CHAPTER 2

Data Types and Variables

This chapter will begin by examining the intrinsic data types supported by Visual Basic and relating them to their corresponding types available in the .NET Framework's Common Type System. It will then examine the ways in which variables are declared in Visual Basic and discuss variable scope, visibility, and lifetime. The chapter will conclude with a discussion of boxing and unboxing (that is, of converting between value types and reference types).

Visual Basic Data Types

At the center of any development or runtime environment, including Visual Basic and the .NET Common Language Runtime (CLR), is a type system. An individual *type* consists of the following two items:

- A set of values.
- A set of rules to convert every value not in the type into a value in the type. (For these rules, see Appendix F.) Of course, every value of another type cannot always be converted to a value of the type; one of the more common rules in this case is to throw an InvalidCastException, indicating that conversion is not possible.

Scalar or primitive types are types that contain a single value. Visual Basic 2005 supports two basic kinds of scalar or primitive data types: structured data types and reference data types. All data types are inherited from either of two basic types in the .NET Framework Class Library. Reference types are derived from System.Object. Structured data types are derived from the System.ValueType class, which in turn is derived from the System.Object class. Although we will discuss the difference between value types and reference types in greater detail later in this chapter, suffice it to say that the .NET Common Language Infrastructure (CLI) manipulates value types by value and reference types by reference.

To appreciate the difference, consider the following very simple code, which defines a StudentRef class (a reference type) and a StudentVal structure (a value type) to store the name and grade point average of a student, and then creates instances of both:

```
Public Class StudentRef
   Public Name As String
   Public GPA As Double
End Class
```

```
Public Structure StudentVal
   Public Name As String
   Public GPA As Double
End Structure

Public Class Test
   Public Shared Sub Main()
      Dim studentRef1 As New StudentRef()
      Dim studentVal1 As New StudentVal()

      studentRef1.Name = "Jill"
      studentRef1.GPA = 92.3
      studentVal1.Name = "Jill"
      studentVal1.GPA = 92.3

      Dim studentRef2 As StudentRef = studentRef1
      Dim studentVal2 As StudentVal = studentVal1

      studentRef2.GPA += 2
      studentVal2.GPA += 2

      Console.WriteLine("{0}'s GPA is: {1}", studentRef1.Name, studentRef1.GPA)
      Console.WriteLine("{0}'s GPA is: {1}", studentVal1.Name, studentVal1.GPA)
   End Sub
End Class
```

The Main procedure creates an instance of the class, named studentRef1, and an instance of the structure, named studentVal1. The value of studentRef1 is then assigned to a new class instance, studentRef2, and the value of studentVal1 is assigned to a new structure instance, studentVal2. The GPA fields of both studentRef2 and studentVal2 are next incremented by two. Then the GPAs of StudentRef1 and StudentVal1 are displayed to the console. The results are as follows:

```
Jill's GPA is: 94.3
Jill's GPA is: 92.3
```

Note that studentRef1.GPA has changed even though it was studentRef2.GPA that our code modified. That's because studentRef1 and studentRef2 are expressions that both refer to the same instance of the StudentRef class. In contrast, studentVal1.GPA remains unchanged. That's because studentVal1 and studentVal2 are instances of the StudentVal structure, a value type. When we assigned studentVal1 to studentVal2, we assigned studentVal2 the values of the first structure, not a reference to a single set of data.

Table 2-1 lists the data types supported by Visual Basic and shows which are value types and which are reference types.

VB 6 and VB.NET Data Types

Visual Basic .NET supports all but two of the scalar data types supported by COM-based versions of Visual Basic. The Currency type has been eliminated; the Decimal type should be used instead. In addition, the Variant type, the "universal type" of COM-based Visual Basic programming, is no longer supported. Instead, Object is the universal data type in Visual Basic .NET.

VB Type Name	.NET Type Name	Type
Boolean	System.Boolean	Value
Byte	System.Byte	Value
Char	System.Char	Value
Date	System.DateTime	Value
Decimal	System.Decimal	Value
Double	System.Double	Value
Integer	System.Int32	Value
Long	System.Int64	Value
Object	System.Object	Reference
SByte	System.SByte	Value
Short	System.Int16	Value
Single	System.Single	Value
String	System.String	Reference
UInteger	System.UInt32	Value
ULong	System.UInt64	Value
UShort	System.UInt16	Value
user-defined class		Reference
user-defined value type		Value

TABLE 2-1 Visual Basic 2005 Data Types

As we'll see, Visual Basic's intrinsic data types directly correspond to particular .NET data types. Because of this, we'll begin by briefly discussing the .NET Common Type System and the relationship of .NET types to Visual Basic data types. Then we'll turn to each of the intrinsic data types supported by Visual Basic. This chapter provides some basic information about each data type. However, you'll also want to look at Appendix G, which briefly surveys the members (the fields, methods, and properties) of each data type. In addition, since type conversion can be much more important in Visual Basic .NET if Option Strict is set to On (its preferred setting), you may want to examine Appendix F, which lists the methods available to convert any data type into any other data type.

The Common Type System and Shared Functionality

One of the basic features of the .NET platform is the Common Type System. It means that the .NET CLI supplies the data types used by .NET-compliant languages and development environments.

This allows language independence and easy interoperability of components created in different languages. The existence of a common type system has enormous implications for Visual Basic: although Visual Basic .NET, like the COM-based versions of Visual Basic, appears to support

a number of "intrinsic" data types, in fact these intrinsic data types are merely wrappers around basic data types defined by the .NET framework. For example, running the following code:

```
Public Class WrappedType
   Public Shared Sub Main()
      Dim intVar As Integer = 100
      Dim intType As Type = intVar.GetType()
      Console.WriteLine(TypeName(intVar) & " = " & intType.FullName)

      Dim stringVar As String = "This is a string."
      Dim stringType As Type = stringVar.GetType()
      Console.WriteLine(TypeName(stringVar) & " = " & stringType. FullName)

      Dim boolVar As Boolean = True
      Dim boolType As Type = boolVar.GetType()
      Console.WriteLine(TypeName(boolVar) & " = " & boolType. FullName)

      Dim singleVar As Single = 3.1417
      Dim singleType As Type = singleVar.GetType()
      Console.WriteLine(TypeName(singleVar) & " = " & singleType. FullName)
   End Sub
End Class
```

produces the following output in a console window:

```
Integer = System.Int32
String = System.String
Boolean = System.Boolean
Single = System.Single
```

This code uses the Visual Basic TypeName function, which returns the name of an "intrinsic" Visual Basic data type, to determine the data type of each variable. It also uses the .NET GetType method to retrieve a Type object representing the data type of a variable. Calling the Type object's FullName method returns the fully qualified name of the data type. The example then uses the Console.WriteLine method to write the results of these two method calls to a command window. As the output shows, the Visual Basic data type name corresponds to a .NET data type name.

