

Pro VB 2005 and the .NET 2.0 Platform

Second Edition



Andrew Troelsen

Pro VB 2005 and the .NET 2.0 Platform

Copyright © 2006 by Andrew Troelsen

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-578-7

ISBN-10 (pbk): 1-59059-578-5

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Ewan Buckingham

Technical Reviewer: Don Reamey

Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Kier Thomas, Matt Wade

Production Director and Project Manager: Grace Wong

Copy Edit Manager: Nicole LeClerc

Senior Copy Editor: Ami Knox

Assistant Production Director: Kari Brooks-Copony

Production Editor: Kelly Winquist

Compositor and Artist: Kinetic Publishing Services, LLC

Proofreaders: April Eddy, Lori Bring, Nancy Sixsmith

Indexer: Broccoli Information Management

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 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.



VB 2005 Programming Constructs, Part I

This chapter begins your formal investigation of the Visual Basic 2005 programming language. Do be aware this chapter and the next will present a number of bite-sized stand-alone topics you must be comfortable with as you explore the .NET Framework. Unlike the remaining chapters in this text, there is no overriding theme in this part beyond examining the core syntactical features of VB 2005.

This being said, the first order of business is to understand the role of the `Module` type as well as the format of a program's entry point: the `Main()` method. Next, you will investigate the intrinsic VB 2005 data types (and their equivalent types in the `System` namespace) as well as various data type conversion routines. We wrap up by examining the set of operators, iteration constructs, and decision constructs used to build valid code statements.

The Role of the Module Type

Visual Basic 2005 supports a specific programming construct termed a *Module*. For example, when you create a console application using Visual Studio 2005, you automatically receive a *.vb file that contains the following code:

```
Module Module1
    Sub Main()
    End Sub
End Module
```

Under the hood, a `Module` is actually nothing more than a class type, with a few notable exceptions. First and foremost, any public function, subroutine, or member variable defined within the scope of a module is exposed as a “shared member” that is directly accessible throughout an application. Simply put, *shared members* allow us to simulate a global scope within your application that is roughly analogous to the functionality of a VB 6.0 *.bas file (full details on shared members can be found in Chapter 5).

Given that members in a `Module` type are directly accessible, you are not required to prefix the module's name when accessing its contents. To illustrate working with modules, create a new console application project (named `FunWithModules`) and update your initial `Module` type as follows:

```
Module Module1
    Sub Main()
        ' Show banner.
        DisplayBanner()
    End Sub
End Module
```

```

    ' Get user name and say howdy.
    GreetUser()
End Sub

Sub DisplayBanner()
    ' Pick your color of choice for the console text.
    Console.ForegroundColor = ConsoleColor.Yellow
    Console.WriteLine("***** Welcome to FunWithModules *****")
    Console.WriteLine("This simple program illustrates the role")
    Console.WriteLine("of the VB 2005 Module type.")
    Console.WriteLine("*****")
    ' Reset to previous color of your console text.
    Console.ForegroundColor = ConsoleColor.Green
    Console.WriteLine()
End Sub

Sub GreetUser()
    Dim userName As String
    Console.Write("Please enter your name: ")
    userName = Console.ReadLine()
    Console.WriteLine("Hello there {0}. Nice to meet ya.", userName)
End Sub
End Module

```

Figure 3-1 shows one possible output.

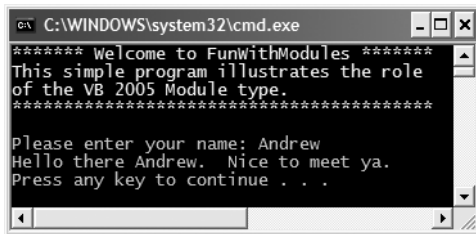


Figure 3-1. *Modules at work*

Projects with Multiple Modules

In our current example, notice that the `Main()` method is able to directly call the `DisplayBanner()` and `GreetUser()` methods. Because these methods are defined within the same module as `Main()`, we are not required to prefix the name of our module (`Module1`) to the member name. However, if you wish to do so, you could retrofit `Main()` as follows:

```

Sub Main()
    ' Show banner.
    Module1.DisplayBanner()
    ' Get user name and say howdy.
    Module1.GreetUser()
End Sub

```

In this case, this is a completely optional bit of syntax (there is no difference in terms of performance or the size of the compiled assembly). However, assume you were to define a new module (`MyModule`) in your project (within the same `*.vb` file, for example), which defines an identically formed `GreetUser()` method:

```
Module MyModule
    Public Sub GreetUser()
        Console.WriteLine("Hello user...")
    End Sub
End Module
```

If you wish to call `MyModule.GreetUser()` from within the `Main()` method, you would now need to *explicitly* prefix the module name. If you do not specify the name of the module, the `Main()` method automatically calls the `Module1.GreetUser()` method, as it is in the same scope as `Main()`:

```
Sub Main()
    ' Show banner.
    DisplayBanner()

    ' Call the GreetUser() method in MyModule.
    MyModule.GreetUser()
End Sub
```

Again, do understand that when a single project defines multiple modules, you are not required to prefix the module name unless the methods are ambiguous. Thus, if your current project were to define yet another module named `MyMathModule`:

```
Module MyMathModule
    Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
        Return x + y
    End Function
    Function Subtract(ByVal x As Integer, ByVal y As Integer) As Integer
        Return x - y
    End Function
End Module
```

you could directly invoke the `Add()` and `Subtract()` functions anywhere within your application (or optionally prefix the module's name):

```
Sub Main()
    ...
    ' Add some numbers.
    Console.WriteLine("10 + 10 is {0}.", Add(10, 10))

    ' Subtract some numbers
    ' (module prefix optional)
    Console.WriteLine("10 - 10 is {0}.", MyMathModule.Subtract(10, 10))
End Sub
```

Note If you are new to the syntax of BASIC languages, rest assured that Chapter 4 will cover the details of building functions and subroutines using VB 2005.

Modules Are Not Creatable

Another trait of the `Module` type is that it cannot be directly created using the VB 2005 `New` keyword (any attempt to do so will result in a compiler error). Therefore the following code is illegal:

```
' Nope! Error, can't allocated modules!
Dim m as New Module1()
```

Rather, a `Module` type simply exposes shared members.

Note If you already have a background in object-oriented programming, be aware that `Module` types cannot be used to build class hierarchies as they are implicitly *sealed*. As well, unlike “normal” classes, modules cannot implement interfaces.

Renaming Your Initial Module

By default, Visual Studio 2005 names the initial `Module` type with the rather nondescript `Module1`. If you were to change the name of the module defining your `Main()` method to a more fitting name (`Program`, for example), the compiler will generate an error such as the following:

'Sub Main' was not found in 'FunWithModules.Module1'.

In order to inform Visual Studio 2005 of the new module name, you are required to reset the “startup object” using the Application tab of the My Project dialog box, as you see in Figure 3-2.

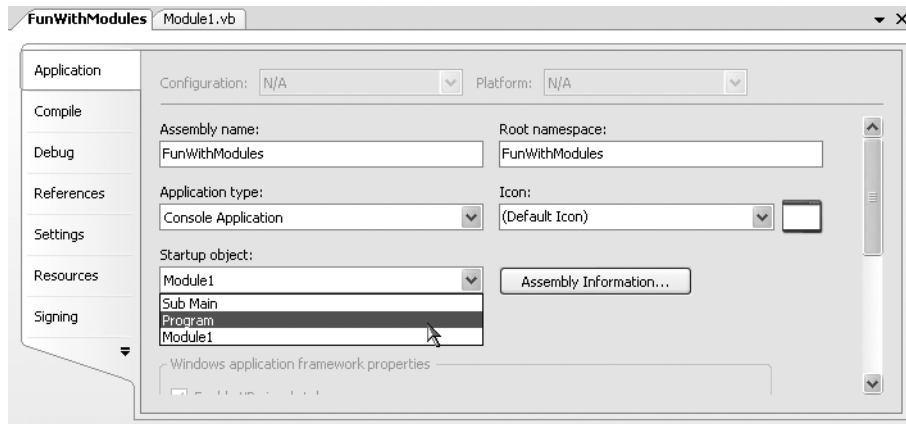


Figure 3-2. Resetting the module name

Once you do so, you will be able to compile your application without error.

Note As a shortcut, if you double-click this specific compiler error within the VS 2005 Error List window, you will be presented with a dialog box that allows you to select the new name of your project's entry point.

Members of Modules

To wrap up our investigation of `Module` types, do know that modules can have additional members beyond subroutines and functions. If you wish to define field data (as well as other members, such as properties or events), you are free to do so. For example, assume you wish to update `MyModule` to contain a single piece of public string data. Note that the `GreetUser()` method will now print out this value when invoked:

```
Module MyModule
    Public userName As String
```

```

Sub GreetUser()
    Console.WriteLine("Hello, {0}.", userName)
End Sub
End Module

```

Like any Module member, the `userName` field can be directly accessed by any part of your application. For example:

```

Sub Main()
...
' Set userName and call second form of GreetUser().
  userName = "Fred"
  MyModule.GreetUser()
...
End Sub

```

Source Code The FunWithModules project is located under the Chapter 3 subdirectory.

The Role of the Main Method

Every VB 2005 executable application (such as a console program, Windows service, or Windows Forms application) must contain a type defining a `Main()` method, which represents the entry point of the application. As you have just seen, the `Main()` method is typically placed within a Module type, which as you recall implicitly defines `Main()` as a shared method.

Strictly speaking, however, `Main()` can also be defined within the scope of a Class type or Structure type as well. If you do define your `Main()` method within either of these types, you must explicitly make use of the `Shared` keyword. To illustrate, create a new console application named FunWithMain. Delete the code within the initial `*.vb` file and replace it with the following:

```

Class Program
' Unlike Modules, members in a Class are not
' automatically shared.
Shared Sub Main()
End Sub
End Class

```

If you attempt to compile your program, you will again receive a compiler error informing you that the `Main()` method cannot be located. Using the Application tab of the My Project dialog box, you can now specify `Sub Main()` as the entry point to the program (as previously shown in Figure 3-2).

Processing Command-line Arguments Using System.Environment

One common task `Main()` will undertake is to process any incoming command-line arguments. For example, consider the VB 2005 command-line compiler, `vbc.exe` (see Chapter 2). As you recall, we specified various options (such as `/target`, `/out`, and so forth) when compiling our code files. The `vbc.exe` compiler processed these input flags in order to compile the output assembly. When you wish to build a `Main()` method that can process incoming command-line arguments for your custom applications, you have a few possible ways to do so.

