Object-Oriented Programming with Visual Basic.NET

Michael McMillan
Pulaski Technical College, North Little Rock, Arkansas

Cambridge University Press

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2004

Printed in the United States of America

Typefaces ITC Berkeley Oldstyle 11/13.5 pt. and ITC Franklin Gothic *System* LaTeX 2ε [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data
McMillan, Michael, 1957–
Object-oriented programming with Visual Basic.Net / Michael McMillan
p. cm.
Includes bibliographical references and index.
ISBN 0-521-53983-8 (pb.)
1. Object-oriented programming (Computer science) 2. Microsoft Visual BASIC.
3. BASIC (Computer program language) 4. Microsoft.NET. I. Title
QA76.64.M389 2004
005.1′17 – dc22 2003055949

ISBN 0 521 53983 8 paperback

Contents

CHAPTER 1

An Overview of the Visual Basic.NET Language

This chapter presents an overview of the syntax and primary constructs of the Visual Basic.NET (VB.NET) language for programmers unfamiliar with VB.NET. This is not a tutorial chapter, however, so if you are new to programming you should study another text on VB.NET before continuing with this book. If, though, you are coming to VB.NET from some other language, such as C++ or Java or even Visual Basic 6, you should read through this chapter to familiarize yourself with the language.

.NET Programs

There are two ways to build programs in VB.NET. One is to use the Visual Studio.NET Integrated Development Environment (IDE). The other is to use the command-line compiler packaged as part of the .NET Framework Software Development Kit (SDK). In this section we'll discuss developing programs with the command-line compiler, since this software is free and can run on any of the modern Windows operating sysems (Windows 98 and beyond).

VB.NET Program Types

With VB.NET, you can write many different kinds of programs. A VB.NET program that makes use of a graphical user interface (GUI) is a Windows application. A VB.NET program that uses the command-prompt console for input and output is called a Console application. You can also write Internet applications, Windows Services applications, and other types of applications. In this book we will focus on Console and Windows applications, though we will look at examples of Windows Services and Internet (ASP.NET) applications in the last few chapters.

Writing a Console Application Using the Command-Line Compiler

You do not have to be running Visual Studio.NET to compile and run VB.NET programs. A command-line compiler is shipped with the .NET Framework and can be used for any VB.NET programs you want to develop.

To get to the compiler, find the Microsoft.NET subdirectory. It is usually found in the Winnt or Windows (for Windows 98) directory. Then change directories to the Framework subdirectory. The compiler resides in yet another subdirectory. The name of the subdirectory depends on which version of the .NET Framework you are using. The current .NET Framework version stores the compiler in the v1.0.3705 subdirectory, but be sure to check this on your own system since your version may be different. The path to the compiler for a typical computer running Windows 2000 is c:\winnt\Microsoft.NET\Framework\v1.0.3705.

Using the compiler is quite simple. First, create a source file using the text editor of your choice. Make sure the file you create has a .vb extension. Let's look at an example of a simple VB.NET program, a program that displays the text "Hello, world!" on the screen:

```
Imports System
Module HelloWorld
  Sub Main()
    Console.WriteLine("Hello, world!")
  End Sub
End Module
```

The first line indicates that the program needs to use a class found in the System *namespace*. A namespace is a tool used to group related classes and other types together. Namespaces also allow different classes to share the same name. Using the keyword Imports allows us to use a class from the specified namespace (System in this case) without using the namespace name first. We can just as easily leave the first line of the program out altogether and type in the fully qualified name of the class:

```
System.Console.WriteLine("Hello, world!")
```

Generally, importing a namespace makes your programs easier to write and easier to read.

The next line defines a module named HelloWorld. A *module* is one of the possible packages into which we can write code that we want to compile and execute. Another package we can use is a class. Generally, though, we want to save the use of classes for defining our own custom types, so we'll use modules for writing Console applications in this book. Modules are begun with the Module keyword and are closed with the line End Sub.

The first line inside the Module definition defines a subroutine called Main. This subroutine is the entry point of the application, and the compiler will report an error if Main is not found somewhere in a module or class. If you are using a class rather than a module as the packaging for your application, Main must be defined as a Shared method, which means that the class does not have to be instantiated for the code to be executed. We'll explain later in the book what we mean by a Shared method. Main must be closed with the line End Sub.

The line that displays the message "Hello, world!" on the display is

```
Console.WriteLine("Hello, world!")
```

To display text on the computer's console, you have to call the Console class and the proper method for writing text to the console, one of which is the WriteLine method. This method displays the text passed to it as the argument on the console and then writes a newline character so that any more text will be written on the next line.

To end this section, we'll look at writing the same HelloWorld program as a class rather than as a module. The codes are similar, and to be honest, the two techniques are virtually identical. However, because in this book we

use classes to define special types, we'll write all our Console applications as modules.