Besides language independence, the identity of Visual Basic data types and .NET data types has a major advantage: all .NET types support members (that is, fields, properties, and methods) that are accessible from Visual Basic intrinsic data types. For example, the Visual Basic Date data type wraps the .NET DateTime structure. The following code uses the DateTime structure's constructor to initialize a Date variable, and then calls the DateTime AddMonths method to add five months to the date. Finally, it uses the addition operator (which is equivalent to the DateTime structure's op_Addition method) to add 30 days to the date:

```
Public Class DateMembers
   Public Shared Sub Main()
      Dim dat As Date = New Date(2000, 1, 1)
      Dim ts As New TimeSpan(30, 0, 0, 0)

      Console.WriteLine("The original date is " & dat)
      dat = dat.AddMonths(5)
```

```
      Console.WriteLine("The new date is " & dat)
      dat = dat + ts
      Console.WriteLine("The new date is " & dat)
   End Sub
End Class
```

Appendix E lists the members of each Visual Basic .NET data type.

In the following sections, we'll examine each of Visual Basic's intrinsic data types in greater detail.

Scalar Data Types

Scalar data types are designed to hold a single item of data—a number, a string, a flag, or an indication of state. In this section, we'll examine each of the scalar data types supported by Visual Basic .NET. They are shown in Table 2-2. Note that the literal identifiers are case insensitive; an unsigned long literal, for example, can be identified with a UL, ul, Ul, or uL.

In the remainder of this section, we'll discuss the data types supported by Visual Basic in greater detail.

Boolean

The Boolean type corresponds to the System.Boolean structure and consists of the set {True, False}. An uninitialized Boolean variable is assigned a value of False.

Byte

The Byte data type, which corresponds to the System.Byte structure, holds an eight-bit unsigned integer; its value can range from 0 to 255. (Visual Basic 2005 also adds intrinsic support for a signed byte data type, SByte.) An uninitialized Byte variable is assigned a value of 0.

Char

The Char data type, which corresponds to the System.Char structure, holds a two-byte character code representing a particular UTF-16 Unicode character. Character codes can range from 0 to 65,535. An uninitialized Char variable is assigned a value of character code 0. The Char data type was not supported in Visual Basic 6.0.

When assigning a value of type Char, you can use the literal type character *C* to treat a literal string as a Char literal. For example, if Option Strict is on, the code

```
Dim ch As Char = "a"
```

produces a compiler error ("Option Strict On disallows implicit conversions from 'String' to 'Char'."). The literal type character defines the data type of the "a" as Char and corrects the error, as the following code illustrates:

```
Dim ch As Char = "a"c
```

Date

The Date data type, which corresponds to the System.DateTime structure, can range from midnight (0:00:00) of January 1, 0001, to 11:59:59 P.M. of December 31, 9999. An uninitialized Date variable is assigned a value of 12:00:00 A.M. on January 1, 0001. Date values occupy eight bytes of memory.

In addition to using the Date constructors, you can also use # characters as literal type characters. For example, the following code initializes a date variable to July 1, 2005:

```
Dim dat As Date = #07/01/2005#
```

Data Type	Type Identifier	Literal Identifier	Range
Boolean	None	None	True, False
Byte	None	None	0 to 255
Char	None	C	UTF-16 character codes from 0 to 65,535
Date	None	#date#	12:00:00 A.M., 1/1/0001 to 11:59:59 P.M., 12/31/9999
Decimal	@	D	–79,228,162,514,264,337,593,543,950,335 to 79,228,162,514,264,337,593,543,950,335. A Decimal value is stored internally as a scaled value, making it far more precise than either the Single or the Double.
Double	#	R	$-1.79769313486231570^{308}$ to $1.79769313486231570^{308}$, as well as positive or negative zero, positive infinity (the result of an operation that exceeds Single.MaxValue or of division of a positive Single value by zero), negative infinity (the result of an operation less than Single.MinValue or of division of a negative Single value by zero), and NaN (not a number), indicating the result of an operation is undefined. A Double value has 15 decimal digits of precision.
Integer	%	I	–2,147,483,648 to 2,147,483,647
Long	&	L	–9,223,372,036,854,775,808 through 9,223,372,036,854,775,807
Object	None	none	Any type
SByte	None	none	–128 to 127
Short	None	S	–32,768 through 32,767
Single	!	F	-3.4028235^{38} to 3.4028235^{38}, as well as positive or negative zero, positive infinity (the result of an operation that exceeds Single.MaxValue or of division of a positive Single value by zero), negative infinity (the result of an operation less than Single.MinValue or of division of a negative Single value by zero), and NaN (not a number), indicating the result of an operation is undefined). A Single value has seven decimal digits of precision).
String	$	Quotation marks	Limited by available memory
UInteger	None	UI	0 to 4,294,967,295
ULong	None	UL	0 to 18,446,744,073,709,551,615
UShort	None	US	0 to 65,535

TABLE 2-2 Scalar Data Types in Visual Basic .NET

In .NET, the Date (or DateTime) type has been substantially enhanced to provide more information about a date (such as the date's time zone) as well as much of the functionality required to work with dates. You can, for instance, convert a Date value to its equivalent FILETIME value (used by the Win32 API) or its equivalent OLE Automation value (a Date value in Visual Basic 6.0). You can also use arithmetic operators to manipulate and compare dates. For details, see the members of the DataTime structure in Appendix G.

Decimal

The Decimal data type, which corresponds to the System.Decimal structure, can be used to hold currency values or values in which rounding error in the most significant fractional digits is unacceptable. (It stores data in a scaled format that minimizes rounding error.) It replaces the Currency data type of Visual Basic 6.0 and earlier versions. The Decimal data type can hold values ranging from –79,228,162,514,264,337,593,543,950,335 to 79,228,162,514,264,337,593,543,950,335. The value of an uninitialized Decimal variable is 0.

You can use the literal type character *D* to identify a value as a Decimal literal in an assignment statement. For example, if Option Strict is on, attempting to compile code like the following

```
Dim amt As Decimal = 100000
```

may produce a compiler error ("Option Strict On disallows implicit conversions from 'Double' to 'Decimal'."). Instead, you can use the literal type character *D* to define the value as a decimal. You can also use @ as an identifier type character, which allows you to eliminate "As Decimal" in a decimal's variable declaration. For example:

```
Dim amt As Decimal = 100000D
Dim int@ = .10d
```

Double

The Double data type, which corresponds to the System.Double structure, can contain a double-precision floating point number. It can have a value ranging from $-1.79769313486231570^{308}$ to $1.79769313486231570^{308}$, as well as positive or negative zero, positive infinity, negative infinity, and NaN (not a number). A Double value has 15 decimal digits of precision. Uninitialized Double variables are automatically set equal to 0.