Your first approach is to make use of the shared `GetCommandLineArgs()` method defined by the `System.Environment` type. This method returns you an array of `String` data types. The first item in the array represents the path to the executable program, while any remaining items in the array represent the command-line arguments themselves. To illustrate, update your current `Main()` method as follows:

```

Class Program
  Shared Sub Main()
    Console.WriteLine("***** Fun with Main() *****")
    ' Get command-line args.
    Dim args As String() = Environment.GetCommandLineArgs()
    Dim s As String
    For Each s In args
      Console.WriteLine("Arg: {0}", s)
    Next
  End Sub
End Class

```

If you were to now run your application at the command prompt, you can feed in your arguments in an identical manner as you did when working with `vbc.exe` (see Figure 3-3).

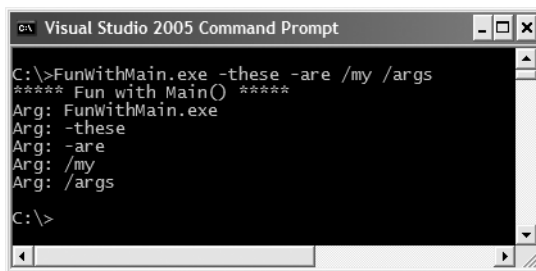


Figure 3-3. Processing command-line arguments

Of course, it is up to you to determine which command-line arguments your program will respond to and how they must be formatted (such as with a `-` or `/` prefix). Here we simply passed in a series of options that were printed to the command prompt. Assume however you were creating a new video game using Visual Basic 2005 and programmed your application to process an option named `-godmode`. If the user starts your application with the flag, you know the user is in fact a *cheater*, and can take an appropriate course of action.

Processing Command-line Arguments with Main()

If you would rather not make use of the `System.Environment` type to process command-line arguments, you can define your `Main()` method to take an incoming array of strings. To illustrate, update your code base as follows:

```

Shared Sub Main(ByVal args As String())
  Console.WriteLine("***** Fun with Main() *****")
  ' Get command-line args.
  Dim s As String
  For Each s In args
    Console.WriteLine("Arg: {0}", s)
  Next
End Sub

```

When you take this approach, the first item in the incoming array is indeed the first command-line argument (rather than the path to the executable). If you were to run your application once again, you will find each command-line option is printed to the console.

Main() As a Function (not a Subroutine)

It is also possible to define `Main()` as a function returning an `Integer`, rather than a subroutine (which never has a return value). This approach to building a `Main()` method has its roots in C-based languages, where returning the value 0 indicates the program has terminated without error. You will seldom (if ever) need to build your `Main()` method in this manner; however, for the sake of completion, here is one example:

```
Shared Function Main(ByVal args As String()) As Integer
    Console.WriteLine("***** Fun with Main() *****")
    Dim s As String
    For Each s In args
        Console.WriteLine("Arg: {0}", s)
    Next
    ' Return a value to the OS.
    Return 0
End Function
```

Regardless of how you define your `Main()` method, the purpose remains the same: interact with the types that carry out the functionality of your application. Once the final statement within the `Main()` method has executed, `Main()` exits and your application terminates.

Simulating Command-line Arguments Using Visual Studio 2005

Finally, let me point out that Visual Studio 2005 does allow you to simulate incoming command-line arguments. Rather than having to run your application at a command line to feed in arguments, you can explicitly specify arguments using the Debug tab of the My Project dialog box, shown in Figure 3-4 (note the Command line arguments text area).

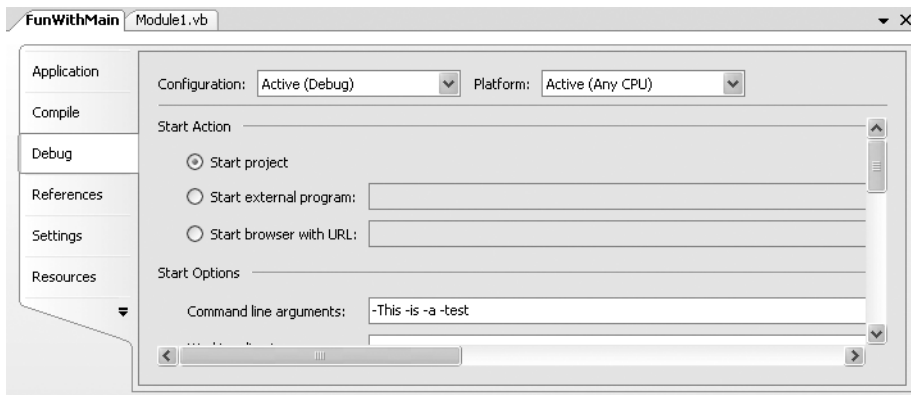


Figure 3-4. *Simulating command-line arguments*

When you compile and run your application under Debug mode, the specified arguments are passed to your `Main()` method automatically. Do know that when you compile and run a Release build of your application (which can be established using the Compile tab of the My Project dialog box), this is no longer the case.

An Interesting Aside: Some Additional Members of the System.Environment Class

The Environment type exposes a number of extremely helpful methods beyond GetCommandLineArgs(). This class allows you to obtain a number of details regarding the operating system currently hosting your .NET application using various shared members. To illustrate the usefulness of System.Environment, update your Main() method with the following logic:

```
Shared Function Main(ByVal args As String()) As Integer
...
' OS running this app?
Console.WriteLine("Current OS: {0}", Environment.OSVersion)

' List the drives on this machine.
Dim drives As String() = Environment.GetLogicalDrives()
Dim d As String
For Each d In drives
    Console.WriteLine("You have a drive named {0}.", d)
Next

' Which version of the .NET platform is running this app?
Console.WriteLine("Executing version of .NET: {0}", _
    Environment.Version)
Return 0
End Function
```

Figure 3-5 shows a possible test run.

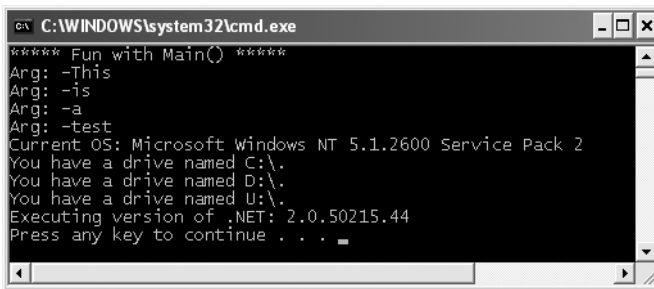


Figure 3-5. *Displaying system environment variables*

The Environment type defines members other than those seen in the previous example. Table 3-1 documents some additional properties of interest; however, be sure to check out the .NET Framework 2.0 SDK documentation for full details.

Table 3-1. *Select Properties of System.Environment*

Property	Meaning in Life
CurrentDirectory	Gets the full path to the current application
MachineName	Gets the name of the current machine
NewLine	Gets the newline symbol for the current environment
ProcessorCount	Returns the number of processors on the current machine
SystemDirectory	Returns the full path to the system directory
UserName	Returns the name of the user that started this application

Source Code The FunWithMain project is located under the Chapter 3 subdirectory.

The System.Console Class

Almost all of the example applications created over the course of the initial chapters of this text make extensive use of the `System.Console` class. While it is true that a console user interface (CUI) is not as enticing as a graphical user interface (GUI) or web-based front end, restricting the early examples to console applications will allow us to keep focused on the syntax of Visual Basic 2005 and the core aspects of the .NET platform, rather than dealing with the complexities of building GUIs.

As its name implies, the `Console` class encapsulates input, output, and error stream manipulations for console-based applications. While `System.Console` has been a part of the .NET Framework since its inception, with the release of .NET 2.0, the `Console` type has been enhanced with additional functionality. Table 3-2 lists some (but definitely not all) of the new members of interest.

Table 3-2. *Select .NET 2.0-Specific Members of System.Console*

Member	Meaning in Life
Beep()	Forces the console to emit a beep of a specified frequency and duration.
BackgroundColor ForegroundColor	These properties set the background/foreground colors for the current output. They may be assigned any member of the <code>ConsoleColor</code> enumeration.
BufferHeight BufferWidth	These properties control the height/width of the console's buffer area.
Title	This property sets the title of the current console.
WindowHeight WindowWidth WindowTop WindowLeft	These properties control the dimensions of the console in relation to the established buffer.
Clear()	This method clears the established buffer and console display area.

Basic Input and Output with the Console Class

In addition to the members in Table 3-2, the `Console` type defines a set of methods to capture input and output, all of which are shared and are therefore called by prefixing the name of the class (`Console`) to the method name. As you have seen, `WriteLine()` pumps a text string (including a carriage return) to the output stream. The `Write()` method pumps text to the output stream without a carriage return. `ReadLine()` allows you to receive information from the input stream up until the carriage return, while `Read()` is used to capture a single character from the input stream.

To illustrate basic I/O using the Console class, create a new console application named BasicConsoleIO and update your Main() method with logic that prompts the user for some bits of information and echoes each item to the standard output stream.

```
Sub Main()
    Console.WriteLine("***** Fun with Console IO *****")
    ' Echo some information to the console.
    Console.Write("Enter your name: ")
    Dim s As String = Console.ReadLine()
    Console.WriteLine("Hello, {0}", s)
    Console.Write("Enter your age: ")
    s = Console.ReadLine()
    Console.WriteLine("You are {0} years old", s)
End Sub
```

Formatting Console Output

During these first few chapters, you have certainly noticed numerous occurrences of the tokens {0}, {1}, and the like embedded within a string literal. The .NET platform introduces a new style of string formatting, which can be used by any .NET programming language (including VB 2005). Simply put, when you are defining a string literal that contains segments of data whose value is not known until runtime, you are able to specify a placeholder within the literal using this curly-bracket syntax. At runtime, the value(s) passed into Console.WriteLine() are substituted for each placeholder. To illustrate, update your current Main() method as follows:

```
Sub Main()
...
    ' Specify string placeholders and values to use at
    ' runtime.
    Dim theInt As Integer = 90
    Dim theDouble As Double = 9.99
    Dim theBool As Boolean = True
    Console.WriteLine("Value of theInt: {0}", theInt)
    Console.WriteLine("theDouble is {0} and theBool is {1}.", _
        theDouble, theBool)
End Sub
```

The first parameter to WriteLine() represents a string literal that contains optional placeholders designated by {0}, {1}, {2}, and so forth. Be very aware that the first ordinal number of a curly-bracket placeholder always begins with 0. The remaining parameters to WriteLine() are simply the values to be inserted into the respective placeholders (in this case, an Integer, a Double, and a Boolean).

Note If you have a mismatch between the number of uniquely numbered curly-bracket placeholders and fill arguments, you will receive a FormatException exception at runtime.

It is also permissible for a given placeholder to repeat within a given string. For example, if you are a Beatles fan and want to build the string "9, Number 9, Number 9" you would write

```
' John says...
Console.WriteLine("{0}, Number {0}, Number {0}", 9)
```

Also know, that it is possible to position each placeholder in any location within a string literal, and need not follow an increasing sequence. For example, consider the following code snippet:

```
' Prints: 20, 10, 30
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30)
```

.NET String Formatting Flags

If you require more elaborate formatting, each placeholder can optionally contain various format characters. Each format character can be typed in either uppercase or lowercase with little or no consequence. Table 3-3 shows your core formatting options.

Table 3-3. .NET String Format Characters

String Format Character	Meaning in Life
C or c	Used to format currency. By default, the flag will prefix the local cultural symbol (a dollar sign [\$] for U.S. English).
D or d	Used to format decimal numbers. This flag may also specify the minimum number of digits used to pad the value.
E or e	Used for exponential notation.
F or f	Used for fixed-point formatting.
G or g	Stands for <i>general</i> . This character can be used to format a number to fixed or exponential format.
N or n	Used for basic numerical formatting (with commas).
X or x	Used for hexadecimal formatting. If you use an uppercase X, your hex format will also contain uppercase characters.

These format characters are suffixed to a given placeholder value using the colon token (e.g., {0:C}, {1:d}, {2:X}, and so on). Now, update the Main() method with the following logic:

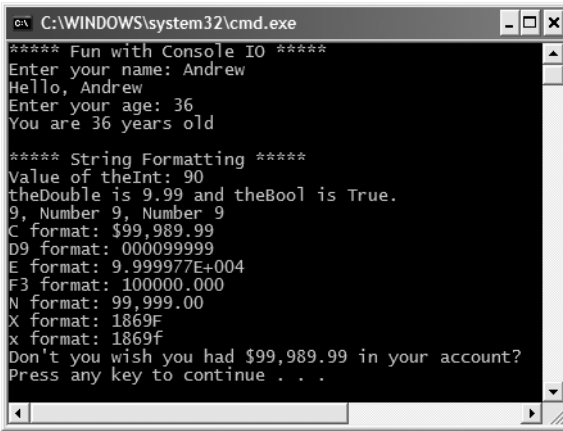
```
' Now make use of some format tags.
Sub Main()
...
    Console.WriteLine("C format: {0:C}", 99989.987)
    Console.WriteLine("D9 format: {0:D9}", 99999)
    Console.WriteLine("E format: {0:E}", 99999.76543)
    Console.WriteLine("F3 format: {0:F3}", 99999.9999)
    Console.WriteLine("N format: {0:N}", 99999)
    Console.WriteLine("X format: {0:X}", 99999)
    Console.WriteLine("x format: {0:x}", 99999)
End Sub
```

Here we are defining numerous string literals, each of which has a segment not known until runtime. At runtime, the format character will be used internally by the Console type to print out the entire string in the desired format.

Be aware that the use of the .NET string formatting characters are not limited to console applications! These same flags can be used when calling the shared String.Format() method. This can be helpful when you need to build a string containing numerical values in memory for use in any application type (Windows Forms, ASP.NET, XML web services, and so on). To illustrate, update Main() with the following final code:

```
' Now make use of some format tags.
Sub Main()
...
    ' Use the shared String.Format() method to build a new string.
    Dim formatStr As String
    formatStr = _
        String.Format("Don't you wish you had {0:C} in your account?", 99989.987)
    Console.WriteLine(formatStr)
End Sub
```

Figure 3-6 shows a test run of our application.



```
***** Fun with Console IO *****
Enter your name: Andrew
Hello, Andrew
Enter your age: 36
You are 36 years old

***** String Formatting *****
Value of theInt: 90
theDouble is 9.99 and theBool is True.
9, Number 9, Number 9
C format: $99,989.99
D9 format: 000099999
E format: 9.999977E+004
F3 format: 100000.000
N format: 99,999.00
X format: 1869F
x format: 1869f
Don't you wish you had $99,989.99 in your account?
Press any key to continue . . .
```

Figure 3-6. *The System.Console type in action*

Source Code The BasicConsoleIO project is located under the Chapter 3 subdirectory.

The System Data Types and VB 2005 Shorthand Notation

Like any programming language, VB 2005 defines an intrinsic set of data types, which are used to represent local variables, member variables, and member parameters. Although many of the VB 2005 data types are named identically to data types found under VB 6.0, be aware that there is *not* a direct mapping (especially in terms of a data type's maximum and minimum range). Furthermore, VB 2005 defines a set of brand new data types not supported by previous versions of the language (UInteger, ULong, SByte) that account for signed and unsigned data.

Note The UInteger, ULong, and SByte data types are *not* CLS compliant (see Chapters 1 and 14 for details on CLS compliance). Therefore, if you expose these data types from an assembly, you cannot guarantee that every .NET programming language will be able to process this data.

The most significant change from VB 6.0 is that the data type keywords of Visual Basic 2005 are actually shorthand notations for full-blown types in the System namespace. Table 3-4 documents the data types of VB 2005 (with the size of storage allocation), the System data type equivalents, and the range of each type.

Table 3-4. *The Intrinsic Data Types of VB 2005*

VB 2005 Data Type	System Data Type	Range
Boolean (platform dependent)	System.Boolean	True or False.
Byte (1 byte)	System.Byte	0 to 255 (unsigned).
Char (2 bytes)	System.Char	0 to 65535 (unsigned).
Date (8 bytes)	System.DateTime	January 1, 0001 to December 31, 9999.
Decimal (16 bytes)	System.Decimal	+/-79,228,162,514,264,337,593,543,950,335 with no decimal point. +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/-0.00000000000000000000000001.
Double (8 bytes)	System.Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values. 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
Integer (4 bytes)	System.Int32	-2,147,483,648 to 2,147,483,647.
Long (8 bytes)	System.Int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
Object (4 bytes)	System.Object	Any type can be stored in a variable of type Object.
SByte (1 byte)	System.SByte	-128 through 127 (signed).
Short (2 bytes)	System.Int16	-32,768 to 32,767.
Single (4 bytes)	System.Single	This single-precision floating-point value can take the range of -3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
String (platform dependent)	System.String	A string of Unicode characters between 0 to approximately 2 billion characters.
UInteger (4 bytes)	System.UInt32	0 through 4,294,967,295 (unsigned).
ULong (8 bytes)	System.UInt64	0 through 18,446,744,073,709,551,615 (unsigned).
SByte (2 bytes)	System.UInt16	0 through 65,535 (unsigned).

Each of the numerical types (Short, Integer, and so forth) as well as the Date type map to a corresponding *structure* in the System namespace. In a nutshell, structures are “value types” allocated on the stack rather than on the garbage-collected heap. On the other hand, String and Object are “reference types,” meaning the variable is allocated on the managed heap. You examine full details of value and reference types in Chapter 11.

Variable Declaration and Initialization

When you are declaring a data type as a local variable (e.g., within a member scope), you do so using the `Dim` and `As` keywords. By way of a few examples:

```
Sub MyMethod()  
    ' Dim variableName As dataType  
    Dim age As Integer  
    Dim firstName As String  
    Dim isUserOnline As Boolean  
End Sub
```

One helpful syntactic change that has occurred with the release of the .NET platform is the ability to declare a sequence of variables on a single line of code. Of course, VB 6.0 also supported this ability, but the semantics were a bit nonintuitive and a source of subtle bugs. For example, under VB 6.0, if you do not explicitly set the data types of each variable, the unqualified variables were set to the VB 6.0 Variant data type:

```
' In this line of VB 6.0 code, varOne  
' is implicitly defined to be of type Variant!  
Dim varOne, varTwo As Integer
```

This behavior is a bit cumbersome, given that the only way you are able to define multiple variables of the same type under VB 6.0 is to write the following slightly redundant code:

```
Dim varOne As Integer, varTwo As Integer
```

or worse yet, on multiple lines of code:

```
Dim varOne As Integer  
Dim varTwo As Integer
```

Although these approaches are still valid using VB 2005, when you declare multiple variables on a single line, they *all* are defined in terms of the specified data type. Thus, in the following VB 2005 code, you have created two variables of type Integer.

```
Sub MyMethod()  
    ' In this line of VB 2005 code, varOne  
    ' and varTwo are both of type Integer!  
    Dim varOne, varTwo As Integer  
End Sub
```

On a final note, VB 2005 now supports the ability to assign a value to a type directly at the point of declaration. To understand the significance of this new bit of syntax, consider the fact that under VB 6.0, you were forced to write the following:

```
' VB 6.0 code.  
Dim i As Integer  
i = 99
```

While this is in no way a major showstopper, VB 2005 allows you to streamline variable assignment using the following notation:


```

Sub MyMethod()
    ' Dim variableName As dataType = initialValue
    Dim age As Integer = 36
    Dim firstName As String = "Sid"
    Dim isUserOnline As Boolean = True
End Sub

```

Default Values of Data Types

All VB 2005 data types have a default value that will automatically be assigned to the variable. The default values are very predictable, and can be summarized as follows:

- Boolean types are set to False.
- Numeric data is set to 0 (or 0.0 in the case of floating-point data types).
- String types are set to empty strings.
- Char types are set to a single empty character.
- Date types are set to 1/1/0001 12:00:00 AM.
- Initialized object references are set to Nothing.

Given these rules, ponder the following code:

```

' Fields of a class or Module receive automatic default assignments.
Module Program
    Public myInt As Integer      ' Set to 0.
    Public myString As String   ' Set to empty String.
    Public myBool As Boolean     ' Set to False.
    Public myObj As Object      ' Set to Nothing.
End Module

```

In Visual Basic 2005, the same rules of default values hold true for local variables defined within a given scope. Given this, the following method would return the value 0, given that each local Integer has been automatically assigned the value 0:

```

Function Add() As Integer
    Dim a, b As Integer
    Return a + b      ' Returns zero.
End Function

```

The Data Type Class Hierarchy

It is very interesting to note that even the primitive .NET data types are arranged in a “class hierarchy.” If you are new to the world of inheritance, you will discover the full details in Chapter 6. Until then, just understand that types at the top of a class hierarchy provide some default behaviors that are granted to the derived types. The relationship between these core system types can be understood as shown in Figure 3-7.

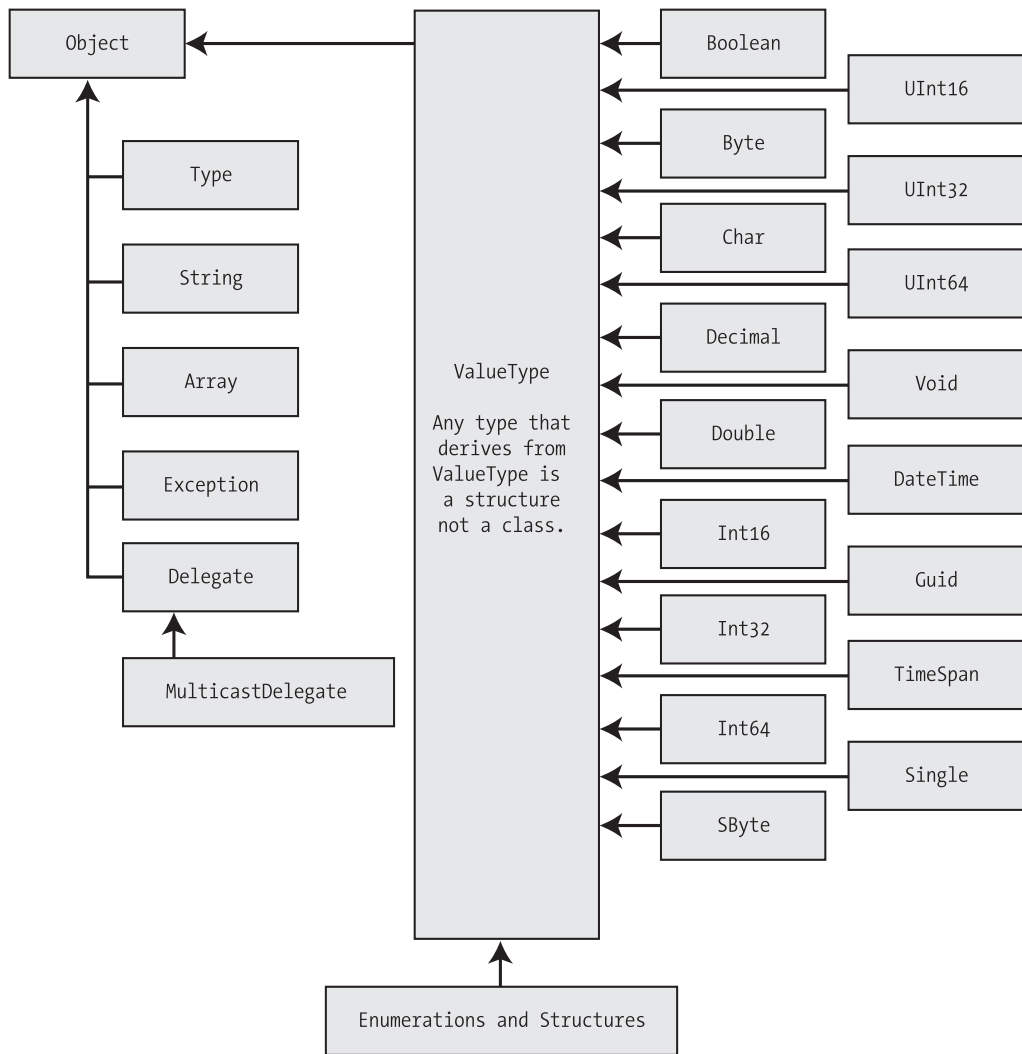


Figure 3-7. *The class hierarchy of System types*

Notice that each of these types ultimately derives from `System.Object`, which defines a set of methods (`ToString()`, `Equals()`, `GetHashCode()`, and so forth) common to all types in the .NET base class libraries (these methods are fully detailed in Chapter 6).

Also note that many numerical data types derive from a class named `System.ValueType`. Descendants of `ValueType` are automatically allocated on the stack and therefore have a very predictable lifetime and are quite efficient. On the other hand, types that do not have `System.ValueType` in their inheritance chain (such as `System.Type`, `System.String`, `System.Array`, `System.Exception`, and `System.Delegate`) are not allocated on the stack, but on the garbage-collected heap.

Without getting too hung up on the details of `System.Object` and `System.ValueType` for the time being (again, more details in Chapter 11), simply know that because a VB 2005 keyword (such as `Integer`) is simply shorthand notation for the corresponding system type (in this case, `System.Int32`), the following is perfectly legal syntax, given that `System.Int32` (the VB 2005 `Integer`) eventually derives from `System.Object`, and therefore can invoke any of its public members:

```

Sub Main()
    ' A VB 2005 Integer is really a shorthand for System.Int32.
    ' which inherits the following members from System.Object.
    Console.WriteLine(12.GetHashCode()) ' Prints the type's hash code value.
    Console.WriteLine(12.Equals(23))    ' Prints False
    Console.WriteLine(12.ToString())     ' Returns the value "12"
    Console.WriteLine(12.GetType())      ' Prints System.Int32
End Sub

```

Note By default, Visual Studio 2005 does not show these “advanced” methods from IntelliSense. To disable this behavior (which I recommend you do), activate the Tools ► Options menu, select Basic from the Text Editor node, and uncheck Hide Advanced members.

“New-ing” Intrinsic Data Types

All intrinsic data types support what is known as a *default constructor* (see Chapter 5). In a nutshell, this feature allows you to create a variable using the `New` keyword, which automatically sets the variable to its default value. Although it is more cumbersome to use the `New` keyword when creating a basic data type variable, the following is syntactically well-formed VB 2005 code:

```

' When you create a basic data type with New,
' it is automatically set to its default value.
Dim b1 As New Boolean() ' b1 automatically set to False.

```

On a related note, you could also declare an intrinsic data type variable using the full type name through either of these approaches:

```

' These statements are also functionally identical.
Dim b2 As System.Boolean = New System.Boolean()
Dim b3 As System.Boolean

```

Of course, the chances that you will define a simple Boolean using the full type name or the `New` keyword in your code is slim to none. It is important, however, to always remember that the VB 2005 keywords for simple data types are little more than a shorthand notation for real types in the `System` namespace.

Experimenting with Numerical Data Types

To experiment with the intrinsic VB 2005 data types, create a new console application named `BasicDataTypes`. First up, understand that the numerical types of .NET support `MaxValue` and `MinValue` properties that provide information regarding the range a given type can store. For example:

```

Sub Main()
    Console.WriteLine("***** Fun with Data Types *****")
    Console.WriteLine("Max of Integer: {0}", Integer.MaxValue)
    Console.WriteLine("Min of Integer: {0}", Integer.MinValue)
    Console.WriteLine("Max of Double: {0}", Double.MaxValue)
    Console.WriteLine("Min of Double: {0}", Double.MinValue)
End Sub

```

In addition to the `MinValue/MaxValue` properties, a given numerical system type may define further useful members. For example, the `System.Double` type allows you to obtain the values for `epsilon` and `infinity` (which may be of interest to those of you with a mathematical flare):

```

Console.WriteLine("Double.Epsilon: {0}", Double.Epsilon)
Console.WriteLine("Double.PositiveInfinity: {0}", _

```

```
Double.PositiveInfinity)
Console.WriteLine("Double.NegativeInfinity: {0}", _
Double.NegativeInfinity)
```

Members of System.Boolean

Next, consider the `System.Boolean` data type. The only valid assignment a VB 2005 Boolean can take is from the set `{True | False}`. Given this point, it should be clear that `System.Boolean` does not support a `MinValue/MaxValue` property set, but rather `TrueString/FalseString` (which yields the string "True" or "False" respectively):

```
Console.WriteLine("Boolean.FalseString: {0}", Boolean.FalseString)
Console.WriteLine("Boolean.TrueString: {0}", Boolean.TrueString)
```

Members of System.Char

VB 2005 textual data is represented by the intrinsic `String` and `Char` keywords, which are simple shorthand notations for `System.String` and `System.Char`, both of which are Unicode under the hood. As you most certainly already know, a string is a contiguous set of characters (e.g., "Hello"). As of the .NET platform, VB now has a data type (`Char`) that can represent a single slot in a `String` type (e.g., 'H').

By default, when you define textual data within double quotes, the VB 2005 compiler assumes you are defining a full-blown `String` type. However, to build a single character string literal that should be typed as a `Char`, place the character between double quotes and tack on a single *c* after the closing quote. Doing so ensures that the double-quoted text literal is indeed represented as a `System.Char`, rather than a `System.String`:

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

Note When you enable `Option Strict` (described in just a moment) for your project, the VB 2005 compiler demands that you tack on the *c* suffix to a `Char` data type when assigning a value.

The `System.Char` type provides you with a great deal of functionality beyond the ability to hold a single point of character data. Using the shared methods of `System.Char`, you are able to determine whether a given character is numerical, alphabetical, a point of punctuation, or whatnot. To illustrate, update `Main()` with the following statements:

```
' Fun with System.Char.
Dim myChar As Char = "a"c
Console.WriteLine("Char.IsDigit('a'): {0}", Char.IsDigit(myChar))
Console.WriteLine("Char.IsLetter('a'): {0}", Char.IsLetter(myChar))
Console.WriteLine("Char.IsWhiteSpace('Hello There', 5): {0}", _
Char.IsWhiteSpace("Hello There", 5))
Console.WriteLine("Char.IsWhiteSpace('Hello There', 6): {0}", _
Char.IsWhiteSpace("Hello There", 6))
Console.WriteLine("Char.IsPunctuation('?'): {0}", _
Char.IsPunctuation("?c"))
```

As illustrated in the previous code snippet, the members of `System.Char` have two calling conventions: a single character or a string with a numerical index that specifies the position of the character to test.

Parsing Values from String Data

The .NET data types provide the ability to generate a variable of their underlying type given a textual equivalent (e.g., parsing). This technique can be extremely helpful when you wish to convert a bit of user input data (such as a selection from a GUI-based drop-down list box) into a numerical value. Consider the following parsing logic:

```
' Fun with parsing
Dim b As Boolean = Boolean.Parse("True")
Console.WriteLine("Value of myBool: {0}", b)
Dim d As Double = Double.Parse("99.884")
Console.WriteLine("Value of myDbl: {0}", d)
Dim i As Integer = Integer.Parse("8")
Console.WriteLine("Value of myInt: {0}", i)
Dim c As Char = Char.Parse("w")
Console.WriteLine("Value of myChar: {0}", c)
```

Source Code The BasicDataTypes project is located under the Chapter 3 subdirectory.