Here's the HelloWorld class code:

```
Imports System
Class HelloWorld
  Shared Sub Main()
    Console.WriteLine("Hello, world!")
  End Sub
End class
```

To compile your program (assuming the source file name is test.vb), issue the following command:

```
vbc test.vb
```

If your program compiles successfully, you can simply run the executable file (test.exe) to run it. If your program has errors in it, the compiler will return the errors to your console.

Writing a Windows Application Using the Command-Line Compiler

One of the surprising things about VB.NET is that you don't have to use Visual Studio.NET to build a Windows application. Unlike previous versions of the language, VB.NET gives the programmer the ability to build a GUI directly from code. Although you probably won't want to use this feature all that often, there will be situations when building a GUI from scratch will be necessary. This is certainly true if you are using Windows 98 or ME and can't run Visual Studio.NET.

To demonstrate how to write a Windows application, we'll rewrite the HelloWorld program so that the text is displayed in a label on a form. First, let's look at the code:

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

Public Class HelloWorld
  Inherits Form
```

```
    Private lblHelloLabel As Label

    Public Shared Sub Main()
      Application.Run(New HelloWorld())
    End Sub

    Public Sub New()
      lblHelloLabel = New Label()
      With lblHelloLabel
        .Location = New Point(50, 50)
        .Size = New Size(392, 64)
        .Font = New Font("Courier", 24)
        .Text = "Hello, world!"
        .TabIndex = 0
        .TextAlign = ContentAlignment.TopCenter
      End With

      Me.Text = "A Hello, world! Windows Example"
      AutoScaleBaseSize = New Size(10, 20)
      FormBorderStyle = FormBorderStyle.FixedSingle
      ClientSize = New Size(599, 125)

      Controls.Add(lblHelloLabel)

    End Sub
  End Class
```

You'll notice first that there are two new namespaces imported into the program. These namespaces are needed for building Windows applications. The next line is just the definition of the class that holds the program. The following line

```
  Inherits Form
```

tells the compiler that the HelloWorld class is inheriting the Form class, which is found in the Systems.Windows.Forms namespace. *Inheritance* is a powerful technique in object-oriented programming and we will spend at least one chapter discussing it later in the book.

The next line declares a label for displaying the "Hello, world!" text. Following this declaration is the Main subroutine. Be sure to use the Shared modifier in the heading since we have to use a class for a Windows application.

The single line inside Sub Main is

```
Application.Run(New HelloWorld())
```

The Application class (which is part of System.Windows.Forms) includes the Run method, which performs the tasks necessary to run the HelloWorld program as a Windows application.

Following Sub Main is another subroutine definition—New. The New subroutine is a special type called a *constructor*. Constructors are used to create a new Class object. This process is called *instantiation* and every new Class object must be instantiated using a constructor. The code inside the constructor definition is run when the constructor is called, which in this program is the line

```
Application.Run(New HelloWorld())
```

Constructors are discussed in much more detail in Chapter 4.

Inside the constructor method are the details for displaying "Hello, world!" in a form. First, a new label is instantiated. We'll place our text inside this label. The next several lines set several of the label's properties, including the font type, the font size, and the location of the label. These lines are placed inside a With statement, a convenient shortcut to use when you need to make several changes to or perform other operations on the same object.

The line after the End With statement sets the caption of the current form. Since there isn't really a name for the form, we refer to it as Me. We'll see other uses for Me throughout the book.

The next three lines set some properties having to do with our form. The last line before the end of the subroutine adds the label to the form's Control collection. The program ends by closing off the subroutine definition and the class definition.

Windows applications are compiled a little differently than Console applications. The command to compile the HelloWorld program is as follows:

```
vbc HelloWorld.vb /reference:System.dll,System.Drawing._
 dll, System.Windows.Forms.dll /target:winexe
```

(Note that the command would be all one line when typed, but here it is broken into two lines for readability.)

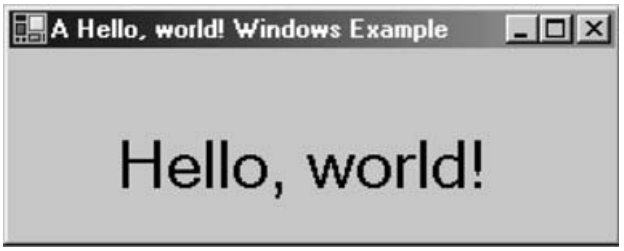

FIGURE 1.1. A Hello, World Windows Example

The first thing you notice is the switch—/reference. We have to add references to the different namespaces we use in this program for creating a Windows application. We didn't need this switch in the Console application because the compiler automatically includes the System.dll file. The other files (including System.dll), though, must be referenced specifically.

The last part of the command tells the compiler to build a Windows application (winexe). A Console application is compiled to just an .exe file. If you look at the file created by the compiler, though, it still displayed as wtest.exe. The compiler adds data internally to the file to enable it as a Windows application.

Now we're ready to run the program and examine the output (see Figure 1.1).

DATA TYPES AND VARIABLES

VB.NET contains the standard data types for storing numeric, string, character, and object values, as well as special data types for times, dates, and monetary values. The primary data types in VB.NET are the following:

- **Boolean:** True or False.
- **Byte:** 0–255 (unsigned).
- **Char:** 0–65535 (unsigned).
- **Date:** A date and time combination.
- **Decimal:** 0 through ±79,228,162,514,264,337,593,543,950,335 with no decimal point; 0 through ±7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is ±0.0000000000000000000000000001 (±1E−28).

- **Double:** −1.79769313486231570E+308 through −4.94065645841246544E−324 for negative values; 4.94065645841246544E−324 through 1.79769313486231570E+308 for positive values.
- **Integer:** −2,147,483,648 through 2,147,483,647.
- **Long:** −9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.
- **Object:** Any object.
- **Short:** −32,768 through 32,767.
- **Single:** −3.4028235E+38 through −1.401298E−45 for negative values; 1.401298E−45 through 3.4028235E+38 for positive values.
- **String:** 0 to approximately 2 billion Unicode characters.
- **Structure:** A user-defined type built from other data type components.

Variable Declaration

Variables are declared using the Dim keyword. For example,

```
Dim mySalary As Integer
Dim empID As String
```

The reason we use the Dim keyword when declaring a variable dates back to the early days of the Basic language. In those days, variables did not have to be declared; they could just pop into existence when needed. Arrays, however, had to declared first with the dimension of the array. The Dim keyword, then, identified a variable as an array and not just a plain variable. The use of Dim has continued through the many different versions of the language right up to VB.NET.

Multiple variables of the same type can be declared on the same line by separating each variable with a comma, like this:

```
Dim num1, num2, num3, num4 As Single
```

Initializers

An *initializer* is a variable declaration in which a value is also assigned to the variable. Initalizers are new to VB.NET, although many other languages have them. Here are some examples of initializers:

```
Dim salary As Integer = 35000
Dim lastName As String = "Durrwood"
```

Named Constants

A named constant is a variable whose value is assigned when it is declared and whose value cannot be changed. Named constants are often called "magic" values because they are usually used to represent important and/or frequently used values in a program.

Named constants are declared with the Const keyword. Here are some examples:

```
Const PI As Single = 3.14159
Const GREETING As String = "Hello, there."
Const LOGIN_CODE As String = :"letmein"
```

It is a common programming practice, though not a requirement of the VB.NET compiler, to use all uppercase letters when declaring a named constant. This helps these "magic" values stand out in your code so that they're easier to find.

Implicit Type Conversions and the Option Strict Switch

There are two ways to perform data type conversions in VB.NET. One way is to simply let the compiler do it for you. This is the easiest way and the one that is most likely to lead to both subtle and not-so-subtle errors in your programs. As an example, let's look at a simple code fragment that converts a Single value to an Integer:

```
Dim pi As Single = 3.14159
Dim intPi As Integer = pi
```

Because intPi is an Integer variable, when it is assigned the value of pi the compiler assigns the value 3 to the variable. This is called a *narrowing conversion* because the value 3.14159. . . is "narrowed" to 3 to fit in an Integer variable.

There are also *widening conversions*. When an Integer value is stored in a Single or Double variable, the value increases in size (widens) to hold the places to the right of the decimal point. Consider the following code fragment:

```
Dim intVal As Integer = 3
Dim dblVal As Double = intVal
```

Here an Integer variable storing the value 256 is assigned to a Double variable, so that the value 256.0 is stored in the Double. These types of conversions are called *implicit conversions* because the compiler performs the conversion behind the scenes.

Although implicit conversions are allowed, as just shown, that's not to say we should prefer allowing the compiler to make conversions for us. There will be situations when implicit conversions are made that are not what we want to happen, leading to logical errors or worse. The VB.NET compiler allows implicit conversions to take place when the Option Strict switch is off.

This switch tells the compiler whether or not to perform strict type checking. When Option Strict is off, implicit conversions will be performed; when Option Strict is on, a design-time error is flagged when an implicit conversion is attempted. Most, though certainly not all, programmers consider it good programming practice to set the Option Strict switch on so that any conversions that take place must be explicitly performed using a conversion function.

The Option Strict switch is set by writing either Option Strict On or Option Strict Off at the beginning of your program. In fact, the statement must precede any declarations or Imports statements, like this:

```
Option Strict On
Imports System
Module Module1
  Sub Main()
    ' Code here
  End Sub
End Module
```

One more word of caution on leaving the Option Strict switch off. It can lead to slower code. A simple example will illustrate the problem:

