CHAPTER 1

VB 2005 Overview

This book is for experienced object-oriented developers. In this overview, we will take a look at some of the major language differences between Visual Basic 2005 (VB 2005), C#, and Visual Basic 6.0 (VB6). Functionally, VB 2005 and C# are nearly identical, and you can use either language to create stable, high-performance applications in the .NET environment. You'll see the biggest differences in the syntax, which is completely different between the two languages. Needless to say, VB 2005 and VB6 are vastly different from each other, and we will look at some of the overreaching differences between the two languages. Next, we will review a simple VB 2005 program to get an idea of the programmatic structure in .NET and wrap up with a summary of what's new for current VB programmers in this latest and greatest version, VB 2005.

Differences Between VB 2005, C#, and VB6

This new version of VB has been specifically designed to target the new programming model provided by .NET 2.0. Both C# and VB 2005 are designed to write programs that work with the .NET runtime. Whereas C# was designed with C and C++ programmers in mind, VB 2005 was designed to target the large base of existing VB programmers. The new language targets the .NET programming model and is derived from previous versions of VB, but you'll find that it's vastly different. The language changes are due to the fact that in order to support the framework, VB 2005 must provide more object-oriented features as well as type safety.

.NET Runtime

To understand VB development in the .NET environment, you first need to understand some components of the .NET environment and how they interact. This section summarizes how VB programs are compiled and run in .NET. The execution engine of .NET is known as the common language runtime (CLR). The CLR is primarily responsible for loading and executing code and also handles memory management, security, and handling of types.

At the top level is the VB language itself, or any language that targets the CLR, that is used to create code. The VB compiler takes the written code and generates intermediate language (IL). For example, a DLL or EXE contains IL that is understood by the CLR. Any code written to run in the CLR is known as *managed code*, because it runs under the control of the CLR. Managed code is an IL because it is halfway between the high-level language (VB) and the lowest-level language (assembly/machine code).

At run time, the CLR compiles the IL into native code on the fly by using the just-in-time (JIT) compiler. The JIT compiler creates native code that is CPU-specific, so you could take the IL for a program and compile it on computers with different architectures. JIT compiling comes with its pros and cons. It may seem that an obvious disadvantage is the inefficiency of compiling code at run time. However, the JIT compiler doesn't convert all the IL into native code; rather, it converts only the code that's executing into native code to be run. At the same time, it creates stubs to any methods not executing so that when those methods are called, the JIT compiler will compile and execute the necessary code.

An advantage of JIT compiling is that the working set of the application is reduced because the memory footprint of intermediate code is smaller. During the execution of the application, only the needed code is JIT-compiled. Unused code, such as printing code if the user never prints a document, is never JIT-compiled. Moreover, the CLR can optimize the program's execution on the fly at run time. For example, on the Windows platform, the CLR may determine a way to reduce page faults in the memory manager by rearranging compiled code in memory, and it could do all this at run time. That said, there are times when JIT compilation can be a performance drawback. In this case, you can use native image generation (NGen) to precompile IL on the machine where it's running.

As you can see, the CLR replaces the traditional VB runtime and also eliminates the use of COM, DCOM, MTS, or COM+. VB applications now run in the context of the CLR, so there's no more need for the host of distributed technologies that were once so prevalent. Of course, you can still access COM components if you need to through the interop layer provided by .NET.

■ **Note** If you wish, you could actually code a program in raw IL while building it with the Microsoft Intermediate Language (MSIL) Assembler. Of course, that would be a very inefficient way to write programs, but the fact that you could do it demonstrates the cross-platform capabilities of .NET. You can convert any language that supports .NET into IL, and the CLR can understand IL from any language. From there, you can JIT-compiled the IL into native code for the particular CPU architecture on which it's running.

VB 2005 and C#

VB 2005 and C# are nearly identical in what you can accomplish with them; you can use either language to access all the classes and functions provided by the .NET Framework. Essentially, you can do everything in VB 2005 that you can do in C#, although one language may provide a more streamlined approach than the other, depending on what you're trying to accomplish.

When discussing VB 2005 and C#, it's easier to talk about their differences than their similarities. In addition to the long-awaited edit-and-continue, the release of VB 2005 brought some new features that were previously available only in C#, such as the use of generics, the `Using` statement, and operator overloading. Later chapters cover each of these topics extensively. The only real differences are that currently C# provides the ability to write unsafe code, and VB 2005 provides late binding. Unsafe code in C# is that which uses pointers to manage

and access memory directly. You might need to use unsafe code for performance reasons or to write low-level Win32 API calls, to the `ReadFile` function, for example. Unsafe code may be warranted at times, but it isn't recommended because it cannot be verified to be safe and creates objects in memory that cannot be removed by garbage collection. Late binding was kept in the VB language to maintain compatibility with previous versions. Late binding allows you to create a variable as a type of `Object` and then later assign it to a variable either implicitly by assigning it to an object instance or using the `CreateObject` function. Early binding is preferred over late binding because late binding incurs a performance hit, and the compiler can't report errors at compilation time, which means you'll receive them at run time instead.

VB 2005 and VB6

The Visual Basic language was completely overhauled to support the CLR, so it bears little resemblance to VB6. This results in a steep learning curve for many developers, but the trade-off is a whole new programming model that, when applied correctly, provides a better development environment and better software. In addition to a whole new compilation model, the CLR provides improved memory management, an object-oriented environment, and type safety. One of the biggest changes to the VB language is that it is now truly object-oriented. This means that each and every object (including data types) derive from `System.Object`. Instead of the VB runtime and the Win32 API, you now have the entire Base Class Library (BCL) of objects to work with. The challenge for programmers new to .NET is navigating this vast library and knowing where to find the classes they need. Becoming familiar with the .NET Framework will make you a better and faster programmer.

CLR Garbage Collection

One of the key facilities in the CLR is the garbage collector (GC). In the managed runtime environment, the GC heap is responsible for managing all objects. It monitors an object's lifetime and frees it from memory when no part of the program references the object. The GC doesn't remove an object from memory as soon as there are no references to it. Rather, it runs periodically and releases objects from memory when necessary. There may be a delay from the time that all references to an object are released to the time that it's destroyed by the GC. Objects have a `Finalize` destructor (it's implicitly created at run time if you don't have it explicitly defined) that's called by the GC. However, the `Finalize` destructor doesn't fire immediately when an object loses scope. The automatic nature of the GC means that object lifetimes are nondeterministic in the .NET environment.

The GC does not remove all resource-handling burdens from your plate. For example, a file handle is a resource that must be freed when the consumer is finished with it. The GC only handles memory resources directly. To handle resources other than memory, such as database connections and file handles, you can use the `Finalize` destructor to free resources when the GC notifies you that your object is being destroyed. You can also implement the `IDisposable` interface in your classes to release resources immediately. Chapter 4 covers both of these topics in more detail.

Common Type System

In order to support multiple programming languages, the CLR implements the common type system (CTS) to ensure that data types in each language mean the same thing and are handled in the same way. The CTS means that a variable defined as `Int32` in VB is the same as an `Int32` variable in C# or COBOL.NET. The CTS provides a framework for all type definitions that ensures consistency and type safety in each .NET language. Another benefit of the CTS is that it provides an object-oriented model for type definition so that all types are handled as objects.

The CTS contains two categories of types: value types and reference types. Value types are often referred to as primitive or built-in types and directly contain their data. Examples of value types are `Integer`, `Boolean`, and `Float`. An `Enum` is another kind of value type, and there are also user-defined value types. Value types are very efficient and take up little memory. A value type variable always has a value. When value type variables are passed in memory, the actual value of the variable is passed. Here's an example that illustrates the nature of value types:

```
Public Class EntryPoint
    Shared Sub main()
        Dim Value1 As Integer = 0
        Dim Value2 As Integer = Value1

        Value2 = 123

        Console.WriteLine("Values:  {0}, {1}", Value1.ToString, Value2.ToString)
    End Sub
End Class
```

Here's the output from the preceding example:

```
Values:  0, 123
```

This demonstrates that you can set the value of a value type to that of another value type, but no reference exists between the two types. On the other hand, reference types store a reference to the memory address of its value, which is an instance of the reference type. For example, let's say you declare a variable of type `Dataset` with the command `Dim ds As New DataSet`. The value of `ds` is actually a pointer (or reference) to the dataset that resides somewhere in memory. Reference type variables are not tied to their value and can have a null reference. When you pass a reference type—let's use the `ds` variable as an example—a copy is made of the reference to the dataset; the entire dataset is not copied. If the `ds` variable is passed into a function and a change is made to it, that change would be seen in the dataset in the calling code as well. This snippet of code illustrates the nature of reference types:

```
Class Class1
    Public Value As Integer = 0
End Class

Public Class entrypoint
```

```
    Shared Sub main()
        Dim Reference1 As New Class1()
        Dim Reference2 As Class1 = Reference1

        Reference2.Value = 123

        Console.WriteLine("Values:  {0}, {1}", Reference1.Value, Reference2.Value)
    End Sub
End Class
```

Here's the output from the previous code:

```
Values:  123, 123
```

The value of the variable `Reference2` is a reference to variable `Reference1` (an instance of `Class1`). So any changes to `Reference2` will be reflected in `Reference1`.

A Simple VB 2005 Program

Now let's take a look at a VB 2005 program from 50,000 feet and consider the ubiquitous Hello World! program. Here's what the code and output looks like:

```
Public Class EntryPoint
    Shared Sub Main()
        System.Console.WriteLine("Hello World!")
    End Sub
End Class
```

```
Hello World!
```

Note the structure of the program. It declares a type (a class named `EntryPoint`) and a member of that type (a method named `Main`). `Main` simply calls `Console.WriteLine` to display "Hello World!" in a Command Prompt window. When you run this program in debug mode, the compiler creates a `HelloWorld.exe` in the `\obj\Debug` directory of your project.

Every program requires an entry point, and in the case of VB, it is usually the `Main` method. You declare the `Main` method inside of a class (in this case, named `EntryPoint`). The return type for the `Main` method is an optional `Integer`. In the example, `Main` has no parameters, but if you need access to the command-line parameters, you can declare a parameter (an array of strings) to access them.

To illustrate VB's platform independence, if you happen to have a Linux OS running (with the Mono VES installed on it), you can copy this `HelloWorld.exe` directly over in its binary form and it will run as expected, assuming everything is set up correctly on the Linux box.

What's New in VB 2005

The latest version of VB features a host of enhancements and new features for VB programmers. These include improvements to the .NET Framework, Visual Studio Integrated Development Environment (IDE) improvements, and changes to the VB language itself. Specific enhancements to the VB language that will be of interest to programmers include new language keywords, generics, operator overloading, and the `My` namespace.

New Commands

The VB language contains three new object-related commands that can simplify coding: `Global`, `Using`, and `Continue`. The `Global` keyword is used to access namespaces that may be out of reach due to namespaces created with the same name. For example, if you have a namespace called `Common.System.Collections` in your project and you're coding from within that namespace, you wouldn't be able to define an `ArrayList` object from the `System.Collections` namespace in the BCL. To define an `ArrayList` object, you would reference the .NET namespace with the `Global` keyword like this: `Global.System.Collections.ArrayList`.

Object lifetimes are nondeterministic because it's unknown when the garbage collector will release resources once an object is no longer referenced. To do its job, the garbage collector calls a finalizer to free up resources. To explicitly release resources in a class, you can create the `Dispose` method and call it to handle any cleanup. The downside is that you must call the `Dispose` method in any end point in your object. Instead of having to call the `Dispose` method explicitly within an object, the `Using` keyword specifies that the `Dispose` method should be called automatically at the end of the `Using` code block. You can implement `Using` for any class that implements the `IDisposable` interface. Here's what the syntax looks like:

```
Dim ds As New DataSet

Using (ds)
            ...
End Using
```

With the `Continue` command, you can skip to the next iteration of a loop without processing the rest of the loop body. For example, in the following code, if the remainder of `Counter` divided by 2 is 0, then execution goes back to the `For` line. If the remainder of `Counter` divided by 2 is not 0, then the `Counter` is added to the `Total` and then normal execution goes to the `For` line. The `Continue` command can be used for `Do`, `While`, and `For` looping constructs (`Continue Do`, `Continue While`, and `Continue For`):

```
Public Class EntryPoint
    Shared Sub Main()
        Dim Counter As Short
        Dim Total As Short

        Total = 0
        For Counter = 0 To 10
            If Counter Mod 2 = 0 Then Continue For
```

```
            Total += Counter

            System.Console.WriteLine("The total is now: {0}", Total.ToString)
        Next
    End Sub
End Class
```

Here are the results:

```
The total is now: 1
The total is now: 4
The total is now: 9
The total is now: 16
The total is now: 25
```

Generics

Generics allow you to create methods, classes, structures, or interfaces in which you define the specific data type that is managed. A class is defined as generic with this syntax:

```
Public Class Class1(Of Type)
    Public MyType As Type
End Class
```

The `Of Type` syntax is how the class type is made generic. This example uses the word `Type`, but you can replace `Type` with any other word. When you actually create an object of type `Class1`, you specify the precise type that it can manage like this:

```
Dim x As New Class1(Of String)

x.MyType = "StringVal"
```

In this example, you could specify `Of Int32` or `Of Customer`. Then the `MyType` method could only be set to an `Int32` value or a `Customer` object. The namespace `System.Collections.Generics` provides classes such as `Dictionary`, `List`, `Queue`, `SortedDictionary`, and `Stack` that you can use to make your own type-safe collection classes. For instance, the `Dictionary` class allows you to create a `Dictionary` object that stores only objects of a specific data type. The code to do this looks like this:

```
Dim Customers As New Dictionary(Of Int32, Customer)
```

This specifies that the `Customers` dictionary object takes only keys of data type `Int32` and stores values only of data type `Customer`. In this case, `Dictionary` only stores `Customer` objects. Attempting to store any other type of data will result in a compile-time error. The topic of generics has such a broad scope in the VB language that Chapter 13 has been devoted to the topic.

Operator Overloading

Operator overloading is used to define how you can use classes with operators. The standard operators in VB include, but aren't limited to, (+, -, <, >, =, and <>). When used with integer values, the + operator performs an arithmetic operation:

```
Dim i As Int32

i = 28 + 108
```

The variable `i` is equal to the sum of 28 and 108. When used with string values, the + operator performs concatenation and creates a new string value:

```
Dim str1 As String

 str1 = "Hello " + "World!"
```

Operator overloading allows you to define how an operator will work with your own classes. For example, let's say you have an `Account` class. It could be a bank account or a credit card account, but each `Account` has a balance. You can implement the standard operators to be applied to the account balance whenever they are used with `Account` objects. This would add two account balances:

```
Dim i As Double

i = Account1 + Account2
```

The topic of operator overloading is an extensive one, so Chapter 8 provides more detail on how to implement this new feature.

My Namespace

Most of your development time is probably spent researching and navigating the .NET BCL to find the classes and methods you need to do something. The BCL contains all the classes you need for most application plumbing tasks, such as variable type definitions, collections, file I/O, data encryption, local computer hardware access, and network access, to name a few. Needless to say, to find the class that you need to get the current username, for example, can be a daunting task. To make finding these namespaces and classes a little easier, the `My` namespaces were added to VB.

Each namespace focuses on a particular area of functionality, and those namespaces are `My.Application`, `My.Computer`, `My.Forms`, `My.Resources`, `My.Settings`, `My.User`, and `My.WebServices`. To get to these namespaces, simply enter `My` in the VB IDE and look at the IntelliSense options. The naming of the `My` namespaces gives you a pretty good idea of what kinds of classes are contained in each one of them.

Summary

In this chapter, we've touched upon the high-level characteristics of programs written in VB. That is, all code is compiled into IL rather than the native instructions for a specific platform. Additionally, the CLR implements a GC to manage raw memory allocation and deallocation, freeing you from having to worry about one of the most common errors in software development: improper memory management.

Next, we explored the CTS with a couple of simple examples to compare value types to reference types. Our first VB program gave us a simple class with a `Main` procedure, which simply echoed "Hello World!" to the Command Prompt.

Finally, we looked at some of the new features that are now a part of VB, including the `Global`, `Using`, and `Continue` commands, as well as generics, operator overloading, and the `My` namespace.

In the next chapter, we'll dive into the details of the CLR, how compilation works, and how to program assemblies in VB.