Understanding the System.String Type

As mentioned, `String` is a native data type in VB 2005. Like all intrinsic types, the VB 2005 `String` keyword actually is a shorthand notation for a true type in the .NET base class library, which in this case is `System.String`. Therefore, you are able to declare a `String` variable using either of these notations (in addition to using the `New` keyword as shown previously):

```
' These two string declarations are functionally equivalent.
Dim firstName As String
Dim lastName As System.String
```

`System.String` provides a number of methods you would expect from such a utility class, including methods that return the length of the character data, find substrings within the current string, convert to and from uppercase/lowercase, and so forth. Table 3-5 lists some (but by no means all) of the interesting members.

Table 3-5. *Select Members of System.String*

Member of String Class	Meaning in Life
<code>Chars</code>	This property returns a specific character within the current string.
<code>Length</code>	This property returns the length of the current string.
<code>Compare()</code>	Compares two strings.
<code>Contains()</code>	Determines whether a string contain a specific substring.
<code>Equals()</code>	Tests whether two string objects contain identical character data.
<code>Format()</code>	Formats a string using other primitives (i.e., numerical data, other strings) and the <code>{0}</code> notation examined earlier in this chapter.
<code>Insert()</code>	Inserts a string within a given string.
<code>PadLeft()</code> <code>PadRight()</code>	These methods are used to pad a string with some character.
<code>Remove()</code> <code>Replace()</code>	Use these methods to receive a copy of a string, with modifications (characters removed or replaced).

Continued

Table 3-5. *Continued*

Member of String Class	Meaning in Life
Split()	Returns a String array containing the substrings in this instance that are delimited by elements of a specified Char or String array.
Trim()	Removes all occurrences of a set of specified characters from the beginning and end of the current string.
ToUpper() ToLower()	Creates a copy of the current string in uppercase or lowercase format.

Basic String Manipulation

Working with the members of `System.String` is as you would expect. Simply create a `String` data type and make use of the provided functionality via the dot operator. Do be aware that a few of the members of `System.String` are shared members, and are therefore called at the class (rather than the object) level. Assume you have created a new console application named `FunWithStrings`, and updated `Main()` as follows:

```
Module Program
    Sub Main()
        Console.WriteLine("***** Fun with Strings *****")
        Dim firstName As String = "June"
        Console.WriteLine("Value of firstName: {0}", firstName)
        Console.WriteLine("firstName has {0} characters.", firstName.Length)
        Console.WriteLine("firstName in uppercase: {0}", firstName.ToUpper())
        Console.WriteLine("firstName in lowercase: {0}", firstName.ToLower())

        Dim myValue As Integer = 3456787
        Console.WriteLine("Hex vaule of myValue is: {0}", _
            String.Format("{0:X}", myValue))
        Console.WriteLine("Currency vaule of myValue is: {0}", _
            String.Format("{0:C}", myValue))
    End Sub
End Module
```

Notice how the shared `Format()` method supports the same formatting tokens as the `Console.WriteLine()` method examined earlier in the chapter. Also notice that unlike `String.Format()`, the `ToUpper()` and `ToLower()` methods have not implemented as shared members and are therefore called directly from the `String` object.

String Concatenation (and the “Newline” Constant)

String variables can be connected together to build a larger `String` via the VB 2005 ampersand operator (`&`). As you may know, this technique is formally termed *string concatenation*:

```
Module Program
    Sub Main()
        Console.WriteLine("***** Fun with Strings *****")
        ...
        Dim s1 As String = "Programming the "
        Dim s2 As String = "PsychoDrill (PTP)"
        Dim s3 As String = s1 & s2
        Console.WriteLine(s3)
    End Sub
End Module
```

Note VB 2005 also allows you to concatenate String objects using the plus sign (+). However, given that the + symbol can be applied to numerous data types, there is a possibility that your String object cannot be “added” to one of the operands. The ampersand, on the other hand, can only apply to Strings, and therefore is the recommended approach.

You may be interested to know that the VB 2005 & symbol is processed by the compiler to emit a call to the shared String.Concat() method. In fact, if you were to compile the previous code and open the assembly within ildasm.exe (see Chapter 1), you would find the CIL code shown in Figure 3-8.

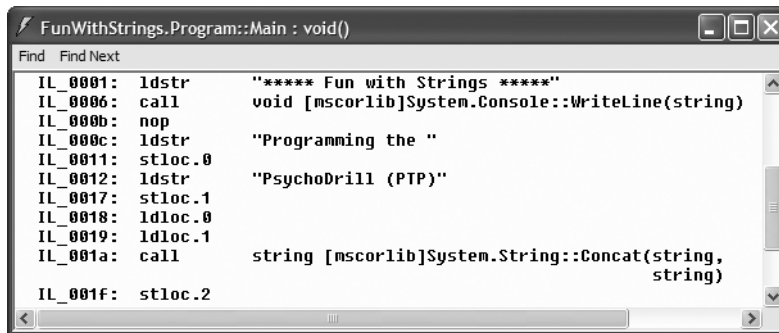


Figure 3-8. The VB 2005 & operator results in a call to String.Concat().

Given this, it is possible to perform string concatenation by calling String.Concat() directly (although you really have not gained anything by doing so, in fact you have incurred additional keystrokes!):

```

Module Program
    Sub Main()
        Console.WriteLine("***** Fun with Strings *****")
    ...
        Dim s1 As String = "Programming the "
        Dim s2 As String = "PsychoDrill (PTP)"
        Dim s3 As String = String.Concat(s1, s2)
        Console.WriteLine(s3)
    End Sub
End Module
  
```

On a related note, do know that the VB 6.0–style string constants (such as vbCrLf, vbCrLf, and vbCrLf) are still exposed through the Microsoft.VisualBasic.dll assembly (see Chapter 2). Therefore, if you wish to concatenate a string that contains various newline characters (for display purposes), you may do so as follows:

```

Module Program
    Sub Main()
        Console.WriteLine("***** Fun with Strings *****")
    ...
        Dim s1 As String = "Programming the "
        Dim s2 As String = "PsychoDrill (PTP)"
  
```

```

    Dim s3 As String = String.Concat(s1, s2)
    s3 += vbLf & "was a great industrial project."
    Console.WriteLine(s3)
End Sub
End Module

```

Note If you have a background in C-based languages, understand that the `vbLf` constant is functionally equivalent to the newline escape character (`\n`).

Strings and Equality

As fully explained in Chapter 11, a *reference type* is an object allocated on the garbage-collected managed heap. By default, when you perform a test for equality on reference types (via the VB 2005 `=` and `<>` operators), you will be returned `True` if the references are pointing to the same object in memory. However, even though the `String` data type is indeed a reference type, the equality operators have been redefined to compare the *values* of `String` objects, not the memory to which they refer:

```

Module Program
    Sub Main()
        Console.WriteLine("***** Fun with Strings *****")
        ...
        Dim strA As String = "Hello!"
        Dim strB As String = "Yo!"
        ' False!
        Console.WriteLine("strA = strB?: {0}", strA = strB)
        strB = "HELLO!"
        ' False!
        Console.WriteLine("strA = strB?: {0}", strA = strB)
        strB = "Hello!"
        ' True!
        Console.WriteLine("strA = strB?: {0}", strA = strB)
    End Sub
End Module

```

Notice that the VB 2005 equality operators perform a case-sensitive, character-by-character equality test. Therefore, "Hello!" is not equal to "HELLO!", which is different from "hello!".

Strings Are Immutable

One of the interesting aspects of `System.String` is that once you assign a `String` object with its initial value, the character data *cannot be changed*. At first glance, this might seem like a flat-out lie, given that we are always reassigning strings to new values and due to the fact that the `System.String` type defines a number of methods that appear to modify the character data in one way or another (uppercase, lowercase, etc.). However, if you look closer at what is happening behind the scenes, you will notice the methods of the `String` type are in fact returning you a brand new `String` object in a modified format:

```

Module Program
    Sub Main()
        Console.WriteLine("***** Fun with Strings *****")
        ...
        ' Set initial string value
        Dim initialString As String = "This is my string."
        Console.WriteLine("Initial value: {0}", initialString)
    End Sub
End Module

```



```

' Uppercase the initialString?
Dim upperString As String = initialString.ToUpper()
Console.WriteLine("Upper case copy: {0}", upperString)

' Nope! initialString is in the same format!
Console.WriteLine("Initial value: {0}", initialString)
End Sub
End Module

```

If you examine the output in Figure 3-9, you can verify that the original String object (`initialString`) is not uppercased when calling `ToUpper()`, rather you are returned a copy of the string in a modified format.

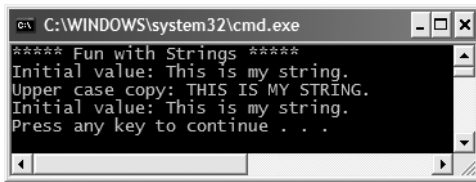


Figure 3-9. *Strings are immutable!*

The same law of immutability holds true when you use the VB 2005 assignment operator. To illustrate, comment out any existing code within `Main()` (to decrease the amount of generated CIL code) and add the following logic:

```

Module Program
Sub Main()
    Dim strObjA As String = "String A reporting."
    strObjA = "This is a new string"
End Sub
End Module

```

Now, compile your application and load the assembly into `ildasm.exe` (again, see Chapter 1). If you were to double-click the `Main()` method, you would find the CIL code shown in Figure 3-10.

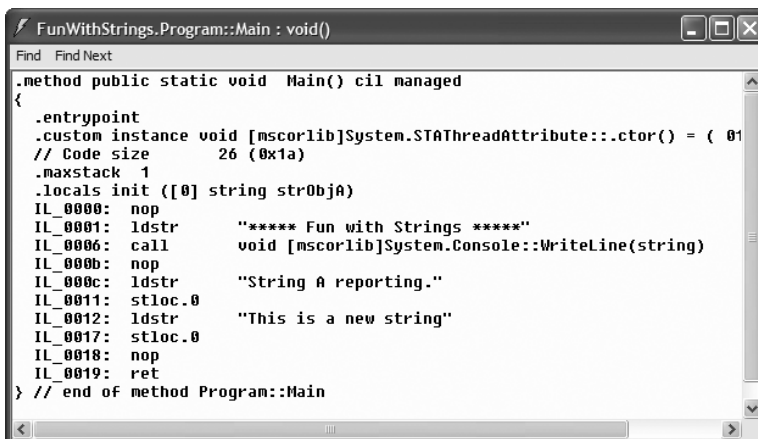


Figure 3-10. *Assigning a value to a String object results in a new String object.*

Although I don't imagine you are too interested in the low-level details of the Common Intermediate Language (CIL), do note that the `Main()` method makes numerous calls to the `ldstr` (load string) opcode. Simply put, the `ldstr` opcode of CIL will always create a new `String` object on the managed heap. The previous `String` object that contained the value "String A reporting." is no longer being used by the program, and will eventually be garbage collected.

So, what exactly are we to gather from this insight? In a nutshell, the `String` type can be inefficient and result in bloated code if misused. If you need to represent basic character data such as a US Social Security number, first or last names, or simple string literals used within your application, the `String` data type is the perfect choice.

However, if you are building an application that makes heavy use of textual data (such as a word processing program), it would be a very bad idea to represent the word processing data using `String` types, as you will most certainly (and often indirectly) end up making unnecessary copies of string data. So what is a programmer to do? Glad you asked.

The System.Text.StringBuilder Type

Given that the `String` type can be quite inefficient when used with reckless abandon, the .NET base class libraries provide the `System.Text` namespace. Within this (relatively small) namespace lives a class named `StringBuilder`. Like the `System.String` class, `StringBuilder` defines methods that allow you to replace or format segments and so forth.

What is unique about the `StringBuilder` is that when you call members of the `StringBuilder`, you are directly modifying the object's internal character data, not obtaining a copy of the data in a modified format (and is thus more efficient). When you create an instance of the `StringBuilder`, you will supply the object's initial startup values via one of many *constructors*. Chapter 5 dives into the details of class constructors; however, if you are new to the topic, simply understand that constructors allow you to create an object with an initial state when you apply the `New` keyword. Consider the following usage of `StringBuilder`:

```
Imports System.Text ' StringBuilder lives here!

Module Program
    Sub Main()
        ...
        ' Use the StringBuilder.
        Dim sb As New StringBuilder("**** Fantastic Games ****")
        sb.Append(vbLf)
        sb.AppendLine("Half Life 2")
        sb.AppendLine("Beyond Good and Evil")
        sb.AppendLine("Deus Ex 1 and 2")
        sb.Append("System Shock")
        sb.Replace("2", "Deus Ex: Invisible War")
        Console.WriteLine(sb)
        Console.WriteLine("sb as {0} chars.", sb.Length)
    End Sub
End Module
```

Here we see constructed a `StringBuilder` set to the initial value "**** Fantastic Games ****". As you can see, we are appending to the internal buffer, and are able to replace (or remove) characters at will. By default, a `StringBuilder` is only able to hold a string of 16 characters or less; however, this initial value can be changed via a constructor argument:

```
' Make a StringBuilder with an initial size of 256.
Dim sb As New StringBuilder("**** Fantastic Games ****", 256)
```

If you append more characters than the specified limit, the `StringBuilder` object will copy its data into a new instance and grow the buffer by the specified limit.

Source Code The FunWithStrings project is located under the Chapter 3 subdirectory.

Final Commentary of VB 2005 Data Types

To wrap up the discussion of intrinsic data types, there are a few points of interest, especially when it comes to changes between VB 6.0 and VB 2005. As you have already seen in Table 3-4, the maximum and minimum bounds of many types have been retrofitted to be consistent with the rules of the .NET-specific Common Type System (CTS). In addition to this fact, also be aware of the following updates:

- VB 2005 does not support a Currency data type. The `Decimal` type supports far greater precision (and functionality) than the VB 6.0 Currency type.
- The Variant data type is no longer supported under the .NET platform. However, if you are using a legacy COM type returning a VB 6.0 Variant, it is still possible to process the data.

At this point, I hope you understand that each data type keyword of VB 2005 has a corresponding type in the .NET base class libraries, each of which exposes a fixed functionality. While I have not detailed each member of these core types, you are in a great position to dig into the details as you see fit. Be sure to consult the .NET Framework 2.0 SDK documentation for full details regarding the intrinsic .NET data types.

Narrowing (Explicit) and Widening (Implicit) Data Type Conversions

Now that you understand how to interact with intrinsic data types, let's examine the related topic of *data type conversion*. Assume you have a new console application (named `TypeConversions`) that defines the following module:

```
Module Program
    Sub Main()
        Console.WriteLine("***** The Amazing Addition Program *****")
        Dim a As Short = 9
        Dim b As Short = 10
        Console.WriteLine("a + b = {0}", Add(a, b))
    End Sub

    Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
        Return x + y
    End Function
End Module
```

Notice that the `Add()` method expects to be sent two `Integer` parameters. However, note that the `Main()` method is in fact sending in two `Short` variables. While this might seem like a complete and total mismatch of data types, the program compiles and executes without error, returning the expected result of 19.

The reason that the compiler treats this code as syntactically sound is due to the fact that there is no possibility for loss of data. Given that the maximum value of a `Short` (32,767) is well within the range of an `Integer` (2,147,483,647), the compiler automatically *widens* each `Short` to an `Integer`. Technically speaking, *widening* is the term used to define a safe “upward cast” that does not result in a loss of data.

Note In other languages (especially C-based languages such as C#, C++, and Java) “widening” is termed an *implicit cast*.

Table 3-6 illustrates which data types can be safely widened to a specific data types.

Table 3-6. *Safe Widening Conversions*

VB 2005 Type	Safely Widens to...
Byte	SByte, UInteger, Integer, ULong, Long, Single, Double, Decimal
SByte	SByte, Integer, Long, Single, Double, Decimal
Short	Integer, Long, Single, Double, Decimal
SByte	UInteger, Integer, ULong, Long, Single, Double, Decimal
Char	SByte, UInteger, Integer, ULong, Long, Single, Double, Decimal
Integer	Long, Double, Decimal
UInteger	Long, Double, Decimal
Long	Decimal
ULong	Decimal
Single	Double

Although this automatic widening worked in our favor for the previous example, other times this “automatic type conversion” can be the source of subtle and difficult-to-debug runtime errors. For example, assume that you have modified the values assigned to the a and b variables within Main() to values that (when added together) overflow the maximum value of a Short. Furthermore, assume you are storing the return value of the Add() method within a new local Short variable, rather than directly printing the result to the console:

```
Module Program
  Sub Main()
    Console.WriteLine("***** The Amazing Addition Program *****")
    Dim a As Short = 30000
    Dim b As Short = 30000
    Dim answer As Short = Add(a, b)
    Console.WriteLine("a + b = {0}", answer)
  End Sub

  Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
    Return x + y
  End Function
End Module
```

In this case, although your application compiles just fine, when you run the application you will find the CLR throws a runtime error; specifically a System.OverflowException, as shown in Figure 3-11.

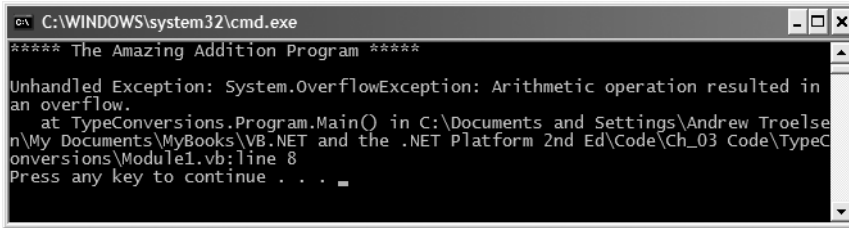


Figure 3-11. *Oops! The value returned from Add() was greater than the maximum value of a Short!*

The problem is that although the Add() method can return an Integer with the value 60,000 (as this fits within the range of an Integer), the value cannot be stored in a Short (as it overflows the bounds of this data type). In this case, the CLR attempts to apply a *narrowing operation*, which resulted in a runtime error. As you can guess, narrowing is the logical opposite of widening, in that a larger value is stored within a smaller variable.

Note In other languages (especially C-based languages such as C#, C++, and Java) “narrowing” is termed an *explicit cast*.

Not all narrowing conversions result in a System.OverflowException of course. For example, consider the following code:

```
' This narrowing conversion is a-OK.
Dim myByte As Byte
Dim myInt As Integer = 200
myByte = myInt
Console.WriteLine("Value of myByte: {0}", myByte)
```

Here, the value contained within the Integer variable myInt is safely within the range of a Byte, therefore the narrowing operation does not result in a runtime error. Although it is true that many narrowing conversions are safe and nondramatic in nature, you may agree that it would be ideal to trap narrowing conversions at *compile time* rather than *runtime*. Thankfully there is such a way, using the VB 2005 Option Strict directive.

Understanding Option Strict

Option Strict ensures compile-time (rather than runtime) notification of any narrowing conversion so it can be corrected in a timely fashion. If we are able to identify these narrowing conversions upfront, we can take a corrective course of action and decrease the possibility of nasty runtime errors.

A Visual Basic 2005 project, as well as specific *.vb files within a given project, can elect to enable or disable implicit narrowing via the Option Strict directive. When turning this option On, you are informing the compiler to check for such possibilities during the compilation process. Thus, if you were to add the following to the very top of your current file:

```
' Option directives must be the very first code statements in a *.vb file!
Option Strict On
```

you would now find a compile-time error for each implicit narrowing conversion, as shown in Figure 3-12.

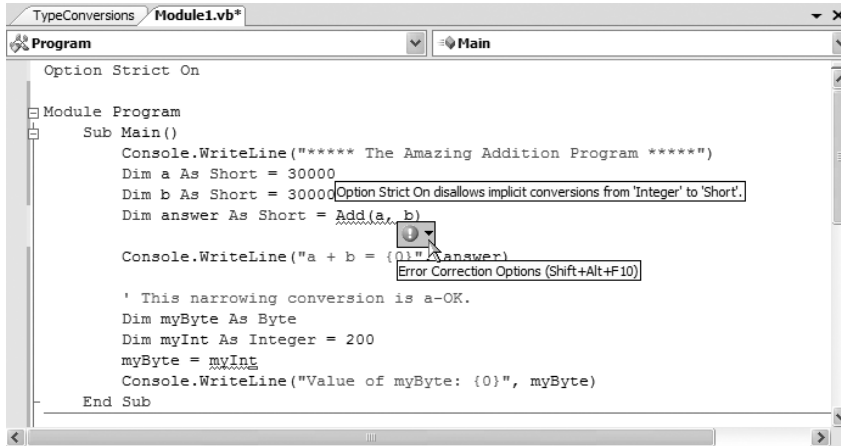


Figure 3-12. Option Strict *disables automatic narrowing of data*

Here, we have enabled Option Strict on a single file within our project. This approach can be useful when you wish to selectively allow narrowing conversions within specific *.vb files. However, if you wish to enable Option Strict for each and every file in your project, you can do so using the Compile tab of the My Project dialog box, as shown in Figure 3-13.

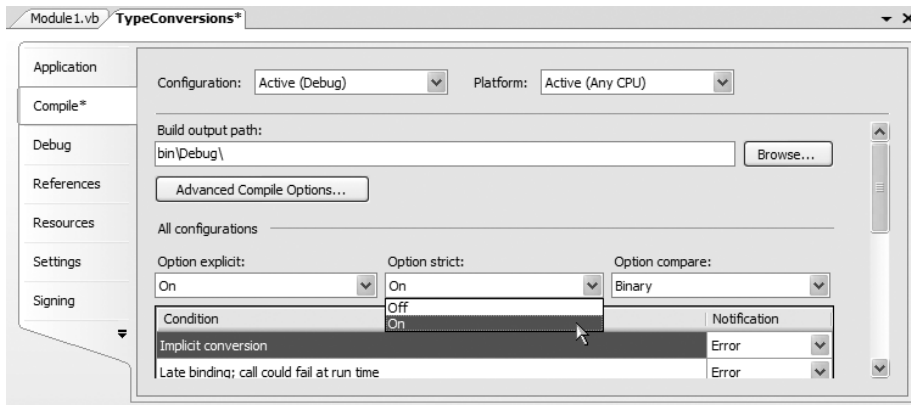


Figure 3-13. Enabling Option Strict on the project level

Note Under Visual Studio 2005, Option Strict is disabled for new Visual Basic 2005 projects. I would recommend, however, that you always enable this setting for each application you are creating, given that it is far better to resolve problems at compile time than runtime!

Now that we have some compile-time checking, we are able to resolve the error using one of two approaches. The most direct (and often more favorable) choice is to simply redefine the variable to a data type that will safely hold the result. For example:

```

Sub Main()
    Console.WriteLine("***** The Amazing Addition Program *****")
    Dim a As Short = 30000
    Dim b As Short = 30000

    ' Nope...Dim answer As Short = Add(a, b)
    Dim answer As Integer = Add(a, b)
    Console.WriteLine("a + b = {0}", answer)
    ...
End Sub

```

Another approach is to make use of an explicit Visual Basic 2005 type conversion function, such as `CByte()` for this example:

```

Sub Main()
    ...
    Dim myByte As Byte
    Dim myInt As Integer = 200
    ' myByte = myInt
    myByte = CByte(myInt)
    Console.WriteLine("Value of myByte: {0}", myByte)
End Sub

```

Note Another useful Option statement is `Option Explicit`. When enabled, the compiler demands that all variables are defined using a proper `As` clause (e.g., `Dim a As Integer` rather than `Dim A`). I would recommend you always enable `Option Explicit`, as this can pinpoint many otherwise unseen programming errors.

Explicit Conversion Functions

Visual Basic 2005 provides a number of conversion functions in addition to `CByte()` that enable you to explicitly allow a narrowing cast when `Option Strict` is enabled. Table 3-7 documents the core VB 2005 conversion functions.

Table 3-7. VB 2005 Conversion Functions

Conversion Function	Meaning in Life
<code>CBool</code>	Converts a Boolean expression into a Boolean value
<code>CByte</code>	Makes an expression a Byte
<code>CChar</code>	Makes the first character of a string into a Char
<code>CDate</code>	Makes a string containing a data expression into a Date
<code>CDbl</code>	Makes a numeric expression double precision
<code>CDec</code>	Makes a numeric expression of the Decimal type
<code>CInt</code>	Makes a numeric expression an Integer by rounding
<code>CLng</code>	Makes a numeric expression a long integer by rounding
<code>CObj</code>	Makes any item into an Object
<code>SByte</code>	Makes a numeric expression into an SByte by rounding
<code>CShort</code>	Makes a numeric expression into a Short by rounding
<code>CSingle</code>	Makes a numeric expression into a Single
<code>CStr</code>	Returns a string representation of the expression
<code>CUInt</code>	Makes a numeric expression into a UInteger by rounding
<code>CULng</code>	Makes a numeric expression into a ULong
<code>CUShort</code>	Makes a numeric expression into a UShort

In addition to these data type–specific conversion functions, with the release of the .NET platform, the Visual Basic language also supports the `CType` function. `CType` takes two arguments, the first is the “thing you have,” while the second is the “thing you want.” For example, the following conversions are functionally equivalent:

```
Sub Main()
...
    Dim myByte As Byte
    Dim myInt As Integer = 200
    myByte = CByte(myInt)
    myByte = CType(myInt, Byte)
    Console.WriteLine("Value of myByte: {0}", myByte)
End Sub
```

One benefit of the `CType` function is that it handles all the conversions of the (primarily VB 6.0-centric) conversion functions shown in Table 3-7. Furthermore, as you will see later in this text, `CType` allows you to convert between base and derived classes, as well as objects and their implemented interfaces.

Note As you will see in Chapter 11, VB 2005 provides two new alternatives to `CType`—`DirectCast` and `TryCast`. However, they can only be used if the arguments are related by inheritance or interface implementation.

The Role of `System.Convert`

To wrap up the topic of data type conversions, I'd like to point out the fact that the `System` namespace defines a class named `Convert` that can also be used to widen or narrow a data assignment:

```
Sub Main()
...
    Dim myByte As Byte
    Dim myInt As Integer = 200
    myByte = CByte(myInt)
    myByte = CType(myInt, Byte)
    myByte = Convert.ToByte(myInt)
    Console.WriteLine("Value of myByte: {0}", myByte)
End Sub
```

One benefit of using `System.Convert` is that it provides a language-neutral manner to convert between data types. However, given that Visual Basic 2005 provides numerous built-in conversion functions (`CBool`, `CByte`, and the like), using the `Convert` type to do your data type conversions is usually nothing more than a matter of personal preference.

Source Code The `TypeConversions` project is located under the Chapter 3 subdirectory.

Building Visual Basic 2005 Code Statements

As a software developer, you are no doubt aware that a *statement* is simply a line of code that can be processed by the compiler (without error, of course). For example, you have already seen how to craft a local variable declaration statement over the course of this chapter:

```
' VB 2005 variable declaration statements.
Dim i As Integer = 10
Dim j As System.Int32 = 20
Dim k As New Integer()
```


On a related note, you have also previewed the correct syntax to declare a Function using the syntax of VB 2005:

```
Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
    Return x + y
End Function
```

While it is true that when you are comfortable with the syntax of your language of choice, you tend to intuitively know what constitutes a valid statement, there are two idioms of VB 2005 code statements that deserve special mention to the uninitiated.

The Statement Continuation Character

White space (meaning blank lines of code) are ignored by the VB 2005 compiler unless you attempt to define a single code statement over multiple lines of code. This is quite different from C-based languages, where white space never matters, given that languages in the C family explicitly mark the end of a statement with a semicolon and scope with curly brackets.

In this light, the following two C# functions are functionally identical (although the second version is hardly readable and very bad style!):

```
// C# Add() method take one.
public int Add(int x, int y)
{ return x + y; }
```

```
// C# Add() method take two.
public int Add(
    int x, int y) { return x
    +
    y; }
```

Under Visual Basic 2005, if you wish to define a statement or member over multiple lines of code, you must split each line using the under bar (_) token, formally termed the *statement continuation character*. Furthermore, there *must* be a blank space on each side of the statement continuation character. Thus:

```
' VB 2005 Add() method take one.
Function Add(ByVal x As Integer, _
    ByVal y As Integer) As Integer
    Return x + y
End Function
```

```
' VB 2005 Add() method take two.
Function Add(ByVal x As Integer, _
    ByVal y As Integer) _
    As Integer
    Return x + y
End Function
```

```
' VB 2005 Add() method take three.
Function Add(ByVal x As Integer, _
    ByVal y As Integer) _
    As Integer
    Return x + y
End _
Function
```

Of course, you would never use the statement continuation character as shown in the last iteration of the Add() method, as the code is less than readable. In the real world, this feature is most helpful when defining a member that takes a great number of arguments, to space them out in such a way that you can view them within your editor of choice (rather than scrolling the horizontal scroll bar to see the full prototype!).

Defining Multiple Statements on a Single Line

Sometimes it is convenient to define multiple code statements on a single line of code within the editor. For example, assume you have a set of local variables that need to be assigned to initial values. While you could assign a value to each variable on discrete lines of code:

```
Sub MyMethod()  
    Dim s As String  
    Dim i As Integer  
    s = "Fred"  
    i = 10  
End Sub
```

you can compact the scope of this subroutine using the colon character:

```
Sub MyMethod()  
    Dim s As String  
    Dim i As Integer  
    s = "Fred" : i = 10  
End Sub
```

Understand that misuse of the colon can easily result in hard-to-read code. As well, when combined with the statement continuation character, you can end up with nasty statements such as the following:

```
Sub MyMethod()  
    Dim s As String : Dim i As Integer  
    s = "Fred" _  
    : i = 10  
End Sub
```

To be sure, defining multiple statements on a single line using the colon character should be used sparingly. For the most part, this language feature is most useful when you need to make simple assignments to multiple variables.

VB 2005 Flow-control Constructs

Now that you can define a single simple code statement, let's examine the flow-control keywords that allow you to alter the flow of your program and several keywords that allow you to build complex code statements using the *And*, *Or*, and *Not* operators.

Like other programming languages, VB 2005 defines several ways to make runtime decisions regarding how your application should function. In a nutshell, we are offered the following flow-control constructs:

- The *If/Then/Else* statement
- The *Select/Case* statement

The If/Then/Else Statement

First up, you have your good friend, the *If/Then/Else* statement. In the simplest form, the *If* construct does not have a corresponding *Else*. Within the *If* statement, you will construct an expression that can resolve to a Boolean value. For example:

```
Sub Main()  
    Dim userDone As Boolean
```

```

' Gather user input to assign
' Boolean value...

If userDone = True Then
    Console.WriteLine("Thanks for running this app")
End If
End Sub

```

A slightly more complex If statement can involve any number of Else statements to account for a range of values set to the expression being tested against:

```

Sub Main()
    Dim userOption As String

    ' Read user option from command line.
    userOption = Console.ReadLine()

    If userOption = "GodMode" Then
        Console.WriteLine("You will never die...cheater!")
    ElseIf userOption = "FullLife" Then
        Console.WriteLine("At the end, heh?")
    ElseIf userOption = "AllAmmo" Then
        Console.WriteLine("Now we can rock and roll!")
    Else
        Console.WriteLine("Unknown option...")
    End If
End Sub

```

Note that any secondary “else” condition is marked with the ElseIf keyword, while the final condition is simply Else.

Building Complex Expressions

The expression tested against within a flow-control construct need not be a simple assignment. If required, you are able to leverage the VB 2005 equality/relational operators listed in Table 3-8.

Table 3-8. *VB 2005 Relational and Equality Operators*

VB 2005 Equality/Relational Operator	Example Usage	Meaning in Life
=	If age = 30 Then	Returns true only if each expression is the same
<>	If "Foo" <> myStr Then	Returns true only if each expression is different
<	If bonus < 2000 Then	Returns true if expression A is less than, greater than, less than or equal to, or greater than or equal to expression B, respectively
>	If bonus > 2000 Then	
<=	If bonus <= 2000 Then	
>=	If bonus >= 2000 Then	

Note Unlike C-based languages, the VB 2005 = token is used to denote both assignment and equality semantics (therefore VB 2005 does not supply a == operator).

In addition, you may build a complex expression to test within a flow-control construct using the code conditional operators (also known as the logical operators) listed in Table 3-9. This table outlines the most common conditional operators of the language.

Table 3-9. *VB 2005 Conditional Operators*

VB 2005 Conditional Operator	Example	Meaning in Life
And	If age = 30 And name = "Fred" Then	Conditional AND operator, where both conditions must be True for the condition to be True
AndAlso	If age = 30 AndAlso name = "Fred" Then	A conditional AND operator that supports <i>short-circuiting</i> , meaning if the first expression is False, the second expression is not evaluated
Or	If age = 30 Or name = "Fred" Then	Conditional OR operator
OrElse	If age = 30 OrElse name = "Fred" Then	Conditional OR operator that supports <i>short-circuiting</i> , meaning if either expression is True, True is returned
Not	If Not myBool Then	Conditional NOT operator

As I am assuming you have prior experience in BASIC or C-based languages, I won't belabor the use of these operators. If you require additional details beyond the following code snippet, I will assume you will consult the .NET Framework SDK documentation. However, here is a simple example:

```
Sub Main()
    Dim userOption As String
    Dim userAge As Integer

    ' Read user option from command line.
    userOption = Console.ReadLine()
    userAge = Console.ReadLine()

    If userOption = "AdultMode" And userAge >= 21 Then
        Console.WriteLine("We call this Hot Coffee Mode...")
    ElseIf userOption = "AllAmmo" Then
        Console.WriteLine("Now we can rock and roll!")
    Else
        Console.WriteLine("Unknown option...")
    End If
End Sub
```

The Select/Case Statement

The other selection construct offered by VB 2005 is the Select statement. This can be a more compact alternative to the If/Then/Else statement when you wish to handle program flow based on a known set of choices. For example, the following `Main()` method prompts the user for one of three known values. If the user enters an unknown value, you can account for this using the `Case Else` statement:

```

Sub Main()
    ' Prompt user with choices.
    Console.WriteLine("Welcome to the world of .NET")
    Console.WriteLine("1 = C# 2 = Managed C++ (MC++) 3 = VB 2005")
    Console.Write("Please select your implementation language: ")

    ' Get choice.
    Dim s As String = Console.ReadLine()
    Dim n As Integer = Integer.Parse(s)

    ' Based on input, act accordingly...
    Select Case n
        Case Is = 1
            Console.WriteLine("C# is all about managed code.")
        Case Is = 2
            Console.WriteLine("Maintaining a legacy system, are we?")
        Case Is = 3
            Console.WriteLine("VB 2005: Full OO capabilities...")
        Case Else
            Console.WriteLine("Well...good luck with that!")
    End Select
End Sub

```

VB 2005 Iteration Constructs

All programming languages provide ways to repeat blocks of code until a terminating condition has been met. Regardless of which language you are coming from, the VB 2005 iteration statements should cause no raised eyebrows and require little explanation. In a nutshell, VB 2005 provides the following iteration constructs:

- For/Next loop
- For/Each loop
- Do/While loop
- Do/Until loop
- With loop

Let's quickly examine each looping construct in turn. Do know that I will only concentrate on the core features of each construct. I'll assume that you will consult the .NET Framework 2.0 SDK documentation if you require further details.

For/Next Loop

When you need to iterate over a block of code statements a fixed number of times, the For statement is the looping construct of champions. In essence, you are able to specify how many times a block of code repeats itself, using an expression that will evaluate to a Boolean:

```

Sub Main()
    ' Prints out the numbers 5 - 25, inclusive.
    Dim i As Integer
    For i = 5 To 25
        Console.WriteLine("Number is: {0}", i)
    Next
End Sub

```

One nice improvement to the For looping construct is we are now able declare the counter variable directly within the For statement itself (rather than in a separate code statement). Therefore, the previous code sample could be slightly streamlined as the following:

```
Sub Main()
    ' A slightly simplified For loop.
    For i As Integer = 5 To 25
        Console.WriteLine("Number is: {0}", i)
    Next
End Sub
```

The For loop can also make use of the Step keyword to offset the value of the counter. For example, if you wish to increment the counter variable by five with each iteration, you would do so with the following:

```
Sub Main()
    ' Increment i by 5 with each pass.
    For i As Integer = 5 To 25 Step 5
        Console.WriteLine("Number is: {0}", i)
    Next
End Sub
```

For/Each Loop

The For/Each construct is a variation of the standard For loop, where you are able to iterate over the contents of an array without the need to explicitly monitor the container's upper limit (as in the case of a traditional For/Next loop). Assume you have defined an array of String types and wish to print each item to the command window (VB 2005 array syntax will be fully examined in the next chapter). In the following code snippet, note that the For Each statement can define the type of item iterated over directly within the statement:

```
Sub Main()
    Dim myStrings() As String = _
        {"Fun", "with", "VB 2005"}

    For Each str As String In myStrings
        Console.WriteLine(str)
    Next
End Sub
```

or on discrete lines of code:

```
Sub Main()
    Dim myStrings() As String = _
        {"Fun", "with", "VB 2005"}

    Dim item As String
    For Each item In myStrings
        Console.WriteLine(item)
    Next
End Sub
```

In these examples, our counter was explicitly defined as a String data type, given that our array is full of strings as well. However, if you wish to iterate over an array of Integers (or any other type), you would simply define the counter in the terms of the items in the array. For example:

```

Sub Main()
    ' Looping over an array of Integers.
    Dim myInts() As Integer = _
        {10, 20, 30, 40}

    For Each int As Integer In myInts
        Console.WriteLine(int)
    Next
End Sub

```

Note The `For Each` construct can iterate over any types that support the correct infrastructure. I'll hold off on the details until Chapter 9, as this aspect of the `For Each` loop entails an understanding of interface-based programming and the system-supplied `IEnumerator` and `IEnumerable` interfaces.

Do/While and Do/Until Looping Constructs

You have already seen that the `For/Next` statement is typically used when you have some foreknowledge of the number of iterations you want to perform (e.g., $j > 20$). The `Do` statements, on the other hand, are useful for those times when you are uncertain how long it might take for a terminating condition to be met (such as when gathering user input).

`Do/While` and `Do/Until` are (in many ways) interchangeable. `Do/While` keeps looping until the terminating condition is *false*. On the other hand, `Do/Until` keeps looping until the terminating condition is *true*. For example:

```

' Keep looping until X is not equal to an empty string.
Do
    ' Some code statements to loop over.
Loop Until X <> ""

' Keep looping as long as X is equal to an empty string.
Do
    ' Some code statements to loop over.
Loop While X = ""

```

Note that in these last two examples, the test for the terminating condition was placed at the end of the `Loop` keyword. Using this syntax, you can rest assured that the code within the loop will be executed at least once (given that the test to exit the loop occurs after the first iteration). If you prefer to allow for the possibility that the code within the loop may never be executed, move the `Until` or `While` clause to the beginning of the loop:

```

' Keep looping until X is not equal to an empty string.
Do Until X <> ""
    ' Some code to loop over.
Loop

' Keep looping as long as X is not equal to an empty string.
Do While X = ""
    ' Some code to loop over.
Loop

```

Finally, understand that VB 2005 still supports the raw `While` loop. However, the `Wend` keyword has been replaced with a more fitting `End While`:

```
Dim j As Integer
While j < 20
    Console.Write(j & ", ")
    j += 1
End While
```

The With Construct

To wrap this chapter up, allow me to say that the VB 6.0 With construct is still supported under VB 2005. In a nutshell, the With keyword allows you to invoke members of a type within a predefined scope. Do know that the With keyword is nothing more than a typing time saver.

For example, the System.Collections namespace has a type named ArrayList, which like any type has a number of members. You are free to manipulate the ArrayList on a statement by statement basis as follows:

```
Sub Main()
    Dim myStuff As New ArrayList()
    myStuff.Add(100)
    myStuff.Add("Hello")
    Console.WriteLine("Size is: {0}", myStuff.Count)
End Sub
```

or use the VB 2005 With keyword:

```
Sub Main()
    Dim myStuff As New ArrayList()
    With myStuff
        .Add(100)
        .Add("Hello")
        Console.WriteLine("Size is: {0}", .Count)
    End With
End Sub
```

Summary

Recall that the goal of this chapter was to expose you to numerous core aspects of the VB 2005 programming language. Here, we examined the constructs that will be commonplace in any application you may be interested in building. After examining the Module type, you learned that every VB 2005 executable program must have a type defining a Main() method, which serves as the program's entry point. Within the scope of Main(), you typically create any number of objects that work together to breathe life into your application.

Next, we dove into the details of the built-in data types of VB 2005, and came to understand that each data type keyword (e.g., Integer) is really a shorthand notation for a full-blown type in the System namespace (System.Int32 in this case). Given this, each VB 2005 data type has a number of built-in members. Along the same vein, you also learned about the role of *widening* and *narrowing* as well as the role of Option Strict.

We wrapped up by checking out the various iteration and decision constructs supported by VB 2005. Now that you have some of the basic nuts-and-bolts in your mind, the next chapter completes our examination of core language features.