When declaring variables of type Double, you can use # as an identifier type character, which replaces the "As Double" clause in a variable declaration. In assignment statements involving literal values of type Double, you can use the literal type character *R* to treat a numeric literal as a Double. For example:

```
Dim dist As Double = 10101.3654R
Dim spd# = 68.7931r
```

Integer

The Integer data type, which corresponds to the System.Int32 structure, occupies 32 bits (or four bytes) of memory and can hold values that range from –2,147,483,648 through 2,147,483,647. Uninitialized Integer variables are automatically set equal to 0.

When declaring a variable of type Integer, you can use % as an identifier type character, which substitutes for the clause "As Integer" in a variable declaration. For example:

```
Dim age1% = 43i
```

Integers in VB 6 and VB.NET
In VB 6.0 and earlier versions, the Integer was a 16-bit data type. In contrast, in Visual Basic .NET, the Integer is a 32-bit data type, which means that it is equivalent to the VB 6.0 Long data type. So, if you are making Win32 API calls from .NET code, you should use an Integer whenever a Win32 function argument requires a Long.

You can also use the literal type character *I* to treat a numeric value as an Integer. For example:

```
Dim age2 As Integer = 48I
```

Long

The Long data type is a signed integer that corresponds to the System.Int64 structure. Variables of type Long occupy 64 bits (or eight bytes) of memory and can range in value from –9,223,372,036,854,775,808 through 9,223,372,036,854,775,807. Uninitialized long integers are automatically set equal to 0.

When declaring a variable of type Long, you can use *&* as an identifier type character, which substitutes for the clause "As Long" in a variable declaration. For example:

```
Dim lng1& = 1800000000001
```

You can also use the literal type character *L* to treat a numeric value as a Long. For example:

```
Dim lng2 As Long = 170000000L
```

Object

The Object data type, which corresponds to the System.Object class, replaces the VB 6.0 (and earlier) Variant as Visual Basic's universal data type. In .NET, Object is a universal data type for another reason: all .NET types other than interfaces are directly or indirectly derived from it. The value of an uninitialized Object variable is Nothing.

You can assign an instance of any reference type or value type to a variable typed as Object. Assigning an instance of a value type to a variable typed as Object incurs a "boxing" operation to convert the value type instance to a reference type instance. We'll discuss boxing and unboxing toward the end of this chapter.

You can determine what kind of data an object variable holds at a particular time by calling the Visual Basic TypeName function, which returns the Visual Basic data type name.

You can also call the object's GetType method, which returns a Type object representing the type of the object instance. Its Name property provides you with the .NET type name, and its

Longs in VB 6 and VB.NET
In VB 6.0 and earlier versions, the Long was a 32-bit data type. In contrast, in Visual Basic .NET, the Long is a 64-bit data type. In other words, VB 6.0 and earlier versions had no equivalent of the Long data type in Visual Basic .NET.

FullName property returns the fully qualified type name (including hierarchical namespaces). For example:

```
Dim o As Object = "This is an object."

Console.Writeline(TypeName(o))                ' Displays String
Console.WriteLine(o.GetType.Name)             ' Displays String
Console.WriteLine(o.GetType.FullName)         ' Displays System.String
```

The Object data type is most convenient when you do not know the precise type of data a variable will hold at runtime. But the use of a variable of type Object to hold other types of data results in a performance penalty as a result of late binding, since only at runtime is the CLR able to perform such functions as member name resolution (determining whether a type actually has a member of a particular name) and argument resolution (determining whether a member has arguments that correspond to those used in a method call). There are numerous benefits as well: strongly typed code is easier to read, easier to optimize, easier to test, easier to debug, and easier to maintain.

You can use early binding in one of two ways:

- By declaring a variable to be a specific type.
- By explicitly converting a variable to the type needed for a particular operation. For conversions, see Appendix F.

The word "object" is used not only to refer to the Object data type, but also to refer generically to any instance of any class. For instance, code such as

```
Dim ht As New System.Collections.Hashtable
```

creates an "object of type Hashtable." Similarly, code such as

```
Public Class Person
End Class
```

defines a class in your own code. You can then instantiate "an object of type Person" or "an object of the Person class" with code like the following:

```
Dim per As Person = New Person()
```

SByte

The SByte data type, which corresponds to the System.SByte structure, holds an eight-bit signed integer; its value can range from –128 through 127. An uninitialized SByte variable is assigned a value of 0. It occupies one byte (eight bits) of memory. The SByte data type is new to Visual Basic 2005.

The SByte data type is not CLS-compliant. This means that to be compliant with the .NET platform, other languages are not required to implement support for a signed byte data type. As a result, although it is safe to use the SByte type internally in your own programs, you may not want to expose it as part of a component's public interface.

Variants in VB 6 and Objects in VB.NET

Instead of defining variables of type Variant as you did in VB 6.0 and earlier versions, you can now define variables to be of type Object. However, you should take a look at the reasons for using Object variables. If you use Visual Basic as a weakly typed language and allow the CLR to determine the precise runtime type of your variables, you might take a more careful look at your code to determine whether the data type of a variable can be more precisely defined. You should use the Object data type only when you legitimately do not know the precise data type of a variable at design time. In addition, when writing methods, take advantage of overloading (a method of the same name can have different signatures) rather than defining a generic Object parameter to allow for values of multiple data types. For example, if a grade report program includes a DisplayResult method that can either display a letter grade, a grade point average, or a numeric grade, rather than having a single method with the signature

```
Private Sub DisplayResult(gradeInfo As Object)
```

you should overload the method as follows to reflect the individual arguments that can be passed to it:

```
Private Sub DisplayResult(letterGrade As String)
Private Sub DisplayResult(gpa As Double)
Private Sub DisplayResult(grade As Integer)
```

Short

The Short is a 16-bit signed integral data type that corresponds to the System.Int16 structure in the .NET Framework Class Library. Values of Short variables can range from –32,768 through 32,767. The Short data type corresponds to the Visual Basic 6.0 Integer data type.

You can use the literal type character *s* to indicate that a numeric literal is a Short. For example:

```
Dim age1 As Short = 48S
Dim age2 As Short = 43s
```

Single

The Single data type, which corresponds to the System.Single structure, holds a single-precision floating point number that can range from -3.4028235^{38} to 3.4028235^{38}, as well as positive or negative zero, positive infinity, negative infinity, and NaN. A Single value has seven decimal digits of precision. Uninitialized Single variables are automatically set equal to 0.

The Short in VB.NET

VB 6.0 had only two signed integer types: Integer and Long. Visual Basic .NET adds two more, one of which is the Short type. However, since Short is a 16-bit signed integer, it corresponds directly to the Integer data type in the 32-bit versions of Visual Basic from Versions 3.0–6.0. If you are making Win32 API calls from .NET code, you can use a Short whenever a Win32 function argument is an Integer.

When declaring a variable of type Single, you can use *!* (the exclamation mark) as an identifier type character, which substitutes for the clause "As Single" in a variable declaration. When assigning a literal single value, you can use the literal type character *F* to indicate that a literal numeric value is a Single. For example:

```
Dim spd! = 68.7931F
Dim dist As Single = 10101.3654f
```

String

The String data type, which corresponds to the System.String type, is used to store a series of anywhere from 0 to about 2 billion 16-bit (two-byte) UTF-16 Unicode characters. Uninitialized strings are automatically initialized as zero-length strings and can be compared to the Visual Basic VbNullString constant, to an empty string, to the String.Empty field, or to the String.IsNullOrEmpty method. For example:

```
If s = VbNullString Then
   ' Do something
```

or

```
If s = "" Then
   ' Do something
```

or

```
If s = String.Empty Then
   ' Do something
```

or

```
If String.IsNullOrEmpty(s) Then
   ' Do something
```

Since String is a class, an uninitialized string has a value of Nothing. Hence, the following code also tests for an uninitialized string:

```
If s Is Nothing Then
   ' Do something
```

When declaring a variable of type String, you can use *$* as an identifier type character, which substitutes for the clause "As String" in a variable declaration. When initializing or assigning a value to a string variable, Visual Basic treats all text enclosed in quotation marks as the literal string to be assigned to the variable. For example:

```
Dim Name$ = "Gregory of Nyassa"
```

String is a reference type. But because many of the details of working with strings as a reference type are handled by the Visual Basic compiler and the CLR, strings appear to behave as value types. The String type has a second characteristic: it is immutable. That is, once you've initialized a string, neither its length nor its contents can be changed. What appears to be a modified string that results, for example, from concatenating text to a string is actually a new string variable that replaces the old.

One implication of the immutability of strings is that using the String class for code that involves extensive string manipulation involves a major performance penalty. When this is the case, you may choose instead to work with the System.Text.StringBuilder class, which is optimized for string manipulation.

UInteger

The UInteger data type is one of four unsigned integer data types supported by Visual Basic. It corresponds to the System.UInt32 structure and occupies 32 bits (or four bytes) of memory. Values of a UInteger variable can range from 0 to 4,294,967,295. Uninitialized UInteger variables are automatically set equal to 0. The UInteger data type was added to Visual Basic .NET as of Version 2.0.

When initializing a variable of type UInteger, you can also use the literal type character *UI* to treat a numeric literal as a UInteger. For example:

```
Dim age1 As UInteger = 48UI
Dim age2 As UInteger = 43ui
```

The UInteger data type is not CLS-compliant. This means that to be compliant with the .NET platform, other languages are not required to implement support for an unsigned integer data type. As a result, although it is safe to use the UInteger type internally in your own programs, you may choose not to expose it as part of a component's public interface if you care about CLS compliance.

ULong

The ULong data type is an unsigned long integer that corresponds to the System.UInt64 structure. Variables of type ULong occupy 64 bits (or 8 bytes) of memory and can range in value from 0 through 18,446,744,073,709,551,615. Uninitialized unsigned long integers are automatically assigned a value of 0. The ULong data type was added to Visual Basic .NET as of Version 2.0.

When initializing a variable of type ULong, you can also use the literal type character *UL* to treat a numeric literal as a ULong. For example:

```
Dim lng1 As ULong = 170000000UL
Dim lng2 As ULong = 180000000000ul
```

The ULong data type is not CLS-compliant. This means that to be compliant with the .NET platform, other languages are not required to implement support for an unsigned long integer data type. As a result, although it is safe to use the ULong type internally in your own programs, it should not be exposed as part of a component's public interface.

Unsigned Integer Data Types in Visual Basic

Although the .NET Framework provides four unsigned integer types, Visual Basic .NET originally offered intrinsic support for the Byte data type only. However, Visual Basic 2005 added support for UShort (or System.UInt16, a 16-bit unsigned integer), UInteger (or System.UInt32, a 32-bit unsigned integer), and ULong (or System.UInt64, a 64-bit unsigned integer).

UShort

UShort is a 16-bit unsigned integral data type that corresponds to the System.UInt16 structure in the .NET Framework Class Library. Values of UShort variables can range from 0 through 65,535. Uninitialized integers are automatically set equal to 0 by the Visual Basic .NET compiler. The UShort data type was added to Visual Basic .NET as of Version 2.0.

When initializing variables of type UShort, you can use the literal type character *US* to treat a numeric literal as a UShort. For example:

```
Dim age1 As Short = 48US
Dim age2 As Short = 43us
```

Like all the other unsigned data types except for Byte, UShort is not CLS-compliant. Although it is safe to use variables of type UShort internally in your components and applications, you should avoid exposing them as part of a component's public interface, since consumers of your component may not support the UShort or UInt16 data types.

Complex Data Types

Visual Basic .NET also supports two kinds of complex data types: structures (which were called user-defined types in Visual Basic 6.0 and earlier editions) and arrays. In this section, we'll examine these two data types.

Structures

A *structure* is a syntactic element that defines a value type, which is a type derived from System.ValueType. A structure defines a composite data type that must consist of at least one field or one event, but usually consists of multiple fields. When an instance of the value type defined by the structure is declared, each field is set equal to its default value. So in the case of a structure with two integers, each integer would be set equal to 0. For example, the following Structure statement defines a value type named Coordinates that consists of two fields of type Double:

```
Public Structure Coordinates
   Public x As Double
   Public y As Double
End Structure
```

When a variable of type Coordinates is instantiated, the values of both its x and y members are automatically set to 0.0.

Structures are declared using the Structure . . . End Structure construct. For details, see the entry for "Structure . . . End Structure" in Chapter 8. You can then work with the structure as a whole by referring to it by name, or you can work with individual members. The following code shows how you might work with the Coordinates structure:

```
Dim coord As Coordinates      ' The structure as a whole
coord.x = 10.032              ' x member
coord.y = 25.351              ' y member
Console.WriteLine(coord.ToString & ": x=" & coord.x & ", y=" & coord.y)
```

You declare an instance of a structure just as you do any variable, and you access a particular member of a structure by using dot notation, as shown in the preceding code.

Structures and Classes

In .NET, structures are enormously enhanced and have many of the characteristics of classes in previous versions of Visual Basic:

- Each field of a structure can have a different access level (Public, Private, and Friend are allowed, although Protected is not). For example, one field can be public, while another can be private.
- In addition to fields, structures can also have properties, methods, and events. A structure can also have a default property, as long as it is parameterized.
- A structure can implement an interface. However, a structure can't explicitly inherit from a class or from another structure. (All structures implicitly inherit from System.ValueType.)
- A structure can have either parameterized or parameterless constructors.
- A structure can raise an event.

Arrays

An *array* is a reference type that serves as a container for a set of data items that are stored together and are accessed as members of the array. Each element of the array can be identified by its index or ordinal position in the array.

Visual Basic allows you to define arrays in a number of ways. You can declare an array to simply be of type Array:

```
Dim arr As Array
```

This might be termed an untyped array. Each element of the array is capable of holding data of any type, and in fact an array of any type can be assigned to it. For example, the following code assigns an array of Integer to the array:

```
Dim arr As Array
Dim int() As Integer = {12, 16, 20, 24, 28, 32}
arr = CType(int, Array)
```

You can also define a "strong" array, which is an array that will hold data of a defined type. For example:

```
Dim byFours() As Integer = {12, 24, 36, 48}
Dim names() As String = {"Koani", "Saya", "Samuel", "Dakota", "Nikita"}
Dim miscData() As Object = {"this", 12d, 16ui, "a"c}
Dim objArray() As Object
```

Here, byFours is an Integer array; all of its members must be of type Integer, and only an Integer array can be assigned to it. Similarly, names is a String array; all of its members must be of type String, and only a String array can be assigned to it. Finally, miscData and objArray are arrays of type Object; their individual members can be any data type, although only arrays of reference types can be assigned to them. For example, the assignment

```
miscData = names
```

is legal, while

```
miscData = byFours
```

generates a compiler error ("Value of type '1-dimensional array of Integer' cannot be converted to '1-dimensional array of System.Object' because 'Integer' is not a reference type.").

All arrays have a particular number of dimensions and a particular number of elements in each dimension. Visual Basic supports arrays of up to 32 dimensions, although typical arrays have a single dimension. You define the number of dimensions of an array by indicating the number of members in that dimension. For example, the following code defines a number of one-dimensional arrays:

```
Dim myArray1a() As Integer        ' Uninitialized array
Dim myArray1b As Integer()        ' Uninitialized array
Dim myArray2(10) As Integer       ' 1-dimensional array with 11 elements
```

The following code, on the other hand, defines two two-dimensional arrays:

```
Dim Employees1( , ) As Object      ' Uninitialized 2-D array
Dim Employees(200, 2) As Object    ' 2-D array with 201 and 3 elements
```

The Lower Bounds of an Array

Visual Basic 6.0 allowed you to define a default lower bound for all arrays using the Option Base statement, and to define a default lower bound for a specific array by using syntax like

```
Dim ArrayStartsAt1(1 To 100) As String
```

In Visual Basic .NET, neither of these methods for defining the lower bound of an array is supported. (Visual Basic 2005 allows you to specify the lower bound of an array, but it must be 0.) However, although non-zero-based arrays are not CLS-compliant, you can explicitly define some other lower bound by calling one of the overloads of the CreateInstance method of the Array class. The following code does this:

```
' Changes lower bound to start at 1
Public Class CArray
   Public Shared Sub Main()
      Dim dims() As Integer = {10}      ' Number of elements in array
      Dim bnds() As Integer = {1}       ' Lower bound of array
      Dim Scores As Array = Array.CreateInstance(GetType(Integer), dims, bnds)
      Console.WriteLine(UBound(Scores))
      Console.WriteLine(LBound(Scores))
   End Sub
End Class
```

This version of the CreateInstance method has three parameters: a Type object representing the data type of the array, an array of Integer whose elements indicate the number of elements in each dimension (the first element represents the first dimension, and so on), and an array of Integer whose elements indicate the lower bound of each dimension (the first element represents the first dimension, and so on).

The first is an uninitialized two-dimensional array, and the second has 201 elements in the first and three elements in the second dimension.

Notice that the number of elements in each dimension is one greater than the value indicated in the array declaration, since .NET arrays are zero-based; the ordinal position of the first array element is 0, not 1. And the ordinal position of the last element in a dimension of the array is one less than the number of elements in that dimension. Actually, a useful way to think about this is not as an ordinal, but as an *offset*. The array index measures the distance of a particular element from the first element. Thus the element with an index of 1 is not the first element of the array but is one element away from the first element in the array. An array of ten elements has an upper bound of 9 because the last element is nine elements away from the first.

In addition to the standard rectangular or symmetrical arrays, Visual Basic supports jagged arrays—that is, arrays whose second dimension is non-rectangular. Basically, a jagged array is a one-dimensional array each of whose members also contains an array. (Jagged arrays, though, are not CLS-compliant, and so other .NET languages may not be able to handle them.) You can declare a jagged array with code like the following:

```
Dim jagged1(9)() As String
Dim jagged2()() As String = New String(9)() {}
```

All Visual Basic arrays are implicitly derived from the System.Array type in the .NET Framework Class Library. This means that, besides the standard Visual Basic syntax, you can also declare an array as an instance of the Array class, as the following declarations illustrate:

```
Dim myArray1a As Array
Dim myArray2a As Array = Array.CreateInstance(GetType(Integer), 10)
Dim members() As Integer = {3, 10}
Dim myArray3a As Array = Array.CreateInstance(GetType(Integer), members)
```

Because arrays are .NET classes, you can call the members of the .NET Array class when working with Visual Basic arrays. The more important of its instance members are shown in Table 2-3.

Unlike the pre-.NET versions of Visual Basic, which recognized both static and dynamic arrays, all .NET arrays are dynamic. That is, the number of elements in a given dimension can be expanded and contracted dynamically without your having to specify in advance that an array is dynamic. For example, using a dynamic array in Visual Basic 6.0 requires that you declare an uninitialized array, as the following code shows:

```
Dim arr() As Integer, index As Integer

index = 10
ReDim Preserve arr(index)
arr(index) = index ^ 2

index = 20
ReDim Preserve arr(index)
arr(index) = index ^ 2

index = 10
ReDim Preserve arr(index)
MsgBox arr(index)
```

Member	Description
Length property	Returns an Integer indicating the total number of elements in all dimensions of the array.
Rank property	Returns an Integer indicating the number of dimensions in the array.
Clear method	Reinitializes a portion of an array.
GetLength method	Returns an Integer indicating the number of elements in a particular dimension of an array. Its syntax is: *arrVar*.GetLength(*dimension*), where *dimension* is an Integer indicating the zero-based dimension of the array.
GetLowerBound method	Returns an Integer indicating the lower bound of an array. Its syntax is: *arrVar*.GetLowerBound(*dimension*), where *dimension* is an Integer indicating the zero-based dimension of the array.
GetUpperBound method	Returns an Integer indicating the upper bound of an array. Its syntax is: *arrVar*.GetUpperBound(*dimension*), where *dimension* is an Integer indicating the zero-based dimension of the array.
Reverse method	An overloaded method that reverses the order of all or a portion of a one-dimensional array.
Sort method	An overloaded method that sorts a one-dimensional array.

TABLE 2-3 Selected Members of the System.Array Class

In contrast, using Visual Basic .NET, there is no need to declare an uninitialized array. So the preceding VB6 code can be simplified as follows:

```
Dim index As Integer = 10
Dim arr(index) As Integer

arr(index) = CInt(index ^ 2)

index = 20
ReDim Preserve arr(index)
arr(index) = CInt(index ^ 2)

index = 10
ReDim Preserve arr(index)
MsgBox (arr(index))
```

Classes

A *class* is a template from which an object, or an instance of a class, is formed. A class defines a reference type that can be derived directly from some other class and that is derived implicitly from System.Object.

Classes are declared using the Class . . . End Class construct. For details, see the entry for "Class . . . End Class" in Chapter 8. The code between these statements can include the following:

- **Variable or field declarations** *Fields* are constants or variables that are exposed by the class (they can be publicly available to any client having a reference to the class instance as well as accessible to any client within the assembly in which the class instance resides or to any other class that is derived from the current class) or that are private to the class.
- **Properties** *Properties* are attributes of the class instance.
- **Methods** *Public* methods express the functionality that the class makes available, while *private* methods are for internal use within the class. Visual Basic .NET supports two kinds of methods: shared and instance. *Shared* methods do not require that an instance of the class be created in order to call the method (though in Visual Basic, shared methods can be called through class instances, even though this is nonsensical), while *instance* methods must be called through an instance of the class.
- **Constructors** A *constructor* is automatically invoked when a new instance of the class is created. In Visual Basic code, constructors are defined by methods named New.
- **Events** *Events* are notifications provided by a class, which is the event source. Event sinks register with the source and receive notification that the event has fired, which causes their event handlers to be invoked. The event itself can be anything of interest, such as a mouse click, a value exceeding or falling below a certain level, or a change to a data item.

In addition to classes defined in your own code, classes can be defined in external libraries that are referenced by your project, such as the .NET Framework Class Library.

Object variables that have been declared but not initialized are assigned a value of Nothing.

Visual Basic Naming Conventions

Visual Basic applies the same set of naming conventions throughout the language. Naming conventions apply to all language elements, including variables, types, and type members.

Visual Basic is a case-insensitive language. The Visual Basic editor attempts to enforce consistency in casing (it attempts to use the same case that was used in a type, member, or variable declaration), but case is ignored by the compiler. This means, of course, that you can't differentiate different program elements by case. Properties, for instance, cannot be assigned proper case names while local variables are assigned identical lowercase names.

- The first character of a name in Visual Basic must be an alphabetic character or an underscore. (Note that underscores are not CLS-compliant. They are often used internally to differentiate local variables from public properties with the same name but no underscore.)
- The remaining characters in a name can consist of alphabetic characters, numeric characters, or underscores.
- The name must not be greater than 1023 characters in length.
- Visual Basic recognizes a number of reserved words, such as Alias, Byte, Date, Default, Friend, Me, and Throw, that the compiler automatically interprets as language elements. If you want to use any of these reserved words to identify program elements in your code, you can indicate to the compiler that the name applies to your program element rather than the Visual Basic language element by enclosing it in brackets. This is called an *escaped name.* For example, to define a Boolean variable named Static (which conflicts with the Static keyword), you use the code:

```
Public [Static] As Boolean = False
```

Any escaped name must also be used together with its brackets subsequently to differentiate the program element from the reserved word. For example, a variable assignment using the Static variable takes the form:

```
[Static] = True
```

Variables in Visual Basic

In this section, we'll examine the rules regarding variable declaration and initialization. The section will also examine variable lifetime and scope.

Variable Declaration

Visual Basic allows you to declare variables of any data type. If you do not declare the precise data type of a variable, it will be of type Object. (This requires, however, that Option Strict be set off, its default value.)

Variables are defined using the Dim keyword or an access modifier. (For details on the access modifiers, see the section "Scope and Accessibility" later in this chapter.) For example,

```
Dim age As Integer
Dim city As String
Dim typ As System.Type
```

You can define multiple variables on the same line of code. So a somewhat unwieldy version of the preceding code is:

```
Dim age As Integer, city As String, typ As System.Type
```

If you define multiple variables on a line but do not specify a data type for each, a single type declaration applies to all of the variables defined to the left of it whose types are not specified. Fore example, the code

```
Dim city, state, zip As String
```

Variable Declaration and Option Explicit

Visual Basic does not actually require that you declare variables in advance. If Option Explicit is set off (by default, it is set on), Visual Basic will create variables dynamically as their names are encountered in code. This, however, is highly undesirable if for no other reason than that variables whose names are misspelled are treated as new variables. For example, the following code reports that an account balance is $30.98 when the correct balance is $254.73, because "balance" is misspelled "balaance" in one assignment statement:

```
Dim balance As Decimal = 200.35
balance -= 169.37
balaance += 223.75
Console.WriteLine("Your current balance is {0}.", FormatCurrency(balance))
```

defines three variables, all of type String. This differs from Visual Basic 6.0 and earlier versions, in which variables whose types were not explicitly declared were created as Variants. In the preceding line of code, for example, Visual Basic 6.0 would have interpreted city and state as Variants and zip as a String.

In Visual Basic .NET, unlike Visual Basic 6.0 and previous versions, you can initialize a variable at the same time as you declare it. For example:

```
Dim age As Integer = 35
Dim city As String = "New York"
Dim typ As System.Type = age.GetType
```

You can initialize multiple variables per line. However, if Option Strict is set on, each must have its own type declaration. For example:

```
Dim age As Integer = 35, weight As Integer = 185
```

You can also create a new class instance at the same time as you declare an object variable by using the New keyword in the declaration. For example:

```
Dim s As New StringBuilder("This is a brand new string.")
```

The New keyword causes the class or structure's constructor to be invoked. That means that you can pass arguments to any parameterized constructor, as in the preceding code fragment.

Variable Initialization

Visual Basic does not require that you explicitly initialize each variable that you declare. Instead, if no initialization is provided by the developer, a default initializer is used. However, it is important to keep in mind that, because class instances are set equal to Nothing if they are not explicitly initialized by the developer, code that expects a valid object reference can throw an exception. (The Visual Basic compiler, however, does produce a warning in this situation.)

Visual Basic and the Set Keyword

Visual Basic 6.0 and earlier versions required that you use the Set keyword when assigning a reference to an object variable. For example:

```
Dim fs As New FileSystemObject
Dim wndFold As Folder

Set wndFold = fs.GetSpecialFolder(WindowsFolder)
```

Since the Set keyword was not necessary in other variable assignments, this requirement was a frequent cause of confusion and error, particularly for inexperienced programmers. In .NET, the Set keyword's use in assigning object references has been eliminated. Set is now used exclusively within the Property . . . End Property construct to assign a value to a property.

Explicit assignment is simply a matter of using the assignment operator in a statement to assign the value of the left side of an expression to the variable on the right. For example:

```
Dim answer As String
answer = "The circumference is " & Circumference(radius) & "."
```

The one qualification is that Visual Basic automatically treats literals as instances of particular data types and generates a compiler error if Option Strict is on and it detects a narrowing conversion. For example, if Option Strict is on, the assignment

```
Dim ch1 As Char
ch1 = "c"
```

generates a compiler error ("Option Strict On disallows implicit conversions from 'String' to 'Char'."). To correct it, you have to explicitly convert the "c" to a Char, as in any of the following statements:

```
ch1 = "c"c
ch1 = CChar("c")
ch1 = Convert.ToChar("c")
```

Variable Scope

All types, type members, and variables have a *scope,* or a set of code to which they are visible without explicit qualification. Visual Basic recognizes five different levels of scope:

- **Global scope** An element is available to all code without qualification.
- **Namespace scope** An element is available to all code within the namespace in which it is declared. Types in a namespace are visible to all other types in the namespace without qualification. In addition, the Imports statement makes types in other namespaces available to program elements outside of the namespace.
- **Member scope** An element is available to all code within the module in which it is declared. A module can include modules defined with the Module . . . End Module construct, classes defined with the Class . . . End Class construct, and structures defined with the Structure . . . End Structure construct.
- **Procedure or local scope** An element is available to all code within the procedure in which it is declared. A procedure can include a function defined with the Function . . . End Function construct, a subroutine defined with the Sub . . . End Sub construct, or a property defined with the Property . . . End Property construct.
- **Block scope** An element is available to all code within the code block in which it is declared. Blocks are defined by the Do . . . Loop, For . . . Next, For Each . . . Next, If . . . End If, Select . . . End Select, SyncLock . . . End SyncLock, Try . . . End Try, While . . . End While, and With . . . End With constructs.

In general, a variable should have the smallest scope possible. This helps ensure the variable's integrity by reducing the number of places in which its value can be modified incorrectly or accidentally.

Scope and Accessibility

Visual Basic also includes four keywords (Public, Friend, Protected, and Private) that define the accessibility of a particular type or type member. Accessibility is related to but not identical with scope. *Accessibility* determines whether a program element is visible. *Scope* determines whether it must be qualified to be accessed. For example, a class defined with the Public keyword is globally accessible. However, it has only namespace scope, since it ordinarily can be accessed without using its namespace name only within its own namespace. Types and members with Friend accessibility are visible within the assembly in which they're declared. They have namespace scope to types in the same namespace in the assembly in which they are defined. But since a namespace can be split across multiple assemblies, they are completely inaccessible to types in the same namespace but a different assembly. Types with Protected accessibility have namespace scope to classes that inherit from them in the same namespace, must have their type names qualified to classes that inherit from them in different namespaces, and are not accessible to classes that do not inherit from them and to other types. Types declared with Private accessibility have member scope.

In the following sections, we'll examine each of these levels of scope.

Global Scope

In some cases, publicly available types and type members can be accessed without qualification regardless of the namespace in which they reside. For example, the "intrinsic" methods of classes in the Microsoft.VisualBasic namespace can be referenced without qualification anywhere in Visual Basic code. In Visual Basic, the Global keyword and HideModuleName statements can be used to give program elements global scope.

Namespace Scope

Variables and other elements that are declared as Public are accessible throughout all the types of a namespace, as well as all nested namespaces. If a namespace is not explicitly defined, they are available in all of the classes, modules, and structures in a project.

Member Scope

Member level refers to code that exists outside of any procedure in a class, structure, or module. Variables or other elements defined at member level are defined outside of any function, subroutine, or property belonging to a class, module, or structure. When declared with either the Private or Dim keyword, they are visible to all procedures within the module, but they are not available to any outside of the module.

Procedure Scope

Variables declared within a procedure (which includes a function, a subroutine, or a property procedure) but not inside a block construct are *local* variables. They have scope within that procedure only and are not visible outside of the procedure in which they are declared.

Dim is the only keyword that can be used to declare a variable with procedure scope. Any of the access modifiers (like Private or Protected) in a variable declaration with procedure scope generates a compiler error.

An advantage of local variables (as well as an occasional source of confusion) is that their names can duplicate the names of other variables, which will be effectively hidden by the local variable. The following code illustrates this:

```
Public Class Dog
   Private Name As String

Public Sub New(name As String)
   If name = String.Empty Then
      Dim message As String = "The dog breed must be specified."
      Throw New ArgumentException("message.")
   End If
   ' Parse into an array
      Dim words() As String = Split(name)
      For ctr As Integer = LBound(words, 1) To UBound(words, 1)
         Dim breedname As String = words(ctr)
         If Not Char.IsUpper(breedname, 1) Then
            Dim firstLetter As String = breedname.Substring(0, 1)
            firstLetter = firstLetter.ToUpper
            name = firstLetter & Mid(breedname,2)
            words(ctr) = breedname
         End If
      Next
      ' return to single string
      name = Join(words)
      Me.name = name
      Console.WriteLine("Created a new record for {0}.", Me.Breed)
   End Sub

   Public ReadOnly Property Breed As String
      Get
         Return Me.Name
      End Get
   End Property
End Class
```

This code declares a class named Dog that has a private variable, name, at the member level to store the value of the Breed property. Its class constructor also has a local variable, name, that is used to validate the string supplied as an argument. However, the two variables are quite independent of one another. Note that in the class constructor, the local name variable hides the private name variable. If we wish to access it, we must preface it with the Me keyword.

Block Scope

In .NET, variables declared inside of a block construct have block scope. That is, they are not visible outside of the block in which they are declared. Block constructs include the following:

- Do . . . Loop
- For . . . Next and For Each . . . Next
- If . . . End If
- Select . . . End Select
- SyncLock . . . End SyncLock
- Try . . . End Try

- While . . . End While
- With . . . End With

The following code declares a variable with block scope, firstChar, to store the first character of a name. This is then used to increment the count of an array element used to track the number of names that begin with that character. The code is

```
Public Class BlockScope
   Public Shared Sub Main()
      Dim maleNames() As String = {"John", "Michael", "Robert", "George", _
                                   "Lawrence", "Charles", "James", "Samuel", _
                                   "David", "Peter", "Paul", "Thomas", "Steven", _
                                   "Abraham", "Isaac", "Joseph", "Ronald", _
                                   "Theodore", "William", "Mark", "Luke"}
      Dim startChar(25) As Integer

      ' Find number of names beginning with a letter
      For Each maleName As String in maleNames
         Dim firstChar As Char
         firstChar = CChar(Left(maleName, 1))
         Dim charValue As Integer = AscW(firstChar)
         ' Go to next name if character invalid or unicode character out of range
         If charValue < 65 Or (charValue > 91 And charValue < 97) _
            Or charValue > 123 Then Continue For

         If Char.IsUpper(firstChar) Then
            startChar(Convert.ToInt32(firstChar) - 65) += 1
         ElseIf Char.IsLower(firstChar) Then
            startChar(Convert.ToInt32(startChar) - 97) += 1
         End If
      Next
      Console.WriteLine("Final Count:")
      Dim charCode As Integer = 65
      For Each charCount As Integer in startChar
         Console.WriteLine("Letter " & ChrW(charCode) & " has " & charCount & _
                           " names.")
         charCode += 1
      Next
   End Sub
End Class
```

Dim is the only keyword that can be used to declare a variable with block scope. Using any of the access modifiers (like Private or Protected) in a variable declaration with block scope generates a compiler error.

Visual Basic and Block Scope

Block scope is new to Visual Basic .NET. In Visual Basic 6.0 and earlier versions, variables declared inside a block construct had procedure-level scope; they were visible from the point that they were declared throughout the rest of the procedure.

The lifetime of variables with block scope (see the next section, "Variable Lifetime") is determined by the procedure in which they are declared. This is illustrated by the following code, in which the value of one of the variables with block scope, blockVar, reaches 110,000 in the last iteration of the inner loop:

```
Public Class BlockScope
   Public Shared Sub Main()
      For outerLoop As Integer = 0 to 10000
         For innerLoop As Integer = 0 to 10
            Dim blockVar As Integer
            blockVar += 1
            If blockVar Mod 1000 = 0 Then Console.WriteLine(blockVar)
         Next
      Next
   End Sub
End Class
```

Variable Lifetime

In addition to their scope, variables also have a lifetime, which indicates when during program execution the variable exists. Scope, on the other hand, indicates whether the variables that exist are accessible or not. The difference is most clear in variables with block-level scope: although they are visible only within the block construct within which they are declared, their lifetime is the lifetime of the procedure in which they are declared. Similarly, the lifetime of variables with procedure scope terminates when a procedure ends, but they are visible only when the procedure is actually executing, not when the procedure has in turn called other procedures.

Ordinarily, the lifetime of variables with procedure scope terminates when the procedure ends. However, we can declare variables to be static, in which case their lifetime is the duration of the application, even though they have procedure scope. For example, the following code shows a simple Windows application that tracks the number of times its single interface object, a command button, is clicked:

```
Public Class Form1
    Private Sub cmdClickCtr_Click(ByVal sender As System.Object, _
                                  ByVal e As System.EventArgs) _
                                  Handles cmdClickCtr.Click
        Static ctr As Integer = 1
        MsgBox("Button was clicked " & ctr & " times.")
        ctr += 1
    End Sub
End Class
```

Static in Visual Basic and C#

A possible source of confusion is the use of the static keyword in Visual Basic .NET and C#. In C#, a variable or another program element marked with the static keyword belongs to the type in which it is declared, rather than to an instance of the type. It corresponds to the Shared keyword rather than the Static keyword in Visual Basic .NET.

Ordinarily, the lifetime of the counter variable, ctr, is limited to the duration that the Click event procedure runs. Declaring it as static, however, causes it to exist throughout the lifetime of the application. Hence, it can track the total number of times the button is clicked since the application started. Note that the Static variable declaration, which also initializes the variable ctr to a value of 1, is executed only once. We do not need to be concerned that the value of ctr will be reset to 1 each time the event procedure executes.

Late and Early Binding

Binding refers to the process of member lookup and type checking. Binding can be handled by the compiler in advance at compile time, in which case it is referred to as *early binding*. It can also be handled at runtime by the .NET runtime environment, in which case it is referred to as *late binding*. Early binding occurs whenever an object variable is declared to be a specific object type. In that case, the compiler can perform all necessary type checking in advance.

Late binding occurs whenever any variable is declared to be of type Object; a more specific type of object is subsequently assigned to it, and the Object variable is used to call a member of that more specific object type. In this case, name lookup and type checking must occur at runtime rather than at compile time, which results in a significant performance penalty. Since type checking and member lookup occur only at runtime, it also makes runtime errors more likely, thereby increasing the need for robust error handling.

Reference and Value Type Conversion

In our earlier discussion of data types, we noted that Visual Basic's data types include value types and reference types. The key difference between reference types and value types is that variables hold references to instances of reference types, while they hold copies of instances of value types. In many ways, the difference is the same as the difference between the ByVal and ByRef keywords: when you pass an argument by reference, you're passing the argument itself. When you pass an argument by value, you're passing a copy of your original. So whenever you pass an instance of reference type, the CLR is passing its reference. Whenever you're passing a value type, the CLR is making a copy of the value type and passing that.

This copy operation actually takes two forms:

- When a value type is assigned to a field or variable of the same type, a copy is made.
- When a value type is assigned to a field or variable of type Object, a copy is made, and a process called *boxing* is performed by the CLR. This consists of wrapping a special "box" around that copy. The box is simply a "thin" reference wrapper that holds onto the boxed value so that the called routine that is expecting an instance of a reference type gets one.

Boxing involves a performance penalty. And often, it's a double whammy. When a boxed value is then assigned to a variable or field typed as a value type, the value must be unboxed. This means that another copy has to be made, just as if the value had never been boxed at all.

Visual Basic can perform the process of boxing and unboxing automatically, without the need for the developer to explicitly handle the conversion of value types to reference types and vice versa. However, Visual Basic's automatic boxing and unboxing involves a performance penalty.

There are two major ways to minimize the possible impact on performance that results from Visual Basic's automatic boxing and unboxing:

- Use instances of classes instead of value types to prevent boxing and unboxing. For example, a routine stores instances of a structure to a Visual Basic Collection object. The Collection class expects that each data item assigned to the collection is of type Object. Hence, some performance improvement would result from replacing the structure with a class.
- Rather than allowing Visual Basic to handle unboxing automatically, use the DirectCast operator so that your code handles unboxing. When handling primitive value types (rather than complex value types, such as structures), this saves Visual Basic from having to determine what conversions are possible before performing the unboxing. Instead, DirectCast will only allow you to convert a reference type to the data type of the original value type.