```
Dim n As Object
Dim names As String
For Each n In NameList
  names & = n & ","
Next
```

In this code, NameList is an ArrayList that holds a list of names. The loop builds a comma-delimited string of the names in the ArrayList. With Option Strict off, this code compiles and runs because the compiler will convert each value of n to a String before appending it to names. And that's the problem with leaving Option Strict off. Each conversion will take more time than necessary because the compiler has to perform a test of the data types and then perform the conversion. An explicit conversion via a conversion function will speed this up considerably. In the next section we'll examine how to perform explicit type conversions using VB.NET's type conversion functions.

Type Conversion Functions

VB.NET has a full set of built-in conversion functions for performing explicit type conversions. The following list shows each function and the type converted to:

- CBool: Boolean
- CByte: Byte
- CChar: Char
- CDate: Date
- CDbl: Double
- CDec: Decimal
- CInt: Integer
- CLng: Long
- CObj: Object
- CSng: Single
- CStr: String

Now let's look at some examples:

```
Salary = CInt(Console.ReadLine()) ' Converts console
                                  ' input to Integer
                                  ' value
```

```
Salary = CInt(txtSalary.Text) ' Does the same with a
                              ' textbox
taxRate = CDbl(5)
```

There are many other type conversions you can perform that are not as intuitive as these. For example, you can convert from a Boolean value to a String. The Boolean values True and False become "True" and "False" after the conversion. You can convert an Integer to Boolean—zero converts to False and a nonzero value converts to True.

Arrays

There are many times when you need to store related values within one variable name. Since regular variables only allow you to store one value in them at a time, you have to use something else—an *array*.

An array is a variable that stores multiple values of the same data type. Each value in an array (also called an element) is indexed by number. Arrays are created by specifying an array name, the number of elements to store, and the data type of the elements. The general form for an array declaration is

```
Dim array-name(n) As Data-type
```

Here are some array declaration examples:

```
Dim grades(9) As Integer
Dim names(39) As String
Dim averages(99) As Single
```

In VB.NET, as in of most other languages, the first index of an array is 0. For that reason, the number you use to declare the size of an array should always be one less than the total number of elements you want to store in the array. In the preceding examples, the grades array stores 10 elements, the names array stores 40 elements, and the averages array stores 100 elements.

An alternative way to declare an array is to provide an initialization list, which is a list of values to store in the array. The values are separated by commas and surrounded within curly braces. Here is an example:

```
Dim grades() As Integer = {65, 72, 83, 97}
```

The compiler automatically sizes the array based on the number of items in the initialization list. Putting a number inside the parentheses after the array name will lead to an exception.

Array objects are treated like class instances in VB.NET. There is a set of methods associated with arrays you can use in your programming. One of the most useful of these methods is GetUpperBound. This method returns the last index number (referencing the last element) in an array. You can use this method when looping through an array, which is demonstrated later in this chapter when we discuss repetition statements.

There are also array methods that perform tasks that used to take specially written code to perform, such as sorting an array and reversing an array. The two methods for these operations are Sort and Reverse. Here's an example:

```
Imports System
Module Array
  Sub Main()
    Dim names() As String = {"Mike", "Francis", "Ed", _
                             "Joan", "Terri"}
    names.Sort(names)
    Dim name As String
    For Each name In names
      Console.Write(name & " ")
    Next
    names.Reverse(names)
    Console.WriteLine()
    For Each name In names
      Console.Write(name & " ")
    Next
  End Sub
End Module
```

Multi-dimensional Arrays

Arrays are not limited to one dimension. You can create arrays of multiple dimensions, though it is uncommon to see arrays of more than three dimensions. The most common multidimensional arrays are two-dimensional arrays that model a table of data.

A two-dimensional array creates a set of data in the form of rows and columns. The rows make up the first, or 0th, dimension of the array, and the columns make up the second, or 1st, dimension of the array. The general form of a two-dimensional array declaration is

```
Dim array-name(rows, cols) As Data-type
```

For example, the following code declares an Integer array with five rows and six columns:

```
Dim nums(4,5) As Integer
```

You can also use an initialization list in a two-dimensional array declaration. Each dimension is delimited by curly braces and separated from each other by a column. Here's an example:

```
Dim grades(,) As Integer = {{76, 83, 91}, {100, 75, 66}}
```

Within the parentheses is a single comma. This comma indicates to the compiler that the array should be created with two dimensions. An array created with three dimensions would have two commas.

Array Element Access

Array elements are accessed by referencing their position in the array by index number. For example, the 0th element of a single-dimensional array named grades is accessed like this:

```
current_grade = grades(0)
```

Accessing an element in a two-dimensional array is similar:

```
current_grade = grades(1, 3)   ' Accesses grade in second
                               ' row, fourth column
```

You can assign data to an array element in the same way: