Accelerated VB 2008

Copyright © 2008 by Guy Fouché and Trey Nash

ISBN-13 (pbk): 978-1-59059-874-0

ISBN-10 (pbk): 1-59059-874-1

ISBN-13 (electronic): 978-1-4302-0339-1

ISBN-10 (electronic): 1-4302-0339-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Dominic Shakeshaft
Technical Reviewers: Tim Patrick, Fabio Claudio Ferracchiati
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Kevin Goff, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Copy Editor: Kim Benbow
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Lynn L'Heureux
Proofreader: Elizabeth Welch
Indexer: Present Day Indexing
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski
Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

VB 2008 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 2008 (VB 2008), C# 3.0, and Visual Basic 6.0 (VB6). Functionally, VB 2008 and C# 3.0 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. VB 2008 and VB6 are also 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 2008 program to get an idea of the programmatic structure in .NET 3.5 and wrap up with a summary of what's new for current VB programmers in this latest and greatest version, VB 2008.

Differences Between VB 2008, C# 3.0, and VB6

This new version of VB has been specifically designed to target the new programming model provided by .NET 3.5. Both C# 3.0 and VB 2008 are designed to write programs that work with the .NET runtime. Whereas C# 3.0 was designed with C and C++ programmers in mind, VB 2008 is designed to be more accessible to the large base of existing VB programmers. The new language targets the .NET 3.5 programming model and is derived from previous versions of VB, but you'll find that it's quite different. The language changes are due to the fact that in order to support the framework, VB 2008 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, as well as 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.

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 2008 and C# 3.0

VB 2008 and C# 3.0 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 2008 that you can do in C# 3.0, although one language may provide a more streamlined approach than the other, depending on what you're trying to accomplish.

When discussing VB 2008 and C# 3.0, it's easier to talk about their differences than their similarities. In the latest release, both languages add new language extensions, including Language-Integrated Query (LINQ), query comprehensions, anonymous types, lambda expressions, and extension methods. Later chapters cover each of these topics extensively.

The only real differences are that currently C# 3.0 provides the ability to write unsafe code, and VB 2008 provides late binding. Unsafe code in C# 3.0 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 2008 and VB6

The Visual Basic language, beginning with VB .NET 2002, was completely overhauled to support the CLR, so it bears only a passing 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. The BCL is a subset of the .NET Framework and provides types that can be utilized by any code written to run in the CLR. The BCL includes the `System.Collections`, `System.Diagnostics`, `System.IO`, `System.Registry`, `System.Globalization`, `System.Reflection`, `System.Text`, and `System.Drawing` namespaces, to name a few.

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 a 32-bit signed integer (`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 primitives 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 that declares a set of related integer values, 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 2008 Program

Now let's take a look at a VB 2008 program from 50,000 feet and consider the ubiquitous Hello World! program. Here's what the code and output look 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` 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 command-line parameters, you can retrieve them via the `My.Application.CommandLineArgs.Item method`. The `My` namespace makes it easier to find classes you will use often for application tasks, file I/O, local computer hardware access, and network access to name a few. A few of the `My` namespaces include `My.Application`, `My.Computer`, `My.Forms`, `My.Settings`, and `My.User`. 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 2008

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 query comprehensions, LINQ support, anonymous types, lambda expressions, and extension methods.

Query Comprehensions

Query comprehensions are very similar to SQL queries in syntax. They are comprised of the familiar `Select`, `From`, and `Where` clauses, named *query operators*. You combine query operator clauses to form *query expressions*. These expressions can then be used to return a data set from various sources, such as XML and collections. An example statement would look like this:

```
Dim SmallCapStocks = From Stock In AllStocks _
    Where Stock.Price < 10.0 _
    Select Stock
```

This code snippet will create `SmallCapStocks` and populate it with stocks priced less than $10.00 from `AllStocks`.

Implicitly Typed Local Variables

Implicitly typed local variables provide a shortcut in declaring your variables. In VB 2005, you would initialize variables in one of the following ways:

```
    Dim CompanyName As String = "ABC Company"
    Dim OutstandingShares As Integer = 10000
    Dim Capitalization As Double = 3000000.0
```

In VB 2008, these can now be written in this manner:

```
    Dim CompanyName = "ABC Company"
    Dim OutstandingShares = 10000
    Dim Capitalization = 3000000.0
```

Declaring local variables this way causes the type of each variable to be inferred from the right-hand expression. In earlier releases of VB, such statements would have caused these variables to be declared as the generic `Object` data type, but VB 2008 now enforces strong data types on these declarations.

Object Initializers

Object initializers allow combining creation of an object and the initialization of its fields into one statement. Consider the following code, written in VB 2005:

```
        Dim WidgetCo As New Stock

        With WidgetCo
            .Ticker = "WC"
            .Name = "Widget Corp."
            .Price = 25.0
        End With
```

In VB 2008, this can now be written like this:

```
        Dim WidgetCo = New Stock With { _
            .Ticker = "WC", _
            .Name = "Widget Corp.", _
            .Price = 25.0 _
        }
```

Array Initializers

Array initializer expressions can create and initialize an array and infer its element types at the point of declaration. This example creates an array of U.S. Stocks and populates it with three new `Stock` elements:

```
    Dim USStocks() = { _
        New Stock With { _
            .Ticker = "US1", _
            .Name = "Western Company", _
            .Price = 15.75}, _
        New Stock With { _
            .Ticker = "US2", _
            .Name = "Eastern Company", _
            .Price = 17.0}, _
        New Stock With { _
            .Ticker = "US3", _
            .Name = "Midwest Company", _
            .Price = 8.0} _
    }
```

LINQ to XML

LINQ to XML is an in-memory API that allows you to read, write, and create XML. XML can be treated as a built-in data type in VB 2008. You can also create XML using *XML Literals* and query *XML documents* using *XML Properties*.

A simple XML document can be created with the following code snippet:

```
Dim Stocks As XElement = _
    <Stocks>
        <Stock StockID="1">
            <Ticker>S1</Ticker>
            <Company>Company 1</Company>
            <PriceQuote>25.00</PriceQuote>
        </Stock>
        <Stock StockID="2">
            <Ticker>S2</Ticker>
            <Company>Company 2</Company>
            <PriceQuote>8.00</PriceQuote>
        </Stock>
    </Stocks>
```

LINQ to Objects

LINQ to Objects allows you execute SQL-like queries over your arrays and collections. Executing these queries, you are able to filter your collection for more specific processing. The following example executes a simple query against an array of integers:

```
Imports System
Imports System.Linq

Public Class EntryPoint
    Shared Sub Main()
        Dim someNumbers As Integer() = {5, 4, 3, 2, 1, 0}

        Dim query = From x In someNumbers _
                Where x >= 3 _
                Select x

        For Each item In query
            Console.WriteLine("{0} is >= 3", item)
        Next
    End Sub
```

This code filters someNumbers to create query, and query will contain the three integers, which are those greater than or equal to 3. The For Each . . . Next statement will loop through query and produce the following results:

```
5 is >= 3
4 is >= 3
3 is >= 3
```

Lambda Expressions (Inline Functions)

Lambda expressions provide you with a way to create inline, anonymous methods. The functions you create do not need to be typed, as they infer their types at run time. Creating a VB method to determine even numbers might look like the following:

```
   Shared Function IsEven(ByVal x As Integer) As Boolean
      If x Mod 2 = 0 Then
         Return True
      Else
         Return False
      End If
   End Function
```

Using an inline function, you can reduce the amount of code needed to accomplish the same purpose, as in the following code snippet:

```
Function(n) n Mod 2 = 0
```

Extension Methods

Extension methods allow you to add methods to an existing CLR type. This enables you to expand the functionality of a type without needing to create a subclass. The following example shows the syntax for an extension method:

```
<Extension()> _
Public Function IsTickerValid(ByVal aTicker As String) As Boolean
    Dim ValidTicker As Boolean = True

    If aTicker.Length > 4 Then
        ValidTicker = False
    End If

    Return ValidTicker
End Function
```

Extension methods avoid the need to create a separate class, with a shared method, to validate the ticker symbol.

Anonymous Types

Using anonymous types, you are able to declare, define, and use types that the VB compiler will generate for you in the background. Anonymous types are able to infer field names and will create properties for these fields as well. This example shows an anonymous type declared in VB:

```
Dim anAnonymous = New With {.FirstName = "Jodi", .LastName = "Fouche"}
```

IntelliSense Everywhere

When using extension methods and anonymous types, you will find that the VB IDE is aware of these types and methods. As such, it includes your extension methods with the extended type's methods in all IntelliSense lists while coding. Anonymous types will show their compiler-generated properties in IntelliSense lists while coding.

Nullable Types, Enhanced in VB 2008

Nullable type syntax is enhanced in VB 2008. A new type modifier (?) allows you to define your types as nullable. The following code snippet demonstrates the current syntax and the new shortcut.

```
Dim x As Nullable(Of Integer)
Dim y As Nullable(Of Integer)

Dim x As Integer?
Dim y As Integer?
```

Relaxed Delegates, Enhanced in VB 2008

Relaxed delegates extend and enhance the VB's implicit conversions to delegate types. Using relaxed delegates allows you to omit arguments from event handlers. For example, the following code statements are now valid in VB:

```
Sub OnClick(sender As Object, e As Object) Handles aButton.Click
Sub OnClick Handles aButton.Click
```

Option Infer

Using `Option Infer` lets you specify whether VB should enforce type declarations for your types. While setting `Option Infer On` may allow for quicker development, it may lead to maintenance challenges down the road.

`Option Infer On` is the default setting for new projects in VB 2008 and can be changed via Project Properties ➤ Compile.

Summary

In this chapter, we've touched upon the high-level characteristics of programs written in VB 2008. That is, all code is compiled into IL rather than the native instructions for a specific platform. Additionally, the CLR implements a garbage collection system to manage raw memory allocation and de-allocation, 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 LINQ, object and array initializers, extension methods, anonymous types, and `Option Infer`.

In the next chapter, we'll dive into VB syntax, explore VB namespaces, and discuss control flow.