples have relied on the use of the `For` statement, which has not yet been covered. Since you've now covered both arrays and collections, it's appropriate to introduce the primary commands for working with the elements contained in those variable types. Both the `For` loop and `While` loop share similar characteristics, and which to use is often a matter of preference.

For Each and For Next

The `For` structure in Visual Basic is the primary way of managing loops. It actually has two different formats. A standard `For Next` statement enables you to set a loop control variable that can be incremented by the `For` statement and custom exit criteria from your loop. Alternatively, if you are working with a collection in which the items in the array are not indexed numerically, then it is possible to use a `For Each` loop to automatically loop through all of the items in that collection. The following code shows a typical `For Next` statement that cycles through each of the items in an array:

```
For i As Integer = 0 To 10 Step 2
    arrMyIntArray1(i) = i
Next
```

The preceding example sets the value of every other array element to its index, starting with the first item, since like all .NET collections, the collection starts at 0. The result is that items 0, 2, 4, 6, 8, 10 are set, but items 1, 3, 5, 7, 9 may not be defined because the loop doesn't address that value.

The `For Next` structure is most commonly set up to traverse an array or similar construct (for example, a data set). The control variable `i` in the preceding example must be numeric. The value can be incremented from a starting value to an ending value, which are 0 and 10, respectively, in this example. Finally, it is possible to accept the default increment of 1; or, if desired, you can add a `Step` qualifier to your command and update the control value by a value other than 1. Note that setting the value of `Step` to 0 means that your loop will theoretically loop an infinite number of times. Best practices suggest your control value should be an integer greater than 0 and not a decimal or other floating-point number.

Visual Basic provides two additional commands that can be used within the `For` loop's block to enhance performance. The first is `Exit For`; and as you might expect, this statement causes the loop to end and not continue to the end of the processing. The other is `Continue`, which tells the loop that you are finished executing code with the current control value and that it should increment the value and reenter the loop for its next iteration:

```
For i = 1 To 100 Step 2
    If arrMyIntArray1.Count <= i Then Exit For
    If i = 5 Then Continue For
    arrMyIntArray1 (i) = i - 1
Next
```

Both the `Exit For` and `Continue` keywords were used in the preceding example. Note how each uses a format of the `If-Then` structure that places the command on the same line as the `If` statement so that no `End If` statement is required. This loop exits if the control value is larger than the number of rows defined for `arrMyIntArray1`.

Next, if the control variable `i` indicates you are looking at the sixth item in the array (index of five), then this row is to be ignored but processing should continue within the loop. Keep in mind that even though the loop control variable starts at 1, the first element of the array is still at zero. The `Continue` statement indicates that the loop should return to the `For` statement and increment the associated control variable. Thus, the code does not process the next line for item six, where `i` equals `5`.

The preceding examples demonstrate that in most cases, because your loop is going to process a known collection, Visual Basic provides a command that encapsulates the management of the loop control variable. The `For Each` structure automates the counting process and enables you to quickly assign the current item from the collection so that you can act on it in your code. It is a common way to process all of the rows in a data set or most any other collection, and all of the loop control elements such as `Continue` and `Exit` are still available:

```
For Each item As Object In objMyArrList
    'Code A1
Next
```

While, Do While, and Do Until

In addition to the `For` loop, Visual Basic includes the `While` and `Do` loops, with two different versions of the `Do` loop. The first is the `Do While` loop. With a `Do While` loop, your code starts by checking for a condition; and as long as that condition is true, it executes the code contained in the `Do` loop. Optionally, instead of starting the loop by checking the `While` condition, the code can enter the loop and then check the condition at the end of the loop. The `Do Until` loop is similar to the `Do While` loop:

```
Do While blnTrue = True
     'Code A1
 Loop
```

The `Do Until` differs from the `Do While` only in that, by convention, the condition for a `Do Until` is placed after the code block, thus requiring the code in the `Do` block to execute once before the condition is checked. It bears repeating, however, that a `Do Until` block can place the `Until` condition with the `Do` statement instead of with the `Loop` statement, and a `Do While` block can similarly have its condition at the end of the loop:

```
Do
     'Code A1
Loop Until (blnTrue = True)
```

In both cases, instead of basing the loop around an array of items or a fixed number of iterations, the loop is instead instructed to continue perpetually until a condition is met. A good use for these loops involves tasks that need to repeat for as long as your application is running. Similar to the `For` loop, there are `Exit Do` and `Continue` commands that end the loop or move to the next iteration, respectively. Note that parentheses are allowed but are not required for both the `While` and the `Until` conditional expression.

The other format for creating a loop is to omit the `Do` statement and just create a `While` loop. The `While` loop works similarly to the `Do` loop, with the following differences. First, the `While` loop's endpoint is an `End While` statement instead of a loop statement. Second, the condition must be at the start of the loop with the `While` statement, similar to the `Do While`. Finally, the `While` loop has an `Exit While` statement instead of `Exit Do`, although the behavior is the same. An example is shown here:

```
While blnTrue = True
    If blnFalse Then
        blnTrue = False
    End if
    If not blnTrue Then Exit While
    System.Threading.Thread.Sleep(500)
    blnFalse = True
End While
```

The `While` loop has more in common with the `For` loop, and in those situations where someone is familiar with another language such as C++ or C#, it is more likely to be used than the older `Do-Loop` syntax that is more specific to Visual Basic.

Finally, before leaving the discussion of looping, note the potential use of endless loops. Seemingly endless, or infinite, loops play a role in application development, so it's worthwhile to illustrate how you might use one. For example, if you were writing an e-mail program, you might want to check the user's mailbox on the server every 20 seconds. You could create a `Do While` or `Do Until` loop that contains the code to open a network connection and check the server for any new mail messages to download. You would continue this process until either the application was closed or you were unable to connect to the server. When the application was asked to close, the loop's `Exit` statement would execute, thus terminating the loop. Similarly, if the code were unable to connect to the server, it might exit the current loop, alert the user, and probably start a loop that would look for network connectivity on a regular basis.

One warning with endless loops: Always include a call to `Thread.Sleep` so that the loop only executes a single iteration within a given time frame to avoid consuming too much processor time.

Boxing

Normally, when a conversion (implicit or explicit) occurs, the original value is read from its current memory location, and then the new value is assigned. For example, to convert a `Short` to a `Long`, the system reads the two bytes of `Short` data and writes them to the appropriate bytes for the `Long` variable. However, under Visual Basic, if a value type needs to be managed as an object, then the system performs an intermediate step. This intermediate step involves taking the value on the stack and copying it to the heap, a process referred to as *boxing*. As noted earlier, the `Object` class is implemented as a reference type, so the system needs to convert value types into reference types for them to be objects. This doesn't cause any problems or require any special programming, because boxing isn't something you declare or directly control, but it does affect performance.

If you're copying the data for a single value type, this is not a significant cost, but if you're processing an array that contains thousands of values, the time spent moving between a value type and a temporary reference type can be significant.

Fortunately, there are ways to limit the amount of boxing that occurs. One method that works well is to create a class based on the value type you need to work with. This might seem counterintuitive at first

because it costs more to create a class. The key is how often you reuse the data contained in the class. By repeatedly using the object to interact with other objects, you avoid creating a temporary boxed object.

Examples in two important areas will help illustrate boxing. The first involves the use of arrays. When an array is created, the portion of the class that tracks the element of the array is created as a reference object, but each element of the array is created directly. Thus, an array of integers consists of the array object and a set of integer value types. When you update one of these values with another integer value, no boxing is involved:

```
Dim arrInt(20) as Integer
Dim intMyValue as Integer = 1

arrInt(0) = 0
arrInt(1) = intMyValue
```

Neither of these assignments of an integer value into the integer array that was defined previously requires boxing. In each case, the array object identifies which value on the stack needs to be referenced, and the value is assigned to that value type. The point here is that just because you have referenced an object doesn't mean you are going to box a value. The boxing occurs only when the values being assigned are being transitioned from value types to reference types:

```
Dim strBldr as New System.Text.StringBuilder()
Dim mySortedList as New System.Collections.SortedList()
Dim count as Integer
For count = 1 to 100
  strBldr.Append(count)
  mySortedList.Add(count, count)
Next
```

The preceding snippet illustrates two separate calls to object interfaces. One call requires boxing of the value `intCount`, while the other does not. Nothing in the code indicates which call is which, but the `Append` method of `StringBuilder` has been overridden to include a version that accepts an integer, while the `Add` method of the `SortedList` collection expects two objects. Although the integer values can be recognized by the system as objects, doing so requires the runtime library to box these values so that they can be added to the sorted list.

The key to boxing isn't that you are working with objects as part of an action, but that you are passing a value to a parameter that expects an object, or you are taking an object and converting it to a value type. However, boxing does not occur when you call a method on a value type. There is no conversion to an object, so if you need to assign an integer to a string using the `ToString` method, there is no boxing of the integer value as part of the creation of the string. Conversely, you are explicitly creating a new string object, so the cost is similar.

Parameter Passing

When an object's methods or an assembly's procedures and methods are called, it's often appropriate to provide input for the data to be operated on by the code. The values are referred to as *parameters*, and any object can be passed as a parameter to a `Function` or `Sub`.

When passing parameters, be aware of whether the parameter is being passed ''by value'' (`ByVal`) or ''by reference'' (`ByRef`). Passing a parameter by value means that if the value of that variable is changed, then

when the `Function/Sub` returns, the system automatically restores that variable to the value it had before the call. Passing a parameter by reference means that if changes are made to the value of a variable, then these changes affect the actual variable and, therefore, are still present when the variable returns.

This is where it gets a little challenging for new Visual Basic developers. Under .NET, passing a parameter by value indicates only how the top-level reference (the portion of the variable on the stack) for that object is passed. Sometimes referred to as a *shallow copy operation*, the system copies only the top-level reference value for an object passed by value. This is important to remember because it means that referenced memory is not protected. When you pass an integer by value, if the program changes the value of the integer, then your original value is restored. Conversely, if you pass a reference type, then only the location of your referenced memory is protected, not the data located within that memory location. Thus, while the reference passed as part of the parameter remains unchanged for the calling method, the actual values stored in referenced objects can be updated even when an object is passed by value.

In addition to mandatory parameters, which must be passed with a call to a given function, it is possible to declare optional parameters. Optional parameters can be omitted by the calling code. This way, it is possible to call a method such as `PadRight`, passing either a single parameter defining the length of the string and using a default of space for the padding character, or with two parameters, the first still defining the length of the string but the second now replacing the default of space with a dash.

```
Public Function PadRight(ByVal intSize as Integer, _
                         Optional ByVal chrPad as Char = " "c)
End Function
```

To use default parameters, it is necessary to make them the last parameters in the function declaration. Visual Basic also requires that every optional parameter have a default value. It is not acceptable to merely declare a parameter and assign it the `Optional` keyword. In Visual Basic, the `Optional` keyword must be accompanied by a value that is assigned if the parameter is not passed in.

ParamArray

In addition to passing explicit parameters, it is also possible to tell .NET that you would like to allow a user to pass any number of parameters of the same type. This is called a *parameter array*, and it enables a user to pass as many instances of a given parameter as are appropriate. For example, the following code creates a function `Add`, which allows a user to pass an array of integers and get the sum of these integers:

```
Public Function Add(ByVal ParamArray values() As Integer) As Long
    Dim result As Long = 0
    For Each value As Integer In values
        result += value
    Next
    Return result
End Function
```

The preceding code illustrates a function (first shown at the beginning of this chapter without its implementation) that accepts an array of integers. Notice that the `ParamArray` qualifier is preceded by a `ByVal` qualifier for this parameter. The `ParamArray` requires that the associated parameters be passed by value; they cannot be optional parameters.

You might think this looks like a standard parameter passed by value except that it's an array, but there is more to it than that. In fact, the power of the `ParamArray` derives from how it can be called, which also explains many of its limitations. The following code shows one way this method can be called:

```
Dim int1 as Integer = 2
Dim int2 as Integer = 3
Dim sum as Long = Add(1, int1, int2)
```

Notice that the preceding line, which calls this `Add` function, doesn't pass an array of integers; instead, it passes three distinct integer values. The `ParamArray` keyword tells Visual Basic to automatically join these three distinct values into an array for use within this method. However, the following lines also represent an acceptable way to call this method, by passing an actual array of values:

```
Dim myIntArray() as Integer = {1, 2, 3, 4}
Dim sum as Long = Add(myIntArray)
```

Finally, note one last limitation on the `ParamArray` keyword: It can only be used on the last parameter defined for a given method. Because Visual Basic is grabbing an unlimited number of input values to create the array, there is no way to indicate the end of this array, so it must be the final parameter.

Variable Scope

The concept of variable scope encapsulates two key elements. In all the discussion so far of variables, we have not focused on the allocation and deallocation of those variables from memory. The first allocation challenge is related to what happens when you declare two variables with the same name but at different locations in the code. For example, suppose a class declares a variable called `myObj` that holds a property for that class. Then, within one of that class's methods, you declare a different variable also named `myObj`. What will happen in that method? *Scope* defines the lifetime and precedence of every variable you declare, and it handles this question.

Similarly, there is question of the removal of variables that you are no longer using, so you can free up memory. Chapter 4 covers the collection of variables and memory once it is no longer needed by an application, so this discussion focuses on priority, with the understanding that when a variable is no longer ''in scope,'' it is available to the garbage collector for cleanup.

.NET essentially defines four levels of variable scope. The outermost scope is *global*. Essentially, just as your source code defines classes, it can also declare variables that exist the entire time that your application runs. These variables have the longest lifetime because they exist as long as your application is executing. Conversely, these variables have the lowest precedence. Thus, if within a class or method you declare another variable with the same name, then the variable with the smaller, more local scope is used before the global version.

After global scope, the next scope is at the *class* or *module* level. When you add properties to a class, you are creating variables that will be created with each instance of that class. The methods of that class will then reference those member variables from the class, before looking for any global variables. Note that because these variables are defined within a class, they are only visible to methods within that class. The scope and lifetime of these variables is limited by the lifetime of that class, and when the class is removed from the system, so are those variables. More important, those variables declared in one instance of a class are not visible in other classes or in other instances of the same class (unless you actively expose them, in

which case the object instance is used to fully qualify a reference to them; this concept is explored further in Chapter 2).

The next shorter lifetime and smaller scope is that of method variables. When you declare a new variable within a method, such variables, as well as those declared as parameters, are only visible to code that exists within that module. Thus, the method `Add` wouldn't see or use variables declared in the method `Subtract` in the same class.

Finally, within a given method are various commands that can encapsulate a block of code (mentioned earlier in this chapter). Commands such as `If Then` and `For Each` create blocks of code within a method, and it is possible within this block of code to declare new variables. These variables then have a scope of only that block of code. Thus, variables declared within an `If Then` block or a `For` loop only exist within the constraints of the `If` block or execution of the loop. Creating variables in a `For` loop is a known performance mistake and should be avoided.

Data Type Conversions

So far, this chapter has focused primarily on individual variables; but when developing software, it is often necessary to take a numeric value and convert it to a string to display in a text box. Similarly, it is often necessary to accept input from a text box and convert this input to a numeric value. These conversions, unlike some, can be done in one of two fashions: *implicitly* or *explicitly*.

Implicit conversions are those that rely on the system taking the data at runtime and adjusting it to the new type without any guidance. Often, Visual Basic's default settings enable developers to write code containing many implicit conversions that the developer may not even notice.

Explicit conversions, conversely, are those for which the developer recognizes the need to change a variable's type and assign it to a different variable. Unlike implicit conversions, explicit conversions are easily recognizable within the code. Some languages such as C# require that essentially all conversions that might be type unsafe be done through an explicit conversion; otherwise, an error is thrown.

It is therefore important to understand what a type-safe implicit conversion is. In short, it's a conversion that cannot fail because of the nature of the data involved. For example, if you assign the value of a smaller type, `Short`, into a larger type, `Long`, then there is no way this conversion can fail. As both values are integer-style numbers, and the maximum and minimum values of a `Short` variable are well within the range of a `Long`, this conversion will always succeed and can safely be handled as an implicit conversion:

```
Dim shortNumber As Short = 32767
Dim longNumber As Long = shortNumber
```

However, the reverse of this is not a type-safe conversion. In a system that demands explicit conversions, the assignment of a `Long` value to a `Short` variable results in a compilation error, as the compiler doesn't have any safe way to handle the assignment when the larger value is outside the range of the smaller value. It is still possible to explicitly cast a value from a larger type to a smaller type, but this

is an explicit conversion. By default, Visual Basic supports certain unsafe implicit conversions. Thus, adding the following line will not, by default, cause an error under Visual Basic:

```
shortNumber = longNumber
```

This is possible for two reasons. One is based on Visual Basic's legacy support. Previous versions of Visual Basic supported the capability to implicitly cast across types that don't fit the traditional implicit casting boundaries. It has been maintained in the language because one of the goals of Visual Basic is to support rapid prototyping. In a rapid prototyping model, a developer is writing code that ''works'' for demonstration purposes but may not be ready for deployment. This distinction is important because in the discussion of implicit conversions, you should always keep in mind that they are not a best practice for production software.

Implicit Conversions and Compiler Options

As noted in the introduction to this section, Visual Basic supports certain unsafe implicit conversions. This capability is on by default but can be disabled in two ways. The first method is specific to each source file and involves adding a line to the top of the source file to indicate to the compiler the status of Option Strict.

The following line will override whatever the default project setting for Option Strict is for your project. However, while this can be done on a per-source listing basis, this is not the recommended way to manage Option Strict. For starters, consistently adding this line to each of your source files isn't a good practice:

```
Option Strict On
```

The preferred method to manage the Option Strict setting is to change the setting for your entire project. Without going into details about the XML associated with your project file, the easiest way to accomplish this is to use Visual Studio 2008. Visual Studio 2008 and the various versions of this tool are discussed in more detail in Chapter 13; however, for completeness, the compilation settings are discussed in this context.

Visual Studio 2008 includes a tab on the Project Settings page to edit the compiler settings for an entire project. You can access this screen by right-clicking the project in the Solution Explorer and selecting Properties from the context menu. When you select the Compile tab of the Project Properties dialog, you should see a window similar to the one shown in Figure 1-4.

Aside from your default project file output directory, this page contains several compiler options. These options are covered here because the Option Explicit and Option Strict settings directly affect your variable usage:

- **Option Explicit** — This option has not changed from previous versions of Visual Basic. When enabled, it ensures that every variable is explicitly declared. Of course, if you are using Option Strict, then this setting doesn't matter because the compiler won't recognize the type of an undeclared variable. To my knowledge, there's no good reason to ever turn this option off.

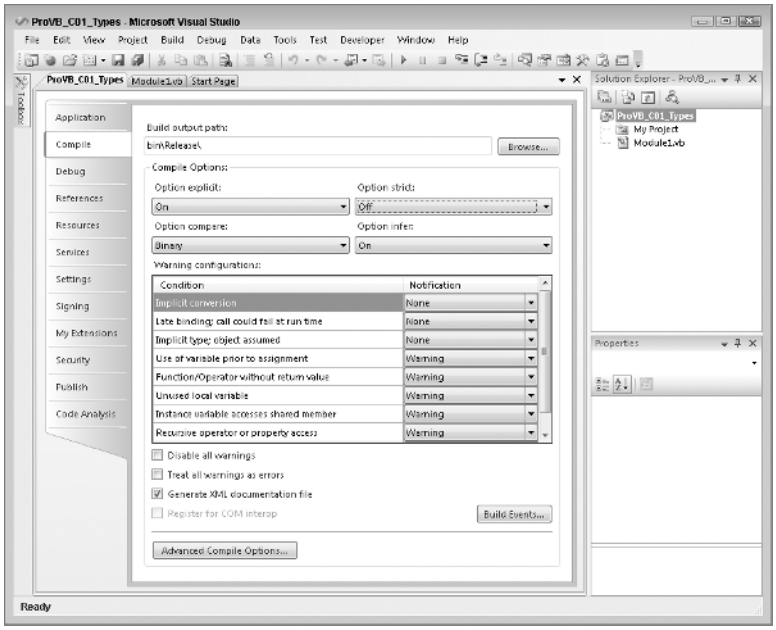

Figure 1-4

- **Option Strict** — When this option is enabled, the compiler must be able to determine the type of each variable, and if an assignment between two variables requires a type conversion — for example, from `Integer` to `Boolean` — then the conversion between the two types must be expressed explicitly. This setting can be edited by adding an Option Strict declaration to the top of your source code file. The statement within a source file applies to all of the code entered in that source file, but only to the code in that file.
- **Option Compare** — This option determines whether strings should be compared as binary strings or whether the array of characters should be compared as text. In most cases, leaving this as binary is appropriate. Doing a text comparison requires the system to convert the binary values that are stored internally prior to comparison. However, the advantage of a text-based comparison is that the character ''A'' is equal to ''a'' because the comparison is case-insensitive. This enables you to perform comparisons that don't require an explicit case conversion of the compared strings. In most cases, however, this conversion still occurs, so it's better to use binary comparison and explicitly convert the case as required.
- **Option Infer** — This option is new to Visual Studio 2008 and is brought to you by the requirements of LINQ. When you execute a LINQ statement, you can have returned a data table that may or may not be completely typed in advance. As a result, the types need to be inferred when the command is executed. Thus, instead of a variable that is declared without an explicit type being defined as an object, the compiler and runtime attempt to infer the correct type for this object.

  Existing code developed with Visual Studio 2005 is unaware of this concept, so this option will be off by default for any project that is migrated to Visual Studio 2008. New projects will have this option turned on, but this means that if you cut and paste code from a Visual Studio 2005 project into a Visual Studio 2008 project, or vice versa, you'll need to be prepared for an error in the pasted code because of changes in how types are inferred.

In addition to setting `Option Explicit`, `Option Strict`, `Option Compare`, and `Option Infer` to either `On` or `Off` for your project, Visual Studio 2008 allows you to customize specific compiler conditions that may occur in your source file. Thus, it is possible to leverage individual settings, such as requiring early binding as opposed to runtime binding, without limiting implicit conversions. These individual settings are included in the table of individual compiler settings listed below the `Option Strict` setting. Therefore, you can literally create a custom version of the `Option Strict` settings by turning on and off individual compiler settings for your project.

Notice that as you change your `Option Strict` setting, the notifications with the top few conditions are automatically updated to reflect the specific requirements of this new setting. In general, this table lists a set of conditions that relate to programming practices you might want to avoid or prevent, and which you should definitely be aware of. The use of warnings for the majority of these conditions is appropriate, as there are valid reasons why you might want to use or avoid each.

Basically, these conditions represent possible runtime error conditions that the compiler can't truly detect, except to identify that an increased possibility for error exists. Selecting Warning for a setting bypasses that behavior, as the compiler will warn you but allow the code to remain. Conversely, setting a behavior to `Error` prevents compilation.

An example of why these conditions are noteworthy is the warning on accessing shared member variables. If you are unfamiliar with shared member values, they are part of the discussion of classes in Chapter 2. At this point, it's just necessary to understand that these values are shared across all instances of a class. Thus, if a specific instance of a class is updating a shared member value, then it is appropriate to get a warning to that effect. The action is one that can lead to errors, as new developers sometimes fail to realize that a shared member value is common across all instances of a class, so if one instance updates the value, then the new value is seen by all other instances.

While many of these conditions are only addressed as individual settings, Visual Studio 2008 carries forward the `Option Strict` setting. Most experienced developers agree that using `Option Strict` and being forced to recognize when type conversions are occurring is a good thing. Certainly, when developing software that will be deployed in a production environment, anything that can be done to help prevent runtime errors is desirable. However, `Option Strict` can slow the development of a program because you are forced to explicitly define each conversion that needs to occur. If you are developing a prototype or demo component that has a limited life, you might find this option limiting.

If that were the end of the argument, then many developers would simply turn the option off and forget about it, but `Option Strict` has a runtime benefit. When type conversions are explicitly identified, the system performs them faster. Implicit conversions require the runtime system to first identify the types involved in a conversion and then obtain the correct handler.

Another advantage of `Option Strict` is that during implementation, developers are forced to consider every place a conversion might occur. Perhaps the development team didn't realize that some of the assignment operations resulted in a type conversion. Setting up projects that require explicit conversions means that the resulting code tends to have type consistency to avoid conversions, thus reducing the number of conversions in the final code. The result is not only conversions that run faster, but also, it is hoped, a smaller number of conversions.

As for `Option Infer`, well, it is a powerful new feature. On the one hand, it will be used as part of LINQ and the features that support LINQ, but it affects all code. In the past you needed to write the `AS <mytype>` portion of every variable definition in order to have a variable defined with an explicit type. However, now you can dimension a variable and assign it an integer or set it equal to another object, and the `AS`

`Integer` portion of your declaration isn't required. On the other hand, you can now dimension a variable and assign it to a specific type without declaration, which reduces the readability of your code. Be careful with `Option Infer`; it can make your code obscure.

In addition, note that `Option Infer` is directly affected by `Option Strict`. In an ideal world, `Option Strict Off` would require that `Option Infer` also be turned off or disabled in the user interface. That isn't the case, although it is the behavior that is seen; once `Option Strict` is off, `Option Infer` is essentially ignored.

How `Option Infer` *is used in LINQ is covered in Chapter 11.*

XML Literals

One of the main new features in Visual Basic 2008 is the introduction of XML literals. With Visual Studio 2008, it is possible within Visual Basic to create a new variable and assign a block of well-formatted XML code to that string. This is being introduced here because it demonstrates a great example of a declaration that leverages `Option Infer`. Start by declaring a string variable called `myString` and setting this to a value such as `"Hello World"`. In the code block that follows, notice that the first `Dim` statement used does not include the `"As"` clause that is typically used in such declarations:

```
        Dim myString = "Hello World"
        Dim myXMLElement = <MyXMLNode attribute1="1">This is formatted Text.
Print these lines separately.
    Ensure whitespace is also maintained.
<%= myString %>
                           </MyXMLNode>
```

Instead, the declaration of the `myString` variable relies on type inference. The compiler recognizes that this newly declared variable is being assigned a string, so the variable is automatically defined as a string. After the first variable is declared on the first line of the code block, the second line of code makes up the remainder of the code block, and you may notice that it spans multiple lines without any line continuation characters.

The second `Dim` statement declares another new variable, but in this case the variable is set equal to raw XML. Note that the ''<'' is not preceded by any quotes in the code. Instead, that angle bracket indicates that what follows will be a well-formed XML statement. At this point the Visual Basic compiler stops treating what you have typed as Visual Basic code and instead reads this text as XML. Thus, the top-level node can be named, attributes associated with that node can be defined, and text can be assigned to the value of the node. The only requirement is that the XML be well formed, which means you need to have a closing declaration, the last line in the preceding code block, to end that XML statement.

By default, because this is just an XML node and not a full document, Visual Basic infers that you are defining an `XMLElement` and will define the `mXMLElement` variable as an instance of that class. Beyond this, however, there is the behavior of your static XML. Note that the text itself contains comments about being formatted. That is because within your static XML, Visual Basic automatically recognizes and embeds literally everything.

Thus, the name *XML literal*. The text is captured as is, with any embedded white space or ''carriage returns''/line feeds captured. The other interesting capability is shown on the line that reads as follows:

```
<%= myString %>
```

This is a shorthand declaration that enables you to insert the value of the variable `myString` into your literal XML. In this case, `myString` is set on the preceding line, but it could easily be an input parameter to a method that returns an XML element. When you run this code, the current value of `myString` will be inserted into your XML declaration.

Figure 1-5 shows a bit more code than you saw in the preceding code block. It also includes a set of `Console.WriteLine` statements. These statements were added to display the data from your new XML element. Two different statements displaying the contents of the XML element as a string appear because each results in slightly different output:

```
Console.WriteLine("----------The XML-----------")
Console.WriteLine(myXMLElement.ToString())
Console.WriteLine()
Console.WriteLine("----------The Data----------")
Console.WriteLine(myXMLElement.Value.ToString())
```

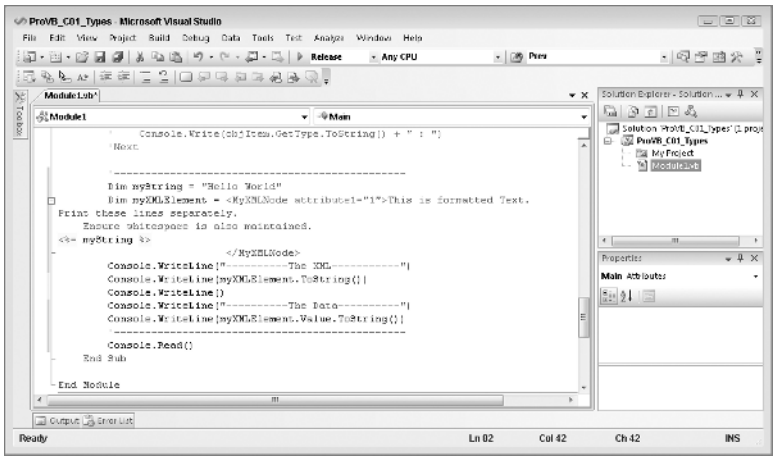

Figure 1-5

Of the five `Console.WriteLine` statements, only the second and fifth are important. The first statement on the second line instructs the XML element object to return a string representing itself. As such, the XML element will return all of the content of that object, including the raw XML itself. The writeline that ends the preceding code block has output the XML element to a string that only reflects the value of the data defined for that element. Note that if the basic XML element you defined in the previous code block had any nested XML elements, then these would be considered part of the contents of your XML element, and their definitions and attributes would be output as part of this statement.

As shown in Figure 1-6, the result of this output is that the first block of text outputted includes your custom XML node and its attribute. Not only do you see the text that identifies the value of the XML, you also see that actual XML structure. However, when you instead print only the value from the XML block, what you see is in fact just that text. Note that XML has embedded the carriage returns and left-hand white space that was part of your XML literal so that your text appears formatted. With the use of XML literals, you ''literally'' have the capability to replace the somewhat cryptic `String.Format` method call with a very explicit means of formatting an output string. Of course, not everything can rely on `Option Infer` and implicit conversions.

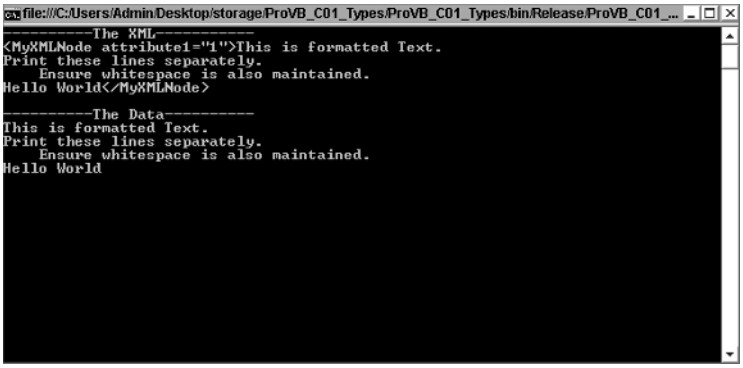

Figure 1-6

Performing Explicit Conversions

Keep in mind that even when you choose to allow implicit conversions, these are only allowed for a relatively small number of data types. At some point you'll need to carry out explicit conversions. The following code is an example of some typical conversions between different integer types when `Option Strict` is enabled:

```
Dim myShort As Short
Dim myUInt16 As UInt16
Dim myInt16 As Int16
Dim myInteger As Integer
Dim myUInt32 As UInt32
Dim myInt32 As Int32
Dim myLong As Long
Dim myInt64 As Int64

myShort = 0
myUInt16 = Convert.ToUInt16(myShort)
myInt16 = myShort
myInteger = myShort
myUInt32 = Convert.ToUInt32(myShort)
myInt32 = myShort
myInt64 = myShort

myLong = Long.MaxValue
If myLong < Short.MaxValue Then
  myShort = Convert.ToInt16(myLong)
End If
myInteger = CInt(myLong)
```

The preceding snippet provides some excellent examples of what might not be intuitive behavior. The first thing to note is that you can't implicitly cast from `Short` to `UInt16`, or any of the other unsigned types for that matter. That's because with `Option Strict` the compiler won't allow an implicit conversion that might result in a value out of range or loss of data. You may be thinking that an unsigned `Short` has a maximum that is twice the maximum of a signed `Short`, but in this case, if the variable `myShort` contained a `-1`, then the value wouldn't be in the allowable range for an unsigned type.

Just for clarity, even with the explicit conversion, if `myShort` were a negative number, then the `Convert.ToUInt32` method would throw a runtime exception. Managing failed conversions requires either an understanding of exceptions and exception handling, as covered in Chapter 8, or the use of a conversion utility such as `TryParse`, covered later in this section.

The second item illustrated in this code is the shared method `MaxValue`. All of the integer and decimal types have this property. As the name indicates, it returns the maximum value for the specified type. There is a matching `MinValue` method for getting the minimum value. As shared properties, the properties can be referenced from the class (`Long.MaxValue`) without requiring an instance.

Finally, although this code will compile, it won't always execute correctly. It illustrates a classic error, which in the real world is often intermittent. The error occurs because the final conversion statement does not check to ensure that the value being assigned to `myInteger` is within the maximum range for an integer type. On those occasions when `myLong` is larger than the maximum allowed, this code will throw an exception.

Visual Basic provides many ways to convert values. Some of them are updated versions of techniques that are supported from previous versions of Visual Basic. Others, such as the `ToString` method, are an inherent part of every class (although the .NET specification does not guarantee how a `ToString` class is implemented for each type).

The following set of conversion methods is based on the conversions supported by Visual Basic. They coincide with the primitive data types described earlier; however, continued use of these methods is not considered a best practice. That bears repeating: While you may find the following methods in existing code, you should strive to avoid and replace these calls:

`CBool()`	`CByte()`
`CChar()`	`CDate()`
`CDbl()`	`CDec()`
`CInt()`	`CLng()`
`CObj()`	`CShort()`
`CSng()`	`CStr()`

Each of these methods has been designed to accept the input of the other primitive data types (as appropriate) and to convert such items to the type indicated by the method name. Thus, the `CStr` class is used to convert a primitive type to a `String`. The disadvantage of these methods is that they have been designed to support any object. This means that if a primitive type is used, then the method automatically boxes the parameter prior to getting the new value. This results in a loss of performance. Finally, although these are available as methods within the VB language, they are actually implemented in a class (as with everything in the .NET Framework). Because the class uses a series of type-specific overloaded methods, the conversions run faster when the members of the `Convert` class are called explicitly:

```
Dim intMyShort As Integer = 200
Convert.ToInt32(intMyShort)
Convert.ToDateTime("9/9/2001")
```

The classes that are part of `System.Convert` implement not only the conversion methods listed earlier, but also other common conversions. These additional methods include standard conversions for things such as unsigned integers and pointers.

All the preceding type conversions are great for value types and the limited number of classes to which they apply, but these implementations are oriented toward a limited set of known types. It is not possible to convert a custom class to an `Integer` using these classes. More important, there should be no reason to have such a conversion. Instead, a particular class should provide a method that returns the appropriate type. That way, no type conversion is required. However, when `Option Strict` is enabled, the compiler requires you to cast an object to an appropriate type before triggering an implicit conversion. Note, however, that the `Convert` method isn't the only way to indicate that a given variable can be treated as another type.

Parse and TryParse

Most value types, at least those which are part of the .NET Framework, provide a pair of shared methods called `Parse` and `TryParse`. These methods accept a value of your choosing and then attempt to convert this variable into the selected value type. The `Parse` and `TryParse` methods are only available on value types. Reference types have related methods called `DirectCast` and `Cast`, which are optimized for reference variables.

The `Parse` method has a single parameter. This input parameter accepts a value that is the target for the object you are looking to create of a given type. This method then attempts to create a value based on the data passed in. However, be aware that if the data passed into the `Parse` method cannot be converted, then this method will throw an exception that your code needs to catch. The following line illustrates how the `Parse` function works:

```
result = Long.Parse("100")
```

Unfortunately, when you embed this call within a `Try-Catch` statement for exception handling, you create a more complex block of code. Because you always need to encapsulate such code within a `Try-Catch` block, the .NET development team decided that it would make more sense to provide a version of this method that encapsulated that exception-handling logic.

This is the origin of the `TryParse` method. The `TryParse` method works similarly to the `Parse` method except that it has two parameters and returns a Boolean, rather than a value. Instead of assigning the value of the `TryParse` method, you test it as part of an `If-Then` statement to determine whether the conversion of your data to the selected type was successful. If the conversion was successful, then the new value is stored in the second parameter passed to this method, which you can then assign to the variable you want to hold that value:

```
Dim converted As Long
If Long.TryParse("100", converted) Then
    result = converted
End If
```

CType

The `CType` method accepts two parameters: the first is the object that is having its type cast, and the second is the name of the object to which it is being cast. This system enables you to cast objects from parent to child types or from child to parent types. There is a limitation to the second parameter in that

it can't be a variable containing the name of the casting target. Casting is defined at compile time, and any form of dynamic name selection would occur at runtime. An example of casting was shown as part of the discussion of working with the `Object` class earlier in this chapter.

Support for a runtime determination of object types is based on treating variables as objects and using the object metadata and the `TypeOf` operator to verify that an object supports various method and property calls. The `CType` method accepts both value and reference types. More detailed information regarding its use is presented in Chapter 2.

DirectCast and TryCast

The `DirectCast` method works similarly to the `CType` method, with a couple of minor differences. First, unlike `CType`, the `DirectCast` method accepts only reference types. This is because the `DirectCast` method is tied much more closely to objects and the use of inheritance and interfaces. Additionally, in order to make it perform faster, `DirectCast` does not include any logic to actually check for and convert an object to the requested type. The `DirectCast` method is meant to allow your code to take an object that has been cast as its base type of object and recast it in its original form. Similar to `CType`, these methods are covered in more detail in Chapter 2.

Summary

This chapter looked at many of the basic building blocks of Visual Basic that are used throughout project development. Understanding not only the basic components, but also how they work will help you to write more stable and better performing software. Note the following highlights of this chapter:

- Beware of array sizes; all arrays start at 0 and are defined not by size, but by the highest index.
- Remember to use the `StringBuilder` class for string manipulation.
- Pay attention to variable scope, and rely on it for cleaning up variables you no longer need.
- Use `Option Strict`; it's not about style, it's about reliability and performance.
- Try to avoid legacy methods for conversions.
- Attempt to leverage the `TryParse` and `TryCast` methods.
- Understand variable scope and when variables will go out of scope.

While this chapter covered many other items, including the `Decimal` type and how boxing works, these bullets highlight some of the more important items. Whether you are creating a new library of methods or a new user interface, these items consistently turn up in some form. You have seen that while .NET provides a tremendous amount of power, that power comes at a sometimes significant performance cost.