

Visual Basic 2008 Recipes: A Problem-Solution Approach

Copyright © 2008 by Todd Herman, Allen Jones, Matthew MacDonald, Rakesh Rajan

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): 978-1-59059-970-9

ISBN-10 (pbk): 1-59059-970-5

ISBN-13 (electronic): 978-1-4302-0604-0

ISBN-10 (electronic): 1-4302-0604-7

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: Jonathan Gennick

Technical Reviewer: Damien Foggon

Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Richard Dal Porto

Copy Editor: Kim Wimpsett

Associate Production Director: Kari Brooks-Copony

Production Editor: Katie Stence

Compositor: Susan Glinert Stevens

Proofreader: Liz Welch

Indexer: Broccoli Information Services

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

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

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

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>.



Application Development

This chapter covers some of the general features and functionality found in Visual Basic .NET 9.0 and Visual Studio 2008. The recipes in this chapter cover the following:

- Using the VB .NET command-line compiler to build console and Windows Forms applications (recipes 1-1 and 1-2)
- Creating and using code modules and libraries (recipes 1-3 and 1-4)
- Compiling and embedding a string resource file (recipe 1-5)
- Compiling applications using MSBuild.exe (recipe 1-6)
- Accessing command-line arguments from within your applications (recipe 1-7)
- Using compiler directives and attributes to selectively include code at build time (recipe 1-8)
- Manipulating the appearance of the console (recipe 1-9)
- Accessing program elements built in other languages whose names conflict with VB .NET keywords (recipe 1-10)
- Giving assemblies strong names and verifying strong-named assemblies (recipes 1-11, 1-12, 1-13, and 1-14)
- Signing an assembly with a Microsoft Authenticode digital signature (recipes 1-15 and 1-16)
- Managing the shared assemblies that are stored in the global assembly cache (recipe 1-17)
- Making your assembly more difficult to decompile (recipe 1-18)
- Understanding the basic functionality required to use Language Integrated Query (LINQ) (recipes 1-19, 1-20, 1-21, 1-22, and 1-23)

Note All the tools discussed in this chapter ship with the Microsoft .NET Framework or the .NET Framework software development kit (SDK). The tools that are part of the .NET Framework are in the main directory for the version of the framework you are running. For example, they are in the directory C:\WINDOWS\Microsoft.NET\Framework\v3.5 if you install version 3.5 of the .NET Framework to the default location. The .NET installation process automatically adds this directory to your environment path.

The tools provided with the SDK are in the Bin subdirectory of the directory in which you install the SDK, which is C:\Program Files\Microsoft Visual Studio 9.0\SDK\v3.5 if you chose the default path during the installation of Microsoft Visual Studio 2008. This directory is *not* added to your path automatically, so you must manually edit your path in order to have easy access to these tools. Your other option is to use the Visual Studio 2008 Command Prompt shortcut that is located under the Microsoft Visual Studio 2008/Visual Studio Tools folder in the Windows Start menu. This will launch vcvarsall.bat, which will set the right environment variables and open the command prompt. Most of the tools support short and long forms of the command-line switches that control their functionality. This chapter always shows the long form, which is more informative but requires additional typing. For the shortened form of each switch, see the tool's documentation in the .NET Framework SDK.

Also, as a final note, if you are using Windows Vista, you should be sure to run all command-line utilities using Run As Administrator, or some of them might not function properly. Doing this will still result in numerous dialog boxes requesting that you ensure you approve of the request to use administrative rights; you must respond to these dialog boxes by clicking Yes.

1-1. Create a Console Application from the Command Line

Problem

You need to use the VB .NET command-line compiler to build an application that does not require a Windows graphical user interface (GUI) but instead displays output to, and reads input from, the Windows command prompt (console).

Solution

In one of your classes, ensure you implement a Shared method named Main with one of the following signatures:

```
Public Shared Sub Main()  
End Sub  
Public Shared Sub Main(ByVal args As String())  
End Sub  
Public Shared Function Main() As Integer  
End Sub  
Public Shared Function Main(ByVal args As String()) As Integer  
End Sub
```

Build your application using the VB .NET compiler (vbc.exe) by running the following command (where HelloWorld.vb is the name of your source code file):

```
vbc /target:exe HelloWorld.vb
```

Note If you own Visual Studio, you will most often use the Console Application project template to create new console applications. However, for small applications, it is often just as easy to use the command-line compiler. It is also useful to know how to build console applications from the command line if you are ever working on a machine without Visual Studio and want to create a quick utility to automate some task.

How It Works

By default, the VB .NET compiler will build a console application unless you specify otherwise. For this reason, it's not necessary to specify the `/target:exe` switch, but doing so makes your intention clearer, which is useful if you are creating build scripts that will be used by others or will be used repeatedly over a period of time.

To build a console application consisting of more than one source code file, you must specify all the source files as arguments to the compiler. For example, the following command builds an application named `MyFirstApp.exe` from two source files named `HelloWorld.vb` and `ConsoleUtils.vb`:

```
vbc /target:exe /main:HelloWorld /out:MyFirstApp.exe HelloWorld.vb ConsoleUtils.vb
```

The `/out` switch allows you to specify the name of the compiled assembly. Otherwise, the assembly is named after the first source file listed—`HelloWorld.vb` in the example. If classes in both the `HelloWorld` and `ConsoleUtils` files contain `Main` methods, the compiler cannot automatically determine which method represents the correct entry point for the assembly. Therefore, you must use the compiler's `/main` switch to identify the name of the class that contains the correct entry point for your application. When using the `/main` switch, you must provide the fully qualified class name (including the namespace); otherwise, you will receive the following:

```
vbc : error BC30420: 'Sub Main' was not found in 'HelloWorld'
```

If you have a lot of VB .NET code source files to compile, you should use a response file. This simple text file contains the command-line arguments for `vbc.exe`. When you call `vbc.exe`, you give the name of this response file as a single parameter prefixed by the `@` character. Here is an example:

```
vbc @commands.rsp
```

To achieve the equivalent of the previous example, `commands.rsp` would contain this:

```
/target:exe /main:HelloWorld /out:MyFirstApp.exe HelloWorld.vb ConsoleUtils.vb
```

For readability, response files can include comments (using the `#` character) and can span multiple lines. The VB .NET compiler also allows you to specify multiple response files by providing multiple parameters that are prefixed with the `@` character.

The Code

The following code lists a class named `ConsoleUtils` that is defined in a file named `ConsoleUtils.vb`:

```
Imports System
```

```
Namespace Apress.VisualBasicRecipes.Chapter01
    Public Class ConsoleUtils
```

```

' This method will display a prompt and read a response from the console.
Public Shared Function ReadString(ByVal message As String) As String

    Console.Write(message)
    Return Console.ReadLine

End Function

' This method will display a message on the console.
Public Shared Sub WriteString(ByVal message As String)

    Console.WriteLine(message)

End Sub

' This method is used for testing ConsoleUtility methods.
' While it is not good practice to have multiple Main
' methods in an assembly, it sometimes can't be avoided.
' You specify in the compiler which Main sub routine should
' be used as the entry point. For this example, this Main
' routine will never be executed.
Public Shared Sub Main()

    ' Prompt the reader to enter a name.
    Dim name As String = ReadString("Please enter a name: ")

    ' Welcome the reader to Visual Basic 2008 Recipes.
    WriteString("Welcome to Visual Basic 2008 Recipes, " & name)

End Sub

End Class
End Namespace

The HelloWorld class listed next uses the ConsoleUtils class to display the message “Hello,
World” to the console (HelloWorld is contained in the HelloWorld.vb file):

Imports System

Namespace Apress.VisualBasicRecipes.Chapter01
    Public Class HelloWorld

        Public Shared Sub Main()

            ConsoleUtils.WriteString("Hello, World")
            ConsoleUtils.WriteString(vbCrLf & "Main method complete. Press Enter.")
            Console.ReadLine()

        End Sub

    End Class
End Namespace

```

Usage

To build HelloWorld.exe from the two source files, use the following command:

```
vbc /target:exe /main:Apres.VizualBasicRecipes.Chapter01.HelloWorld ➡  
/out:HelloWorld.exe ConsoleUtils.vb HelloWorld.vb
```

1-2. Create a Windows-Based Application from the Command Line

Problem

You need to use the VB .NET command-line compiler to build an application that provides a Windows Forms-based GUI.

Solution

Create a class that inherits from the `System.Windows.Forms.Form` class. (This will be your application's main form.) In one of your classes, ensure you implement a Shared method named `Main`. In the `Main` method, create an instance of your main form class and pass it to the Shared method `Run` of the `System.Windows.Forms.Application` class. Build your application using the command-line VB .NET compiler, and specify the `/target:winexe` compiler switch.

How It Works

Building an application that provides a simple Windows GUI is a world away from developing a full-fledged Windows-based application. However, you must perform certain tasks regardless of whether you are writing the Windows equivalent of “Hello, World” or the next version of Microsoft Word, including the following:

- For each form you need in your application, create a class that inherits from the `System.Windows.Forms.Form` class.
- In each of your form classes, declare members that represent the controls that will be on that form, such as buttons, labels, lists, and text boxes. These members should be declared `Private` or at least `Protected` so that other program elements cannot access them directly. If you need to expose the methods or properties of these controls, implement the necessary members in your form class, providing indirect and controlled access to the contained controls.
- Declare methods in your form class that will handle events raised by the controls contained by the form, such as button clicks or key presses when a text box is the active control. These methods should be `Private` or `Protected` and follow the standard .NET event pattern (described in recipe 15-10). It's in these methods (or methods called by these methods) where you will define the bulk of your application's functionality.
- Declare a constructor for your form class that instantiates each of the form's controls and configures their initial state (size, color, position, content, and so on). The constructor should also wire up the appropriate event handler methods of your class to the events of each control.

- Declare a Shared method named Main—usually as a member of your application’s main form class. This method is the entry point for your application, and it can have the same signatures as those mentioned in recipe 1-1. In the Main method, call Application.EnableVisualStyles to allow support for themes (supported by Windows XP, Windows Server 2003, and Windows Vista), create an instance of your application’s main form, and pass it as an argument to the Shared Application.Run method. The Run method makes your main form visible and starts a standard Windows message loop on the current thread, which passes the user input (key presses, mouse clicks, and so on) to your application form as events.

The Code

The Recipe01_02 class shown in the following code listing is a simple Windows Forms application that demonstrates the techniques just listed. When run, it prompts a user to enter a name and then displays a message box welcoming the user to “Visual Basic 2008 Recipes.”

```
Imports System
Imports System.Windows.Forms

Namespace Apress.VisualBasicRecipes.Chapter01

    Public Class Recipe01_02
        Inherits Form

        ' Private members to hold references to the form's controls.
        Private Label1 As Label
        Private TextBox1 As TextBox
        Private Button1 As Button

        ' Constructor used to create an instance of the form and configure
        ' the form's controls.
        Public Sub New()
            ' Instantiate the controls used on the form.
            Me.Label1 = New Label
            Me.TextBox1 = New TextBox
            Me.Button1 = New Button

            ' Suspend the layout logic of the form while we configure and
            ' position the controls.
            Me.SuspendLayout()

            ' Configure Label1, which displays the user prompt.
            Me.Label1.Location = New System.Drawing.Size(16, 36)
            Me.Label1.Name = "Label1"
            Me.Label1.Size = New System.Drawing.Size(155, 16)
            Me.Label1.TabIndex = 0
            Me.Label1.Text = "Please enter your name:"

            ' Configure TextBox1, which accepts the user input.
            Me.TextBox1.Location = New System.Drawing.Point(172, 32)
            Me.TextBox1.Name = "TextBox1"
            Me.TextBox1.TabIndex = 1
            Me.TextBox1.Text = ""
```

```

        ' Configure Button1, which the user clicks to enter a name.
        Me.Button1.Location = New System.Drawing.Point(109, 80)
        Me.Button1.Name = "Button1"
        Me.Button1.TabIndex = 2
        Me.Button1.Text = "Enter"
        AddHandler Button1.Click, AddressOf Button1_Click

        ' Configure WelcomeForm, and add controls.
        Me.ClientSize = New System.Drawing.Size(292, 126)
        Me.Controls.Add(Me.Button1)
        Me.Controls.Add(Me.TextBox1)
        Me.Controls.Add(Me.Label1)
        Me.Name = "Form1"
        Me.Text = "Visual Basic 2008 Recipes"

        ' Resume the layout logic of the form now that all controls are
        ' configured.
        Me.ResumeLayout(False)

    End Sub

    Private Sub Button1_Click(ByVal sender As Object, ➡
ByVal e As System.EventArgs)

        ' Write debug message to the console.
        System.Console.WriteLine("User entered: " + TextBox1.Text)

        ' Display welcome as a message box.
        MessageBox.Show("Welcome to Visual Basic 2008 Recipes, " + ➡
TextBox1.Text, "Visual Basic 2008 Recipes")

    End Sub

    ' Application entry point, creates an instance of the form, and begins
    ' running a standard message loop on the current thread. The message
    ' loop feeds the application with input from the user as events.
    Public Shared Sub Main()
        Application.EnableVisualStyles()
        Application.Run(New Recipe01_02())
    End Sub

End Class

End Namespace

```

Usage

To build the `Recipe01_02` class into an application, use this command:

```
vbc /target:winexe Recipe01-02.vb
```

The `/target:winexe` switch tells the compiler that you are building a Windows-based application. As a result, the compiler builds the executable in such a way that no console is created when you run your application. If you use the `/target:exe` switch instead of `/target:winexe` to build a Windows Forms application, your application will still work correctly, but you will have a console window

visible while the application is running. Although this is undesirable for production-quality software, the console window is useful if you want to write debug and logging information while you're developing and testing your Windows Forms application. You can write to this console using the `Write` and `WriteLine` methods of the `System.Console` class.

Figure 1-1 shows the `WelcomeForm.exe` application greeting a user named John Doe. This version of the application is built using the `/target:exe` compiler switch, resulting in the visible console window in which you can see the output from the `Console.WriteLine` statement in the `button1_Click` event handler.

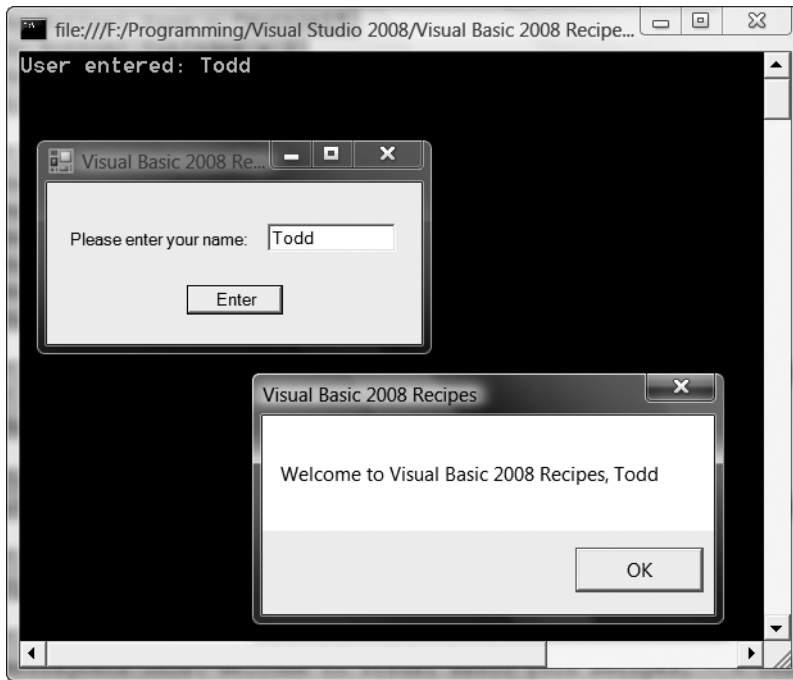


Figure 1-1. *A simple Windows Forms application*

1-3. Create and Use a Code Module from the Command Line

Problem

You need to do one or more of the following:

- Improve your application's performance and memory efficiency by ensuring the runtime loads rarely used types only when they are required.
- Compile types written in VB .NET to a form you can build into assemblies being developed in other .NET languages.
- Use types developed in another language and build them into your VB .NET assemblies.

Solution

Build your VB .NET source code into a module by using the command-line compiler and specifying the `/target:module` compiler switch. To incorporate existing modules into your assembly, use the `/addmodule` compiler switch.

How It Works

Modules are the building blocks of .NET assemblies and should not be confused with the `Module` object type block. Modules consist of a single file that contains the following:

- Microsoft Intermediate Language (MSIL) code created from your source code during compilation
- Metadata describing the types contained in the module
- Resources, such as icons and string tables, used by the types in the module

Assemblies consist of one or more modules and an assembly manifest. An *assembly manifest* is metadata that contains important information (such as the name, version, culture, and so on) regarding the assembly. If the assembly contains a single module, the module and assembly manifest are usually built into a single file for convenience. If more than one module exists, the assembly represents a logical grouping of more than one file that you must deploy as a complete unit. In these situations, the assembly manifest is either contained in a separate file or built into one of the modules. Visual Studio includes the MSIL Disassembler tool (`Ildasm.exe`), which lets you view the raw MSIL code for any assembly. You can use this tool to view an assembly manifest.

By building an assembly from multiple modules, you complicate the management and deployment of the assembly, but under some circumstances, modules offer significant benefits:

- The runtime will load a module only when the types defined in the module are required. Therefore, where you have a set of types that your application uses rarely, you can partition them into a separate module that the runtime will load only if necessary. This can improve performance, especially if your application is loaded across a network, and minimize the use of memory.
- The ability to use many different languages to write applications that run on the common language runtime (CLR) is a great strength of the .NET Framework. However, the VB .NET compiler can't compile your Microsoft C# or COBOL .NET code for inclusion in your assembly. To use code written in another language, you can compile it into a separate assembly and reference it. But if you want it to be an integral part of your assembly, you must build it into a module. Similarly, if you want to allow others to include your code as an integral part of their assemblies, you must compile your code as modules. When you use modules, because the code becomes part of the same assembly, members marked as `Friend` or `Protected Friend` are accessible, whereas they would not be if the code had been accessed from an external assembly.

Usage

To compile a source file named `ConsoleUtils.vb` (see recipe 1-1 for the contents) into a module, use the command `vbcs /target:module ConsoleUtils.vb`. The result is the creation of a file named `ConsoleUtils.netmodule`. The `.netmodule` extension is the default extension for modules, and the file name is the same as the name of the VB .NET source file.

You can also build modules from multiple source files, which results in a single file containing the MSIL and metadata (the assembly manifest) for all types contained in all of the source files. The command `vbcs /target:module ConsoleUtils.vb WindowsUtils.vb` compiles two source files named `ConsoleUtils.vb` and `WindowsUtils.vb` to create the module named `ConsoleUtils.netmodule`. The module is named after the first source file listed unless you override the name with the `/out` compiler switch. For example, the command `vbcs /target:module /out:Utilities.netmodule ConsoleUtils.vb WindowsUtils.vb` creates a module named `Utilities.netmodule`.

To build an assembly consisting of multiple modules, you must use the `/addmodule` compiler switch. To build an executable named `MyFirstApp.exe` from two modules named `WindowsUtils.netmodule` and `ConsoleUtils.netmodule` and two source files named `SourceOne.vb` and `SourceTwo.vb`, use the

```
command vbc /out:MyFirstApp.exe /target:exe /addmodule:WindowsUtils.netmodule,ConsoleUtils.netmodule SourceOne.vb SourceTwo.vb.
```

This command will result in an assembly that is composed of the following components:

- MyFirstApp.exe, which contains the assembly manifest as well as the MSIL for the types declared in the SourceOne.vb and SourceTwo.vb source files
- ConsoleUtils.netmodule and WindowsUtils.netmodule, which are now integral components of the multifile assembly but are unchanged by this compilation process

1-4. Create and Use a Code Library from the Command Line

Problem

You need to build a set of functionality into a reusable code library so that multiple applications can reference and reuse it.

Solution

Build your library using the command-line VB .NET compiler, and specify the `/target:library` compiler switch. To reference the library, use the `/reference` compiler switch when you build your application, and specify the names of the required libraries.

How It Works

Recipe 1-1 showed you how to build an application named MyFirstApp.exe from the two source files ConsoleUtils.vb and HelloWorld.vb. The ConsoleUtils.vb file contains the ConsoleUtils class, which provides methods to simplify interaction with the Windows console. If you were to extend the functionality of the ConsoleUtils class, you could add functionality useful to many applications. Instead of including the source code for ConsoleUtils in every application, you could build it into a library and deploy it independently, making the functionality accessible to many applications.

Usage

To build the ConsoleUtils.vb file into a library, use the command `vbc /target:library ConsoleUtils.vb`. This will produce a library file named ConsoleUtils.dll. To build a library from multiple source files, list the name of each file at the end of the command. You can also specify the name of the library using the `/out` compiler switch; otherwise, the library is named after the first source file listed. For example, to build a library named MyFirstLibrary.dll from two source files named ConsoleUtils.vb and WindowsUtils.vb, use the command `vbc /out:MyFirstLibrary.dll /target:library ConsoleUtils.vb WindowsUtils.vb`.

Before distributing your library, you might consider strong naming it so that no one can modify your assembly and pass it off as being the original. Strong naming your library also allows people to install it into the global assembly cache (GAC), which makes reuse much easier. (Recipe 1-12 describes how to strong name your assembly, and recipe 1-17 describes how to install a strong-named assembly into the GAC.) You might also consider signing your library with an Authenticode signature, which allows users to confirm you are the publisher of the assembly. (See recipe 1-15 for details on signing assemblies with Authenticode.)

To compile an assembly that relies on types declared within external libraries, you must tell the compiler which libraries are referenced using the `/reference` compiler switch. For example, to compile the HelloWorld.vb source file (from recipe 1-1) if the ConsoleUtils class is contained in the ConsoleUtils.dll library, use the command `vbc /reference:ConsoleUtils.dll HelloWorld.vb`. Remember these four points:

- If you reference more than one library, separate each library name with a comma or semicolon, but don't include any spaces. For example, use `/reference:ConsoleUtils.dll,WindowsUtils.dll`.
- If the libraries aren't in the same directory as the source code, use the `/libpath` switch on the compiler to specify the additional directories where the compiler should look for libraries. For example, use `/libpath:c:\CommonLibraries,c:\Dev\ThirdPartyLibs`.
- Note that additional directories can be relative to the source folder. Don't forget that at runtime, the generated assembly must be in the same folder as the application that needs it, except if you deploy it into the GAC.
- If the library you need to reference is a multifile assembly, reference the file that contains the assembly manifest. (For information about multifile assemblies, see recipe 1-3.)

1-5. Embed a Resource File in an Assembly

Problem

You need to create a string-based resource file and embed it in an assembly.

Solution

Use the Resource Generator (`resgen.exe`) to create a compiled resource file. You then use the `/resource` switch of the compiler to embed the file in the assembly.

Note The Assembly Linker tool (`al.exe`) also provides functionality for working with and embedding resource files. Refer to the Assembly Linker information in the .NET Framework SDK documentation for details.

How It Works

If you need to store strings in an external file and have them accessible to your assembly, you can use a resource file. *Resources* are some form of data (a string or an image, for example) that is used by an application. A *resource file* is a repository of one or more resources that can be easily accessed.

If you need to store only strings, you can create a simple text file that contains one or more key/value pairs in the form of `key=value`. You cannot create image resources starting from a text file.

Once you have your text file, you compile it using the Resource Generator (`resgen.exe`). Using this utility, you can convert the text file into either of two types:

- An `.resx` file, which is an XML resource file. This file is fully documented and can be edited manually. It is also capable of supporting image resources, unlike the text file. Consult the .NET Framework SDK documentation for more details on the `.resx` format.
- A `.resource` file, which is a compiled binary file and is required if you are embedding the file into your assembly using the command-line compiler. You embed the `.resource` file into your assembly by using the `/resource` switch of the VB .NET compiler. The `.resource` file can be compiled from a `.txt` or `.resx` file.

You access the contents of the resource file by instantiating a `ResourceManager` object. The `GetString` method is used to retrieve the value for the specified string. If you have stored something other than a string such as an image in your resource file, use the `GetObject` method and cast the return value to the appropriate type.

The Code

This example borrows the code from recipe 1-2. The dialog box titles and message prompt have been removed from the code and are now contained within an external resource file. The new program uses the `ResourceManager` object to access the resources.

```
Imports System
Imports System.Windows.Forms
Imports System.Resources

Namespace Apress.VisualBasicRecipes.Chapter01
    Public Class Recipe01_05
        Inherits Form

        ' Private members to hold references to the form's controls.
        Private label1 As Label
        Private textbox1 As TextBox
        Private button1 As Button
        Private resManager As New ResourceManager("MyStrings", ➡
System.Reflection.Assembly.GetExecutingAssembly())

        ' Constructor used to create an instance of the form and configure
        ' the form's controls.
        Public Sub New()
            ' Instantiate the controls used on the form.
            Me.label1 = New Label
            Me.textbox1 = New TextBox
            Me.button1 = New Button

            ' Suspend the layout logic of the form while we configure and
            ' position the controls.
            Me.SuspendLayout()

            ' Configure label1, which displays the user prompt.
            Me.label1.Location = New System.Drawing.Size(16, 36)
            Me.label1.Name = "label1"
            Me.label1.Size = New System.Drawing.Size(155, 16)
            Me.label1.TabIndex = 0
            Me.label1.Text = resManager.GetString("UserPrompt")

            ' Configure textbox1, which accepts the user input.
            Me.textbox1.Location = New System.Drawing.Point(172, 32)
            Me.textbox1.Name = "textbox1"
            Me.textbox1.TabIndex = 1
            Me.textbox1.Text = ""

            ' Configure button1, which the user clicks to enter a name.
            Me.button1.Location = New System.Drawing.Point(109, 80)
            Me.button1.Name = "button1"
            Me.button1.TabIndex = 2
            Me.button1.Text = resManager.GetString("ButtonCaption")
            AddHandler button1.Click, AddressOf button1_Click

            ' Configure WelcomeForm, and add controls.
            Me.ClientSize = New System.Drawing.Size(292, 126)
```

```

        Me.Controls.Add(Me.button1)
        Me.Controls.Add(Me.textbox1)
        Me.Controls.Add(Me.label1)
        Me.Name = "form1"
        Me.Text = resManager.GetString("FormTitle")

        ' Resume the layout logic of the form now that all controls are
        ' configured.
        Me.ResumeLayout(False)

    End Sub

    Private Sub button1_Click(ByVal sender As Object, ➡
        ByVal e As System.EventArgs)

        ' Write debug message to the console.
        System.Console.WriteLine("User entered: " + textbox1.Text)

        ' Display welcome as a message box.
        MessageBox.Show(resManager.GetString("Message") + textbox1.Text, ➡
        resManager.GetString("FormTitle"))

    End Sub

    ' Application entry point, creates an instance of the form, and begins
    ' running a standard message loop on the current thread. The message
    ' loop feeds the application with input from the user as events.
    Public Shared Sub Main()
        Application.EnableVisualStyles()
        Application.Run(New Recipe01_05())
    End Sub

End Class
End Namespace

```

Usage

First, you must create the MyStrings.txt file that contains your resource strings:

```

;String resource file for Recipe01-05
UserPrompt=Please enter your name:
FormTitle=Visual Basic 2008 Recipes
Message=Welcome to Visual Basic 2008 Recipes,
ButtonCaption=Enter

```

You compile this file into a resource file by using the command `resgen.exe MyStrings.txt Recipe01_05.MyStrings.resources`. To build the example and embed the resource file, use the command `vbc /resource:Recipe01_05.MyStrings.resources Recipe01-05.vb`.

Notes

Using resource files from Visual Studio is a little different from using resource files from the command line. For this example, the resource file must be in the XML format (.resx) and added directly to the project. Instead of initially creating the .resource file, you can use the command `resgen.exe MyStrings.txt MyStrings.resx` to generate the .resx file required by Visual Studio.

1-6. Build Projects from the Command Line Using MSBuild.exe

Problem

You need to compile one or more VB .NET files from the command line, and you need to have more precise control over the build process.

Solution

Create a project file, and use the MSBuild.exe utility that ships with Visual Studio 2008. The build project should reference each VB .NET file and compile them using the VB .NET compiler (vbc.exe) via the vbc task.

How It Works

MSBuild.exe is a utility that ships with Visual Studio. It is located in the directory specific to the target framework, such as C:\Windows\Microsoft.NET\Framework\v3.5\. This utility uses an XML project file to perform specified actions on specified files. If you build an application in Visual Studio, a file with the extension .vbproj is automatically generated. This is actually an XML project file used by MSBuild.exe to build your project.

Note For general information on working with XML files, please refer to Chapter 7.

The first step is creating a project file. As mentioned earlier, this is an XML file that contains key elements that MSBuild.exe interprets. The first element, which is required for any project file, is Project. This element must include the xmlns attribute set to <http://schemas.microsoft.com/developer/msbuild/2003>. The root Project element can contain any of the child elements listed in Table 1-1.

Table 1-1. *Common Child Elements*

Name	Description
Choose	Allows you to specify ItemGroup or PropertyGroup elements based on one or more condition.
Import	Imports an external project file.
ItemGroup	A group of user-defined Item elements. Each Item element represents some data to be reference elsewhere in the build project.
ProjectExtensions	Information that can be included in the build project but is ignored by MSBuild.exe.
PropertyGroup	A group of user-defined Property elements. Each Property element represents some property to be referenced elsewhere in the build project.
Target	Defines one or more Task elements. Each Task element performs some action as part of the build process.
UsingTask	Registers tasks to be made available to MSBuild.exe.

If your build project is going to reference files, your next step is to create an `ItemGroup` element with an `Item` element for each file. Item elements can be named anything, but it is best to use a name that represents what the file is. For example, if you had two VB .NET files, you might use `SourceFile`, which represents an `Item` element, as shown here:

```
<ItemGroup>
  <SourceFile Include="FileOne.vb" />
  <SourceFile Include="FileTwo.vb" />
</ItemGroup>
```

Using the same name, such as `SourceFile` used in the previous example, will group the files together. You can accomplish the same thing by putting the files on a single line and separating them with a semicolon like this:

```
<SourceFile Include="FileOne.vb;FileTwo.vb" />
```

Each `Item` element *must* contain the `Include` attribute, which is used to define the value of the element. When you need to reference a defined `Item` element, you just surround it with parentheses and precede it with the `@` symbol, as in `@(SourceFile)`.

Once you have defined files, you need to do something with them. You do this by creating a `Target` element and defining any appropriate predefined `Task` elements. By default, `MSBuild.exe` includes several tasks, some of which are listed in Table 1-2. These tasks are defined in `Microsoft.Build.Tasks.v3.5.dll` and are referenced by the `MSBuild.exe` utility by way of the `Microsoft.Common.Tasks` project file, which is included for any build by default.

Table 1-2. *Common MSBuild.exe Tasks*

Name	Description
Copy	Copies the specified files to the specified location
MakeDir	Creates the specified directory
RemoveDir	Removes the specified directory
SignFile	Uses the specified certificate to sign the specified file
Message	Writes the specified message to the build log
Exec	Executes the specified application using the specified parameters
Vbc	Compiles code using the VB .NET compiler (vbc.exe)
GenerateResource	Creates resource files similar to the <code>resgen.exe</code> utility discussed in recipe 1-5

One of the most common tasks that will be used is the `Vbc` task. This task actually wraps `vbc.exe`, making it possible to compile any VB .NET files. All the parameters available to `vbc.exe` are available as properties to the `Vbc` task, although some of the names have changed. Table 1-3 lists some of the most common properties and their matching `vbc.exe` parameters.

Table 1-3. *Common Vbc Task Properties*

Vbc Task Property	Vbc.exe Parameter	Description
KeyFile	/keyfile	Specifies the cryptographic key to be used (discussed in further detail in recipe 1-9)
KeyContainer	/keycontainer	Specifies the name of the cryptographic container where the cryptographic key can be found (discussed in further detail in recipe 1-9)
References	/reference	References additional assemblies to be compiled (discussed in further detail in recipe 1-4)
TargetType	/target	Defines the format of the output file (discussed in further detail in recipes 1-1, 1-2, and 1-3)
Resources	/resources	Embeds a resource (discussed in further detail in recipe 1-5)
OutputAssembly	/out	Defines the name of the output file (discussed in further detail in recipes 1-1 and 1-3)
MainEntryPoint	/main	Specifies the location of the Sub Main routine (discussed in further detail in recipe 1-1)
AddModules	/addmodule	Imports the specified modules (discussed in further detail in recipe 1-3)

Usage

If you wanted to create a project using the files from recipe 1-1, it would look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
  <ItemGroup>
    <SourceFile Include="ConsoleUtils.vb" />
    <SourceFile Include="HelloWorld.vb" />
  </ItemGroup>
  <Target Name="TestBuild" >
    <Vbc TargetType="exe"
      MainEntryPoint="Apress.VisualBasicRecipes.Chapter01.HelloWorld"
      OutputAssembly ="HelloWorld.exe"
      Sources="@ (SourceFile)" />
  </Target>
</Project>
```

Once you have created the project file, you use MSBuild.exe to build it. MSBuild.exe includes many parameters (such as /property, /logger, and /verbosity) that can be used to fine-tune the build process. For example, we will use the simplest form, which requires only the name of the project file:

```
Msbuild.exe HelloWorld.proj
```

Executing this command will create the HelloWorld.exe file and produce results similar to the following:

```
Microsoft (R) Build Engine Version 3.5.20706.1  
[Microsoft .NET Framework, Version 2.0.50727.1378]  
Copyright (C) Microsoft Corporation 2007. All rights reserved.
```

```
Build started 9/1/2007 9:01:22 PM.
```

```
Build succeeded.  
    0 Warning(s)  
    0 Error(s)
```

```
Time Elapsed 00:00:02.42
```

Note This recipe covers only the very basics of MSBuild.exe. If you view the build project file that is automatically created by Visual Studio (as mentioned earlier), you will notice how in-depth it is. For a complete reference to the MSBuild.exe utility, refer to the online documentation at <http://msdn2.microsoft.com/en-us/library/0k6kkbsd.aspx>.

1-7. Access Command-Line Arguments

Problem

You need to access the arguments that were specified on the command line when your application was executed.

Solution

Use a signature for your Main method that exposes the command-line arguments as a String array. Alternatively, access the command-line arguments from anywhere in your code using the Shared members of the System.Environment class.

How It Works

Declaring your application's Main method with one of the following signatures provides access to the command-line arguments as a String array:

```
Public Shared Sub Main(ByVal args As String())  
End Sub  
Public Shared Function Main(ByVal args As String()) As Integer  
End Sub
```

At runtime, the args argument will contain a string for each value entered on the command line after your application's name. The application's name is not included in the array of arguments.

If you need access to the command-line arguments at places in your code other than the Main method, you can process the command-line arguments in your Main method and store them for later access. However, this is not necessary since you can use the System.Environment class, which provides two Shared members that return information about the command line: CommandLine and GetCommandLineArgs. The CommandLine property returns a string containing the full command line

that launched the current process. Depending on the operating system on which the application is running, path information might precede the application name. Windows Server 2003, Windows Server 2008, Windows NT 4.0, Windows 2000, Windows XP, and Windows Vista don't include path information, whereas Windows 98 and Windows ME do. The `GetCommandLineArgs` method returns a `String` array containing the command-line arguments. This array can be processed in the same way as the `String` array passed to the `Main` method, as discussed at the start of this section. Unlike the array passed to the `Main` method, the first element in the array returned by the `GetCommandLineArgs` method is the file name of the application.

Note As an alternative, you can use the `My.Application.CommandLineArgs` method (which works identically to the `GetCommandLineArgs` method). We discuss the `My` namespace more thoroughly in Chapter 5.

The Code

To demonstrate the access of command-line arguments, the `Main` method in the following example steps through each of the command-line arguments passed to it and displays them to the console. The example then accesses the command line directly through the `Environment` class.

Imports System

Namespace Apress.VisualBasicRecipes.Chapter01

Public Class Recipe01_07

Public Shared Sub Main(ByVal args As String())

 ' Step through the command-line arguments

 For Each s As String In args

 Console.WriteLine(s)

 Next

 ' Alternatively, access the command-line arguments directly.

 Console.WriteLine(Environment.CommandLine)

 For Each s As String In Environment.GetCommandLineArgs()

 Console.WriteLine(s)

 Next

 ' Wait to continue

 Console.WriteLine(vbCrLf & "Main method complete. Press Enter.")

 Console.ReadLine()

End Sub

End Class

End Namespace

Usage

If you execute the `Recipe01-07` example using the following command:

```
Recipe01-07 "one \"two\"    three" four 'five    six'
```

the application will generate the following output on the console:

```
one "two"    three
four
'five
six'
recipe01-07  "one \"two\"    three" four 'five    six'
recipe01-07
one "two"    three
four
'five
six'
```

Main method complete. Press Enter.

Notice that the use of double quotes (") results in more than one word being treated as a single argument, although single quotes (') do not. Also, you can include double quotes in an argument by escaping them with the backslash character (\). Finally, notice that all spaces are stripped from the command line unless they are enclosed in double quotes.

1-8. Include Code Selectively at Build Time

Problem

You need to selectively include and exclude sections of source code from your compiled assembly.

Solution

Use the `#If`, `#ElseIf`, `#Else`, and `#End If` preprocessor directives to identify blocks of code that should be conditionally included in your compiled assembly. Use the `System.Diagnostics.ConditionalAttribute` attribute to define methods that should be called conditionally only. Control the inclusion of the conditional code using the `#Const` directive in your code, or use the `/define` switch when you run the VB .NET compiler from the command line.

How It Works

If you need your application to function differently depending on factors such as the platform or environment on which it runs, you can build runtime checks into the logic of your code that trigger the variations in operation. However, such an approach can bloat your code and affect performance, especially if many variations need to be supported or many locations exist where evaluations need to be made.

An alternative approach is to build multiple versions of your application to support the different target platforms and environments. Although this approach overcomes the problems of code bloat and performance degradation, it would be an untenable solution if you had to maintain different source code for each version, so VB .NET provides features that allow you to build customized versions of your application from a single code base.

The `#If`, `#ElseIf`, `#Else`, and `#End If` preprocessor directives allow you to identify blocks of code that the compiler should include or exclude in your assembly at compile time. This is accomplished by evaluating the value of specified symbols. Since this happens at compile time, it may result in multiple executables being distributed. Symbols can be any literal value. They also support the use of all standard comparison and logical operators or other symbols. The `#If. .#End If` construct evaluates `#If`

and `#ElseIf` clauses only until it finds one that evaluates to true, meaning that if you define multiple symbols (`winXP` and `win2000`, for example), the order of your clauses is important. The compiler includes only the code in the clause that evaluates to true. If no clause evaluates to true, the compiler includes the code in the `#Else` clause.

You can also use logical operators to base conditional compilation on more than one symbol. Use parentheses to group multiple expressions. Table 1-4 summarizes the supported operators.

Table 1-4. *Logical Operators Supported by the `#If` . . . `#End If` Directive*

Operator	Example	Description
NOT	<code>#If NOT winXP</code>	Inequality. Evaluates to true if the symbol <code>winXP</code> is not equal to <code>True</code> . Equivalent to <code>#If NOT winXP</code> .
AND	<code>#If winXP AND release</code>	Logical AND. Evaluates to true only if the symbols <code>winXP</code> and <code>release</code> are equal to <code>True</code> .
AndAlso	<code>#If winXP AndAlso release</code>	Logical AND. Works the same as the <code>AND</code> operator, except that the second expression (<code>release</code>) is not evaluated if the first expression (<code>winXP</code>) is <code>False</code> .
OR	<code>#If winXP OR release</code>	Logical OR. Evaluates to true if either of the symbols <code>winXP</code> or <code>release</code> is equal to <code>True</code> .
OrElse	<code>#If winXP OrElse release</code>	Logical OR. Works the same as the <code>OR</code> operator, except that the second expression (<code>release</code>) is not evaluated if the first expression (<code>winXP</code>) is <code>True</code> .
XOR	<code>#If winXP XOR release</code>	Logical XOR. Evaluates to true if only one of the symbols, <code>winXP</code> or <code>release</code> , is equal to <code>True</code> .

Caution You must be careful not to overuse conditional compilation directives and not to make your conditional expressions too complex; otherwise, your code can quickly become confusing and unmanageable—especially as your projects become larger.

To define a symbol, you can either include a `#Const` directive in your code or use the `/define` compiler switch. Symbols defined using `#Const` are active until the end of the file in which they are defined. Symbols defined using the `/define` compiler switch are active in all source files that are being compiled. All `#Const` directives must appear at the top of your source file before any code, including any `Imports` statements.

If you need to determine only whether a symbol has been defined, a more elegant alternative to the `#If` preprocessor directive is the attribute `System.Diagnostics.ConditionalAttribute`. If you apply `ConditionalAttribute` to a method, the compiler will ignore any calls to the method if the symbol specified by `ConditionalAttribute` is not defined, or set to `False`, at the calling point.

Using `ConditionalAttribute` centralizes your conditional compilation logic on the method declaration and means you can freely include calls to conditional methods without littering your code with `#If` directives. However, because the compiler literally removes calls to the conditional method from your code, your code can't have dependencies on return values from the conditional method. This means you can apply `ConditionalAttribute` only to subroutines.

The Code

In this example, the code assigns a different value to the local variable `platformName` based on whether the `winVista`, `winXP`, `win2000`, `winNT`, or `Win98` symbols are defined. The head of the code defines the `win2000` symbol. In addition, the `ConditionalAttribute` specifies that calls to the `DumpState` method should be included in an assembly only if the symbol `DEBUG` is defined during compilation. The `DEBUG` symbol is defined by default in debug builds.

```
#Const winXP = True

Imports System
Imports System.Diagnostics

Namespace APress.VisualBasicRecipes.Chapter01
    Public Class Recipe01_08

        ' Declare a string to contain the platform name
        Private Shared platformName As String
        <Conditional("DEBUG")> _
        Public Shared Sub DumpState()
            Console.WriteLine("Dump some state...")
        End Sub

        Public Shared Sub Main()

            #If winVista Then          ' Compiling for Windows Vista
                platformName = "Microsoft Windows Vista"
            #ElseIf winXP Then        ' Compiling for Windows XP
                platformName = "Microsoft Windows XP"
            #ElseIf win2000 Then      ' Compiling for Windows 2000
                platformName = "Microsoft Windows 2000"
            #ElseIf winNT Then        ' Compiling for Windows NT
                platformName = "Microsoft Windows NT"
            #ElseIf win98 Then        ' Compiling for Windows 98
                platformName = "Microsoft Windows 98"
            #Else                    ' Unknown platform specified
                platformName = "Unknown"
            #End If

            Console.WriteLine(platformName)

            ' Call the conditional DumpState method
            DumpState()

            ' Wait to continue...
            Console.WriteLine(vbCrLf & "Main method complete. Press Enter.")
            Console.Read()

        End Sub

    End Class
End Namespace
```

Usage

To build the example and define the symbol `winVista`, use the command `vbcs /define:winVista Recipe01-08.vb`. If you compile this sample without defining the `winVista` symbol, the `winXP` symbol

will be used since it was defined directly in the code. Otherwise, both `winVista` and `winXP` will be defined, but `Microsoft Windows Vista` will be the `platformName` value because of the order in which the symbols are checked.

Notes

You can apply multiple `ConditionalAttribute` instances to a method in order to produce logical OR behavior. Calls to the following version of the `DumpState` method will be compiled only if the `DEBUG` or `TEST` symbols are defined:

```
<Conditional("DEBUG"), Conditional("TEST")> _
Public Shared Sub DumpState()
    ...
End Sub
```

Achieving logical AND behavior is not as clean and involves the use of an intermediate conditional method, quickly leading to overly complex code that is hard to understand and maintain. You should be cautious with this approach, because you might end up with code in your assembly that is never called. The following is a quick example that requires the definition of both the `DEBUG` and `TEST` symbols for the `DumpState` functionality (contained in `DumpState2`) to be called:

```
<Conditional("DEBUG")> _
Public Shared Sub DumpState()
    DumpState2()
End Sub

<Conditional("TEST")> _
Public Shared Sub DumpState2()
    ...
End Sub
```

It's important to remember that you are not limited to Boolean values for your symbols. You can define a symbol with a string value, like this:

```
#Const OS = "Vista"
```

You could also do this using the command `vbc /define:OS=\"winVista\" Recipe01-08.vb`. You must escape quotation marks using the `\` character.

To use this new symbol, the preprocessor `#If...#End If` construct must be changed accordingly:

```
#If OS = "winVista" Then      ' Compiling for Windows Vista
    platformName = "Microsoft Windows Vista"
#ElseIf OS = "XP" Then       ' Compiling for Windows XP
    platformName = "Microsoft Windows XP"
#ElseIf OS = "2000" Then     ' Compiling for Windows 2000
    platformName = "Microsoft Windows 2000"
#ElseIf OS = "NT" Then       ' Compiling for Windows NT
    platformName = "Microsoft Windows NT"
#ElseIf OS = "98" Then       ' Compiling for Windows 98
    platformName = "Microsoft Windows 98"
#Else                         ' Unknown platform specified
    platformName = "Unknown"
#End If
```

1-9. Manipulate the Appearance of the Console

Problem

You want to control the visual appearance of the Windows console.

Solution

Use the Shared properties and methods of the `System.Console` class.

How It Works

The .NET Framework includes the `Console` class, which gives you control over the appearance and operation of the Windows console. Table 1-5 describes the properties and methods of this class that you can use to control the console's appearance.

Table 1-5. *Properties and Methods to Control the Appearance of the Console*

Member	Description
Properties	
<code>BackgroundColor</code>	Gets and sets the background color of the console using one of the values from the <code>System.ConsoleColor</code> enumeration. Only new text written to the console will appear in this color. To make the entire console this color, call the method <code>Clear</code> after you have configured the <code>BackgroundColor</code> property.
<code>BufferHeight</code>	Gets and sets the buffer height in terms of rows. Buffer refers to the amount of actual data that can be displayed within the console window.
<code>BufferWidth</code>	Gets and sets the buffer width in terms of columns. Buffer refers to the amount of actual data that can be displayed within the console window.
<code>CursorLeft</code>	Gets and sets the column position of the cursor within the buffer.
<code>CursorSize</code>	Gets and sets the height of the cursor as a percentage of a character cell.
<code>CursorTop</code>	Gets and sets the row position of the cursor within the buffer.
<code>CursorVisible</code>	Gets and sets whether the cursor is visible.
<code>ForegroundColor</code>	Gets and sets the text color of the console using one of the values from the <code>System.ConsoleColor</code> enumeration. Only new text written to the console will appear in this color. To make the entire console this color, call the method <code>Clear</code> after you have configured the <code>ForegroundColor</code> property.
<code>LargestWindowHeight</code>	Returns the largest possible number of rows based on the current font and screen resolution.
<code>LargestWindowWidth</code>	Returns the largest possible number of columns based on the current font and screen resolution.
<code>Title</code>	Gets and sets text shown in the title bar.

Table 1-5. *Properties and Methods to Control the Appearance of the Console (Continued)*

Member	Description
WindowHeight	Gets and sets the physical height of the console window in terms of character rows.
WindowWidth	Gets and sets the physical width of the console window in terms of character columns.
Methods	
Clear	Clears the console.
ResetColor	Sets the foreground and background colors to their default values as configured within Windows.
SetWindowSize	Sets the width and height in terms of columns and rows.

The Code

The following example demonstrates how to use the properties and methods of the Console class to dynamically change the appearance of the Windows console:

Imports System

```
Namespace Apress.VisualBasicRecipes.Chapter01
    Public Class Recipe01_09
```

```
        Public Shared Sub Main(ByVal args As String())

            ' Display the standard console.
            Console.Title = "Standard Console"
            Console.WriteLine("Press Enter to change the console's appearance.")
            Console.ReadLine()

            ' Change the console appearance and redisplay.
            Console.Title = "Colored Text"
            Console.ForegroundColor = ConsoleColor.Red
            Console.BackgroundColor = ConsoleColor.Green
            Console.WriteLine("Press Enter to change the console's appearance.")
            Console.ReadLine()

            ' Change the console appearance and redisplay.
            Console.Title = "Cleared / Colored Console"
            Console.ForegroundColor = ConsoleColor.Blue
            Console.BackgroundColor = ConsoleColor.Yellow
            Console.Clear()
            Console.WriteLine("Press Enter to change the console's appearance.")
            Console.ReadLine()

            ' Change the console appearance and redisplay.
            Console.Title = "Resized Console"
            Console.ResetColor()
            Console.Clear()
            Console.SetWindowSize(100, 50)
```

```

        Console.BufferHeight = 500
        Console.BufferWidth = 100
        Console.CursorLeft = 20
        Console.CursorSize = 50
        Console.CursorTop = 20
        Console.CursorVisible = False
        Console.WriteLine("Main method complete. Press Enter.")
        Console.ReadLine()

    End Sub

End Class
End Namespace

```

1-10. Access a Program Element That Has the Same Name As a Keyword

Problem

You need to access a member of a type, but the type or member name is the same as a VB .NET keyword.

Solution

Surround all instances of the identifier name in your code with brackets ([]).

How It Works

The .NET Framework allows you to use software components developed in other .NET languages from within your VB .NET applications. Each language has its own set of keywords (or reserved words) and imposes different restrictions on the names programmers can assign to program elements such as types, members, and variables. Therefore, it is possible that a programmer developing a component in another language will inadvertently use a VB .NET keyword as the name of a program element. Using brackets ([]) enables you to use a VB .NET keyword as an identifier and overcome these possible naming conflicts.

The Code

The following code fragment creates the new Operator (perhaps a telephone operator) class. A new instance of this class is created, and its Friend property is set to True—both Operator and Friend are VB .NET keywords:

```

Public Class [Operator]
    Public [Friend] As Boolean
End Class

' Instantiate an operator object
Dim operator1 As New [Operator]

' Set the operator's Friend property
operator1.[Friend] = True

```

1-11. Create and Manage Strong-Named Key Pairs

Problem

You need to create public and private keys (a key pair) so that you can assign strong names to your assemblies.

Solution

Use the Strong Name tool (sn.exe) to generate a key pair and store the keys in a file or cryptographic service provider (CSP) key container.

Note A CSP is an element of the Win32 CryptoAPI that provides services such as encryption, decryption, and digital signature generation. CSPs also provide key container facilities, which use strong encryption and operating system security to protect any cryptographic keys stored in the container. A detailed discussion of CSPs and CryptoAPI is beyond the scope of this book. All you need to know for this recipe is that you can store your cryptographic keys in a CSP key container and be relatively confident that it is secure as long as no one knows your Windows password. Refer to the CryptoAPI information in the platform SDK documentation for complete details.

How It Works

To generate a new key pair and store the keys in the file named `MyKeys.snk`, execute the command `sn -k MyKeys.snk`. (.snk is the usual extension given to files containing strong-named keys.) The generated file contains both your public and private keys. You can extract the public key using the command `sn -p MyKeys.snk MyPublicKeys.snk`, which will create `MyPublicKey.snk` containing only the public key. Once you have this file in hand, you can view the public key using the command `sn -tp MyPublicKeys.snk`, which will generate output similar to the (abbreviated) listing shown here:

```
Microsoft (R) .NET Framework Strong Name Utility Version 3.5.20706.1
Copyright (c) Microsoft Corporation. All rights reserved.
```

Public key is

```
0024000004800000940000000602000000240000525341310004000001000100c5810bb3c095d0
6de71d6cafba0b2088b45951ba76407d981d20bf1be825990619b6888d56146b9532981374df9a
fa1001b1336e262a09fa8c7d989cf4a0ad6bbe5684f9cd82cc38ba6d6707acaf13f058e22d6796
2dc72212bf797da89c08d8e65338c2972de659385472a603e00d3cc3c9f348b51d7c47a8611479
deb3f0ab
```

```
Public key token is 442a698bee81cc00
```

The public key token shown at the end of the listing is the last 8 bytes of a cryptographic hash code computed from the public key. Because the public key is so long, .NET uses the public key token for display purposes and as a compact mechanism for other assemblies to reference your public key. (Recipes 11-14 and 11-15 discuss cryptographic hash codes.)

As the name suggests, you don't need to keep the public key (or public key token) secret. When you strong name your assembly (discussed in recipe 1-12), the compiler uses your private key to generate a digital signature (an encrypted hash code) of the assembly's manifest. The compiler embeds the digital signature and your public key in the assembly so that any consumer of the assembly can verify the digital signature.

Keeping your private key secret is imperative. People with access to your private key can alter your assembly and create a new strong name—leaving your customers unaware they are using modified code. No mechanism exists to repudiate compromised strong-named keys. If your private key is compromised, you must generate new keys and distribute new versions of your assemblies that are strong named using the new keys. You must also notify your customers about the compromised keys and explain to them which versions of your public key to trust—in all, a very costly exercise in terms of both money and credibility. You can protect your private key in many ways; the approach you use will depend on several factors:

- The structure and size of your organization
- Your development and release process
- The software and hardware resources you have available
- The requirements of your customer base

Tip Commonly, a small group of trusted individuals (the *signing authority*) has responsibility for the security of your company's strong name signing keys and is responsible for signing all assemblies just prior to their final release. The ability to delay sign an assembly (discussed in recipe 1-14) facilitates this model and avoids the need to distribute private keys to all development team members.

One feature provided by the Strong Name tool to simplify the security of strong-named keys is the use of CSP key containers. Once you have generated a key pair to a file, you can install the keys into a key container and delete the file. For example, to store the key pair contained in the file `MyKeys.snk` to a CSP container named `StrongNameKeys`, use the command `sn -i MyKeys.snk StrongNameKeys`. You can install only one set of keys to a single container. (Recipe 1-12 explains how to use strong-named keys stored in a CSP key container.)

An important aspect of CSP key containers is that they include user-based containers and machine-based containers. Windows security ensures users can access only their own user-based key containers. However, any user of a machine can access a machine-based container.

By default, the Strong Name tool uses machine-based key containers, meaning that anyone who can log on to your machine and who knows the name of your key container can sign an assembly with your strong-named keys. To change the Strong Name tool to use user-based containers, use the command `sn -m n`, and to switch to machine-based stores, use the command `sn -m y`. The command `sn -m` will display whether the Strong Name tool is currently configured to use machine-based or user-based containers.

To delete the strong-named keys from the `StrongNameKeys` container (as well as delete the container), use the command `sn -d StrongNameKeys`.

1-12. Give an Assembly a Strong Name

Problem

You need to give an assembly a strong name for several reasons:

- So it has a unique identity, which allows people to assign specific permissions to the assembly when configuring code access security policy
- So it can't be modified and passed off as your original assembly
- So it can be installed in the GAC and shared across multiple applications

Solution

When you build your assembly using the command-line VB .NET compiler, use the `/keyfile` or `/keycontainer` compiler switch to specify the location of your strong-named key pair. Use assembly-level attributes to specify optional information such as the version number and culture for your assembly. The compiler will strong name your assembly as part of the compilation process.

Note If you are using Visual Studio, you can configure your assembly to be strong named by opening the project properties, selecting the Signing tab, and checking the Sign the Assembly box. You will need to specify the location of the file where your strong-named keys are stored—Visual Studio does not allow you to specify the name of a key container.

How It Works

To strong name an assembly using the VB .NET compiler, you need the following:

- A strong-named key pair contained either in a file or in a CSP key container. (Recipe 1-11 discusses how to create strong-named key pairs.)
- Compiler switches to specify the location where the compiler can obtain your strong-named key pair:
 - If your key pair is in a file, use the `/keyfile` compiler switch, and provide the name of the file where the keys are stored. For example, use `/keyfile:MyKeyFile.snk`.
 - If your key pair is in a CSP container, use the `/keycontainer` compiler switch, and provide the name of the CSP key container where the keys are stored. For example, use `/keycontainer:MyKeyContainer`.
- Optionally, specify the culture that your assembly supports by applying the attribute `System.Reflection.AssemblyCultureAttribute` to the assembly. (If you attempt to use this attribute with an executable assembly, you will receive a compile error because executable assemblies support only the neutral culture.)
- Optionally, specify the version of your assembly by applying the attribute `System.Reflection.AssemblyVersionAttribute` to the assembly.

The Code

The executable code that follows (from a file named `Recipe01-09.vb`) shows how to use the optional attributes (shown in bold) to specify the culture and the version for the assembly:

```
Imports System
Imports System.Reflection
```

```
<Assembly: AssemblyCulture("")>
<Assembly: AssemblyVersion("1.1.0.5")>
```

```
Namespace Apress.VisualBasicRecipes.Chapter01
    Public Class Recipe01_12
```

```
        Public Shared Sub main()
            Console.WriteLine("Welcome to Visual Basic 2008 Recipes")
```

```
        ' Wait to continue...
        Console.WriteLine(vbCrLf & "Main method complete. Press Enter.")
        Console.Read()
    End Sub

End Class
End Namespace
```

Usage

To create a strong-named assembly from the example code, create the strong-named keys and store them in a file named `MyKeyFile` using the command `sn -k MyKeyFile.snk`. Then install the keys into the CSP container named `MyKeys` using the command `sn -i MyKeyFile.snk MyKeys`. You can now compile the file into a strong-named assembly using the command `vb /keycontainer:MyKeys Recipe01-12.vb`. If you are not using a CSP container, you can specify the specific key file using the command `vb /keyfile:MyKeyFile.snk Recipe01-12.vb`.

Notes

If you use Visual Studio, you may not be able to include the optional `AssemblyVersion` attribute in your code. This is because the attribute may already exist for the assembly. By default, Visual Studio automatically creates a folder called `MyProject`. This folder stores multiple files, including `AssemblyInfo.vb`, which contains standard assembly attributes for the project. These can be manually edited or edited through the Assembly Information dialog box (see Figure 1-2), accessible from the Application tab of the project properties. Since the `AssemblyInfo.vb` file is an efficient way to store information specific to your assembly, it is actually good practice to create and use a similar file, even if you are not using Visual Studio to compile.

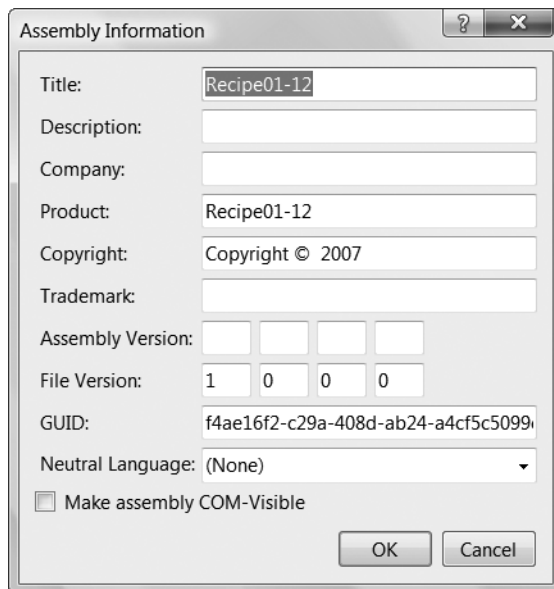


Figure 1-2. The Assembly Information dialog box

1-13. Verify That a Strong-Named Assembly Has Not Been Modified

Problem

You need to verify that a strong-named assembly has not been modified after it was built.

Solution

Use the Strong Name tool (sn.exe) to verify the assembly's strong name.

How It Works

Whenever the .NET runtime loads a strong-named assembly, the runtime extracts the encrypted hash code that's embedded in the assembly and decrypts it with the public key, which is also embedded in the assembly. The runtime then calculates the hash code of the assembly manifest and compares it to the decrypted hash code. This verification process will identify whether the assembly has changed after compilation.

If an executable assembly fails strong name verification, the runtime will display an error message or an error dialog box (depending on whether the application is a console or Windows application). If executing code tries to load an assembly that fails verification, the runtime will throw a `System.IO.FileLoadException` with the message "Strong name validation failed," which you should handle appropriately.

As well as generating and managing strong-named keys (discussed in recipe 1-11), the Strong Name tool allows you to verify strong-named assemblies. To verify that the strong-named assembly `Recipe01-12.exe` is unchanged, use the command `sn -vf Recipe01-12.exe`. The `-v` switch requests the Strong Name tool to verify the strong name of the specified assembly, and the `-f` switch forces strong name verification even if it has been previously disabled for the specified assembly. (You can disable strong name verification for specific assemblies using the `-Vr` switch, as in `sn -Vr Recipe01-12.exe`; see recipe 1-14 for details about why you would disable strong name verification.)

If the assembly passes strong name verification, you should see the following output:

```
Microsoft (R) .NET Framework Strong Name Utility Version 3.5.20706.1
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Assembly 'recipe01-12.exe' is valid
```

However, if the assembly has been modified, you will see this message:

```
Microsoft (R) .NET Framework Strong Name Utility Version 3.5.20706.1
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
recipe01-12.exe does not represent a strongly named assembly
```

1-14. Delay Sign an Assembly

Problem

You need to create a strong-named assembly, but you don't want to give all members of your development team access to the private key component of your strong-named key pair.

Solution

Extract and distribute the public key component of your strong-named key pair. Follow the instructions in recipe 1-12 that describe how to give your assembly a strong name. In addition, specify the `/delaysign` switch when you compile your assembly. Disable strong name verification for the assembly using the `-Vr` switch of the Strong Name tool (`sn.exe`).

Note If you are using Visual Studio, you can configure your strong-named assembly to be delay signed by opening the project properties, selecting the Signing tab, and checking the Delay Sign Only box. Doing so will prohibit your project from being run or debugged. You can get around this by skipping verification using the `-Vr` switch of the Strong Name tool.

How It Works

Assemblies that reference strong-named assemblies contain the public key token of the referenced assemblies. This means the referenced assembly must be strong named before it can be referenced. In a development environment in which assemblies are regularly rebuilt, this would require every developer and tester to have access to your strong-named key pair—a major security risk.

Instead of distributing the private key component of your strong-named key pair to all members of the development team, the .NET Framework provides a mechanism named *delay signing* with which you can partially strong name an assembly. The partially strong-named assembly contains the public key and the public key token (required by referencing assemblies) but contains only a placeholder for the signature that would normally be generated using the private key.

After development is complete, the signing authority (who has responsibility for the security and use of your strong-named key pair) re-signs the delay-signed assembly to complete its strong name. The signature is calculated using the private key and embedded in the assembly, making the assembly ready for distribution.

To delay sign an assembly, you need access only to the public key component of your strong-named key pair. No security risk is associated with distributing the public key, and the signing authority should make the public key freely available to all developers. To extract the public key component from a strong-named key file named `MyKeyFile.snk` and write it to a file named `MyPublicKey.snk`, use the command `sn -p MyKeyFile.snk MyPublicKey.snk`. If you store your strong-named key pair in a CSP key container named `MyKeys`, extract the public key to a file named `MyPublicKey.snk` using the command `sn -pc MyKeys MyPublicKey.snk`.

Once you have a key file containing the public key, you build the delay-signed assembly using the command-line VB .NET compiler by specifying the `/delaysign` compiler switch. For example, to build a delay-signed assembly using the `MyPublicKey.snk` public key from a source file named `Recipe01-14.vb`, use this command:

```
vbc /delaysign /keyfile:MyPublicKey.snk Recipe01-14.vb
```

When the runtime tries to load a delay-signed assembly, it will identify the assembly as strong named and will attempt to verify the assembly, as discussed in recipe 1-13. Because it doesn't have

a digital signature, you must configure the runtime on the local machine to stop verifying the assembly's strong name using the command `sn -Vr Recipe01-14.exe`. Note that you need to do so on every machine on which you want to run your application.

Tip When using delay-signed assemblies, it's often useful to be able to compare different builds of the same assembly to ensure they differ only by their signatures. This is possible only if a delay-signed assembly has been re-signed using the `-R` switch of the Strong Name tool. To compare the two assemblies, use the command `sn -D assembly1 assembly2`.

Once development is complete, you need to re-sign the assembly to complete the assembly's strong name. The Strong Name tool allows you to do this without changing your source code or recompiling the assembly; however, you must have access to the private key component of the strong-named key pair. To re-sign an assembly named `Recipe01-14.exe` with a key pair contained in the file `MyKeys.snk`, use the command `sn -R Recipe01-14.exe MyKeys.snk`. If the keys are stored in a CSP key container named `MyKeys`, use the command `sn -Rc Recipe01-14.exe MyKeys`.

Once you have re-signed the assembly, you should turn strong name verification for that assembly back on using the `-Vu` switch of the Strong Name tool, as in `sn -Vu Recipe01-14.exe`. To enable verification for all assemblies for which you have disabled strong name verification, use the command `sn -Vx`. You can list the assemblies for which verification is disabled using the command `sn -Vl`.

1-15. Sign an Assembly with an Authenticode Digital Signature

Problem

You need to sign an assembly with Authenticode so that users of the assembly can be certain you are its publisher and the assembly is unchanged after signing.

Solution

Use the Sign Tool (`signtool.exe`) to sign the assembly with your software publisher certificate (SPC).

How It Works

Strong names provide a unique identity for an assembly as well as proof of the assembly's integrity, but they provide no proof as to the publisher of the assembly. The .NET Framework allows you to use Authenticode technology to sign your assemblies. This enables consumers of your assemblies to confirm that you are the publisher, as well as confirm the integrity of the assembly. Authenticode signatures also act as evidence for the signed assembly, which people can use when configuring code access security policy.

To sign your assembly with an Authenticode signature, you need an SPC issued by a recognized *certificate authority* (CA). A CA is a company entrusted to issue SPCs (along with many other types of certificates) for use by individuals or companies. Before issuing a certificate, the CA is responsible for confirming that the requesters are who they claim to be and also for making sure the requesters sign contracts to ensure they don't misuse the certificates that the CA issues them.

To obtain an SPC, you should view the Microsoft Root Certificate Program Members list at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/rootcertprog.asp>. Here you will find a list of CAs, many of whom can issue you an SPC. For testing purposes, you can create a test SPC using the process described in recipe 1-16. However, you can't

distribute your software signed with this test certificate. Because a test SPC isn't issued by a trusted CA, most responsible users won't trust assemblies signed with it.

Once you have an SPC, you use the Sign Tool to Authenticode sign your assembly. The Sign Tool creates a digital signature of the assembly using the private key component of your SPC and embeds the signature and the public part of your SPC in your assembly (including your public key). When verifying your assembly, the consumer decrypts the encrypted hash code using your public key, recalculates the hash of the assembly, and compares the two hash codes to ensure they are the same. As long as the two hash codes match, the consumer can be certain that you signed the assembly and that it has not changed since you signed it.

Usage

The Sign Tool provides a graphical wizard that walks you through the steps to Authenticode sign your assembly. To sign an assembly named `MyAssembly.exe`, run this command:

```
signtool signwizard MyAssembly.exe
```

Click Next on the introduction screen, and you will see the File Selection screen, where you must enter the name of the assembly to Authenticode sign (see Figure 1-3). Because you specified the assembly name on the command line, it is already filled in. If you are signing a multifile assembly, specify the name of the file that contains the assembly manifest. If you intend to both strong name and Authenticode sign your assembly, you must strong name the assembly first. (See recipe 1-12 for details on strong naming assemblies.)



Figure 1-3. The Sign Tool's File Selection screen

Clicking Next takes you to the Signing Options screen (see Figure 1-4). If your SPC is in a certificate store, select the Typical radio button. If your SPC is in a file, select the Custom radio button. Then click Next.



Figure 1-4. *The Sign Tool's Signing Options screen*

Assuming you want to use a file-based certificate (like the test certificate created in recipe 1-16), click the Select from File button on the Signature Certificate screen (see Figure 1-5), select the file containing your SPC certificate, and then click Next.



Figure 1-5. *The Sign Tool's Signature Certificate screen*

The Private Key screen allows you to identify the location of your private keys, which will either be in a file or be in a CSP key container, depending on where you created and stored them (see Figure 1-6). The example assumes they are in a file named PrivateKeys.pvk.

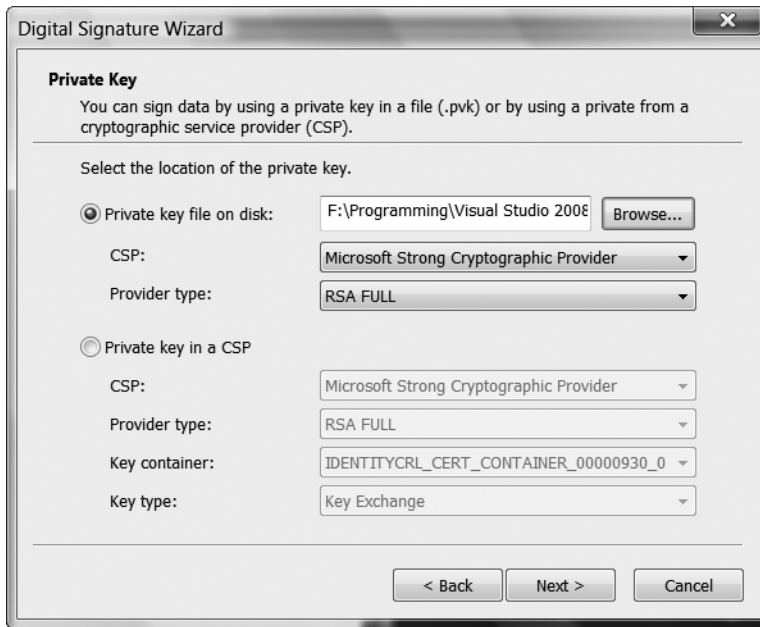


Figure 1-6. *The Sign Tool's Private Key screen*

When you click Next, if you selected to use a file, you will be prompted (see Figure 1-7) to enter a password to access the file (if required).



Figure 1-7. *Prompt for password to private key*

You can then select whether to use the sha1 or md5 hash algorithm (see Figure 1-8). The default is sha1, which is suitable for most purposes. On the Hash Algorithm screen, pick an algorithm, and then click Next.



Figure 1-8. *The Sign Tool's Hash Algorithm screen*

Click Next to leave the default values on the Additional Certificates screen, the Data Description screen, and the Timestamping screen. This will bring you to the final screen (see Figure 1-9), which shows you all the previous choices you made. If everything is accurate, click Finish. If you are using a file-based private key that is password protected, you will once again be prompted to enter the password, after which the Sign Tool will Authenticode sign your assembly.



Figure 1-9. *The Sign Tool's completion screen*

Note The Sign Tool uses capicom.dll version 2.1.0.1. If an error occurs when you run signtool.exe that indicates capicom.dll is not accessible or not registered, change to the directory where capicom.dll is located (which is C:\Program Files\Common Files\Microsoft Shared\CAPICOM by default), and run the command `regsvr32 capicom.dll`.

1-16. Create and Trust a Test Software Publisher Certificate

Problem

You need to create an SPC to allow you to test the Authenticode signing of an assembly.

Solution

Use the Certificate Creation tool (makecert.exe) to create a test X.509 certificate, and use the Software Publisher Certificate Test tool (cert2spc.exe) to generate an SPC from this X.509 certificate. Trust the root test certificate using the Set Registry tool (setreg.exe).

How It Works

To create a test SPC for a software publisher named Todd Herman, create an X.509 certificate using the Certificate Creation tool. The command `makecert -n "CN=Todd Herman" -sk MyKeys TestCertificate.cer` creates a file named TestCertificate.cer containing an X.509 certificate and stores the associated private key in a CSP key container named MyKeys (which is automatically created if it does not exist). Alternatively, you can write the private key to a file by substituting the `-sk` switch with `-sv`. For example, to write the private key to a file named PrivateKeys.pvk, use the command `makecert -n "CN=Todd Herman" -sv PrivateKey.pvk TestCertificate.cer`. If you write your private key to a file, the Certificate Creation tool will prompt you to provide a password with which to protect the private key file (see Figure 1-10).



Figure 1-10. The Certificate Creation tool requests a password when creating file-based private keys.

The Certificate Creation tool supports many arguments, and Table 1-6 lists some of the more useful ones. You should consult the .NET Framework SDK documentation for full coverage of the Certificate Creation tool.

Table 1-6. *Commonly Used Switches of the Certificate Creation Tool*

Switch	Description
-e	Specifies the date when the certificate becomes invalid.
-m	Specifies the duration—in months—that the certificate remains valid.
-n	Specifies an X.500 name to associate with the certificate. This is the name of the software publisher that people will see when they view details of the SPC you create.
-sk	Specifies the name of the CSP key store in which to store the private key.
-ss	Specifies the name of the certificate store where the Certificate Creation tool should store the generated X.509 certificate.
-sv	Specifies the name of the file in which to store the private key.

Once you have created your X.509 certificate with the Certificate Creation tool, you need to convert it to an SPC with the Software Publisher Certificate Test tool (`cert2spc.exe`). To convert the certificate `TestCertificate.cer` to an SPC, use the command `cert2spc TestCertificate.cer TestCertificate.spc`. The Software Publisher Certificate Test tool doesn't offer any optional switches.

The final step before you can use your test SPC is to trust the root test CA, which is the default issuer of the test certificate. The Set Registry tool (`setreg.exe`) makes this a simple task with the command `setreg 1 true`. You can now Authenticode sign assemblies with your test SPC using the process described in recipe 1-15. When you have finished using your test SPC, you must remove trust of the root test CA using the command `setreg 1 false`.

1-17. Manage the Global Assembly Cache

Problem

You need to add or remove assemblies from the GAC.

Solution

Use the Global Assembly Cache tool (`gacutil.exe`) from the command line to view the contents of the GAC as well as to add and remove assemblies.

How It Works

Before you can install an assembly in the GAC, the assembly must have a strong name. (See recipe 1-12 for details on how to strong name your assemblies.) To install an assembly named `SomeAssembly.dll` into the GAC, use the command `gacutil /i SomeAssembly.dll`. You can install different versions of the same assembly in the GAC to meet the versioning requirements of different applications.

To uninstall the `SomeAssembly.dll` assembly from the GAC, use the command `gacutil /u SomeAssembly`. Notice that you don't use the `.dll` extension to refer to the assembly once it's installed in the GAC. This will uninstall all assemblies with the specified name. To uninstall a particular version, specify the version along with the assembly name; for example, use `gacutil /u SomeAssembly,Version=1.0.0.5`.

To view the assemblies installed in the GAC, use the command `gacutil /l`. This will produce a long list of all the assemblies installed in the GAC, as well as a list of assemblies that have been precompiled to binary form and installed in the native image (ngen) cache. To avoid searching through this

list to determine whether a particular assembly is installed in the GAC, use the command `gacutil /l SomeAssembly`.

Note The .NET Framework uses the GAC only at runtime; the VB .NET compiler won't look in the GAC to resolve any external references that your assembly references. During development, the VB .NET compiler must be able to access a local copy of any referenced shared assemblies. You can either copy the shared assembly to the same directory as your source code or use the `/libpath` switch of the VB .NET compiler to specify the directory where the compiler can find the required assemblies.

1-18. Make Your Assembly More Difficult to Decompile

Problem

You want to make sure that people cannot decompile your .NET assemblies.

Solution

The *only* way to ensure that your assembly cannot be decompiled is by not making it directly accessible. This can be accomplished using a server-based solution. If you must distribute assemblies, you have *no* way to stop people from decompiling them. The best you can do is use obfuscation and components compiled to native code to make your assemblies more difficult to decompile.

How It Works

Because .NET assemblies consist of a standardized, platform-independent set of instruction codes and metadata that describes the types contained in the assembly, they are relatively easy to decompile. This allows decompilers to generate source code that is close to your original code with ease, which can be problematic if your code contains proprietary information or algorithms that you want to keep secret.

The only way to ensure people can't decompile your assemblies is to prevent them from getting your assemblies in the first place. Where possible, implement server-based solutions such as Microsoft ASP.NET applications and web services. With the security correctly configured on your server, no one will be able to access your assemblies, and therefore they won't be able to decompile them.

When building a server solution is not appropriate, you have the following two options:

- Use an obfuscator to make it difficult to understand your code once it is decompiled. Some versions of Visual Studio include the Community Edition of an obfuscator named Dotfuscator. Obfuscators use a variety of techniques to make your assembly difficult to decompile; principal among these techniques are renaming `Private` methods and fields in such a way that it's difficult to read and understand the purpose of your code, as well as inserting control flow statements to make the logic of your application difficult to follow.
- Build the parts of your application that you want to keep secret in native DLLs or COM objects, and then call them from your managed application using `P/Invoke` or `COM Interop`. (See Chapter 14 for recipes that show you how to call unmanaged code.)

Neither approach will stop a skilled and determined person from reverse engineering your code, but both approaches will make the job significantly more difficult and deter most casual observers.

Note The risks of application decompilation aren't specific to VB .NET or .NET in general. Determined people can reverse engineer any software if they have the time and the skill.

1-19. Use Implicitly Typed Variables

Problem

You need to create a strongly typed variable without explicitly declaring its type in an effort to save some development time or support LINQ, which is discussed in more detail in Chapter 6.

Solution

Ensure `Option Infer` is `On`, and then create a variable and assign it a value without using `As` and specifying a type.

How It Works

VB .NET 9.0 allows you to create strongly typed variables without explicitly setting their data types. You could do this in previous versions of VB .NET, if `Option Strict` were set to `Off`, but the variable was always typed as an `Object`. In this case, its type is automatically inferred based on its value.

To use this new functionality, `Option Infer` must be set to `On`. You can specify this setting in the Project Settings dialog box or by adding `Option Infer On` to the top of your code. If you create a new project in Visual Studio 2008, the project settings will have `Option Infer` set to `On` by default. Any projects that you migrate from previous Visual Studio versions will have `Option Infer` set to `Off`. If you are compiling your code using the VB compiler (`vbc`), you can use the `/optioninfer` option.

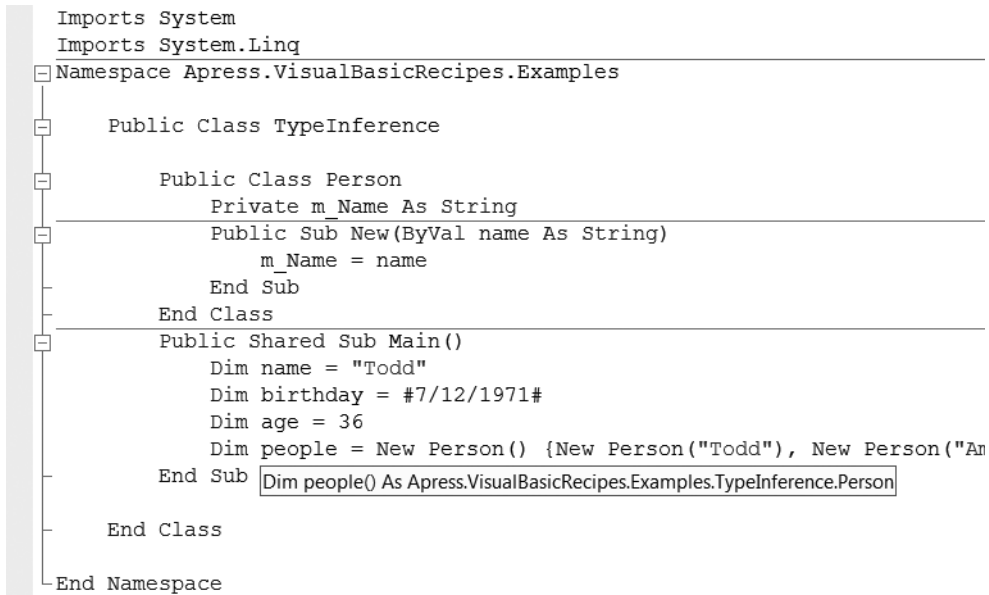
The following example demonstrates how to use type inference or implicit typing:

```
Dim name = "Todd"
Dim birthday = #7/12/1971#
Dim age = 36
Dim people = New Person() {New Person("Todd"), New Person("Amy"),
New Person("Alaina"), New Person("Aidan")}
```

If you hover your cursor over any of the variables in the preceding example in the Visual Studio IDE, you will see a tool tip that shows that they are actually being strongly typed. `name` is *inferred* as a `String`, `birthday` is a `Date`, `age` is an `Integer`, and, as shown in Figure 1-11, `people` is an array of `Person` objects.

When your code is compiled to Microsoft Intermediate Language (MSIL), all variables are strongly typed. (See recipes 1-3 and 2-6 for more information about MSIL.) If you looked at this compiled MSIL code using the MSIL Disassembler tool (`Ildasm.exe`), you would see that it has explicitly and correctly typed each variable. The following output was taken from the `Ildasm.exe` results for the sample code shown previously.

```
.locals init ([0] int32 age,
[1] valuetype [mscorlib]System.DateTime birthday,
[2] string name,
[3] class Apress.VisualBasicRecipes.Examples.TypeInference/Person[] people,
[4] class Apress.VisualBasicRecipes.Examples.TypeInference/Person[] VB$t_array$S0)
```



```
Imports System
Imports System.Linq
Namespace Apress.VisualBasicRecipes.Examples
    Public Class TypeInference
        Public Class Person
            Private m_Name As String
            Public Sub New(ByVal name As String)
                m_Name = name
            End Sub
        End Class
        Public Shared Sub Main()
            Dim name = "Todd"
            Dim birthday = #7/12/1971#
            Dim age = 36
            Dim people = New Person() {New Person("Todd"), New Person("An...")}
        End Sub
    End Class
End Namespace
```

Figure 1-11. A tool tip showing inferred type

Implicitly typing variables is an important part of creating and using LINQ queries, which are discussed in further detail in Chapters 6, 7, and 8. It is also a required component of *anonymous types*, which are discussed in recipe 1-21.

1-20. Use Object Initializers

Problem

You need to initialize the properties of a class when it is first instantiated, without relying on the class constructor or default values in an effort to save some development time or support LINQ, which is discussed in more detail in Chapter 6.

Solution

Instantiate a new class instance, and initialize any writable public fields or properties using the `With` keyword.

How It Works

VB .NET 9.0 includes the ability to initialize the writable public fields or properties of a class when it is first instantiated. When you use object initializers, the default constructor of the class is called automatically. This means any class you want to use object initializers for *must* have a default constructor. Any properties or fields that you do not initialize retain their default values.

Object initialization is made possible by using the `With` keyword. `With` is not new to VB .NET but was not previously usable in this manner. Here is a simple example of a class:

```

Public Class Person
    Private m_FirstName As String
    Private m_LastName As String

    Public Sub New()
        m_FirstName = String.Empty
        m_LastName = String.Empty
    End Sub

    Public Property FirstName() As String
        Get
            Return m_FirstName
        End Get
        Set(ByVal value As String)
            m_FirstName = value
        End Set
    End Property

    Public Property LastName() As String
        Get
            Return m_LastName
        End Get
        Set(ByVal value As String)
            m_LastName = value
        End Set
    End Property
End Class

```

In previous versions of VB .NET, you would instantiate and set property values like this:

```

Dim todd = New Person

With todd
    .FirstName = "Todd"
    .LastName = "Herman"
End With

```

The other option, if you had access to modify the class, is to use constructors to pass the property values. However, this method can become cumbersome quickly if you have a class with many properties. You further complicate things if you use an array, like this:

```

Dim people As Person() = New Person(2) {New Person, New Person, New Person}

With people(0)
    .FirstName = "Todd"
    .LastName = "Herman"
End With

With people(1)
    .FirstName = "Alaina"
    .LastName = "Herman"
End With

```

```
With people(2)
    .FirstName = "Aidan"
    .LastName = "Herman"
End With
```

Object initializers simplify this by allowing you to specify values during instantiation, like this:

```
Dim todd = New Person With {.FirstName = "Todd", .LastName = "Herman"}
```

or like this:

```
Dim people = New Person() {
    {New Person With {.FirstName = "Todd", _
        .LastName = "Herman"}, _
    New Person With {.FirstName = "Amy", _
        .LastName = "Herman"}, _
    New Person With {.FirstName = "Alaina", _
        .LastName = "Herman"}, _
    New Person With {.FirstName = "Aidan", _
        .LastName = "Herman"}}}
```

Note Although it is not required, both of the preceding examples of object initialization use type inference (see recipe 1-19), rather than relying on explicit typing.

As the examples show, you use the `With` keyword followed by a comma-delimited list of fields or properties and their values. The objects being initialized and their values should be surrounded by curly braces (`{}`). As shown in Figure 1-12, the VB 9.0 IDE provides IntelliSense for all objects that can be initialized.

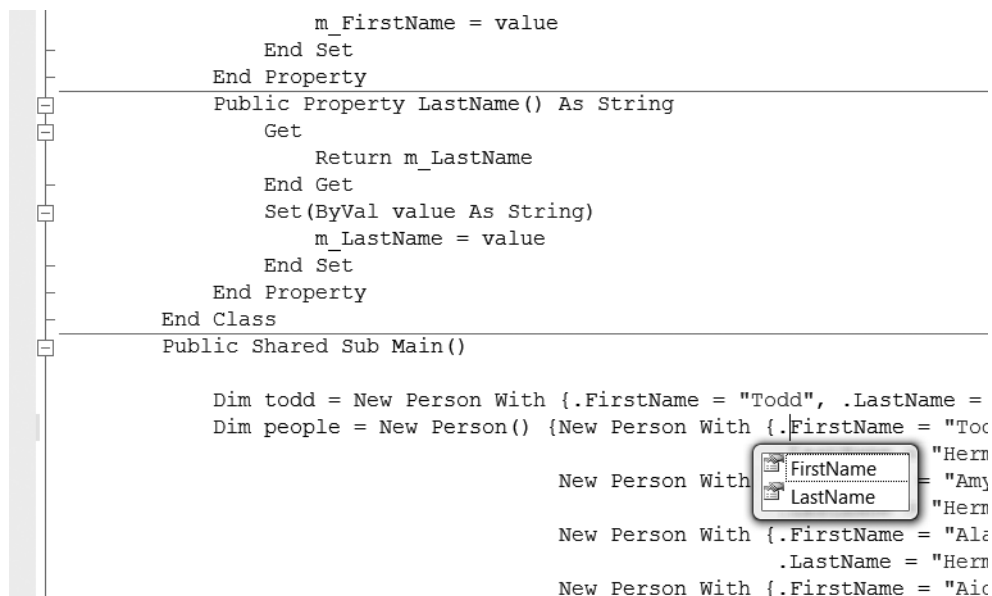


Figure 1-12. IntelliSense for object initializers

Object initializers are using anonymous types (see recipe 1-21) and making LINQ queries concise and efficient.

1-21. Use Anonymous Types

Problem

You need to use a simple type class that doesn't exist without actually creating it in an effort to save some development time or support LINQ, which is discussed in more detail in Chapter 6.

Solution

Instantiate a class as you would normally, using the `New` keyword, but do not specify a type. You must also use object initialization (see recipe 1-20) to specify at least one property.

How It Works

When you use the `New` keyword to instantiate an object, you typically specify the name of the type you want to create. In VB 9.0, when you omit this name, the compiler automatically generates the class for you. This class inherits from `Object` and overloads the `ToString`, `GetHashCode`, and `Equals` methods. The overloaded version of `ToString` returns a string representing all the properties concatenated together. The overloaded `Equals` method returns `True` if all property comparisons are `True` and there are the same number of properties in the same order with the same names.

Figure 1-13 shows the MSIL Disassembler tool (`Ildasm.exe`) displaying the MSIL that the compiler would automatically generate for the following example (see recipes 1-3 and 2-6 for more information about MSIL):

```
Dim person = New With { .FirstName = "Todd", .LastName = "Herman" }
```

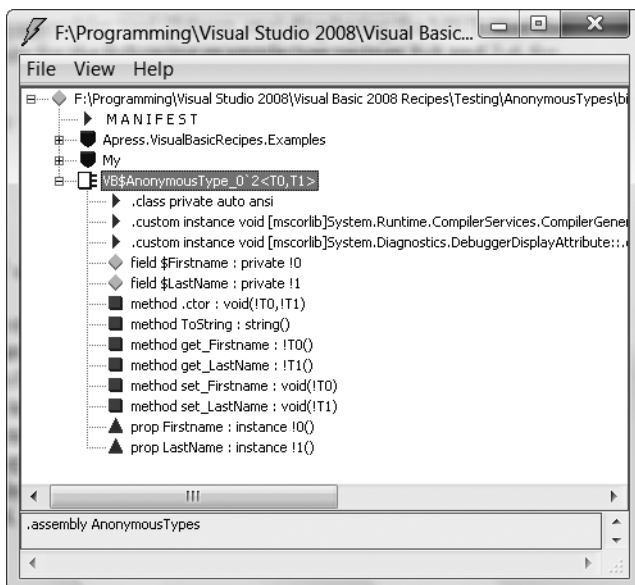


Figure 1-13. MSIL Disassembler tool view of an anonymous type

Creating anonymous types relies on several other new features of VB 9.0. As the name implies, the real name of an anonymous type is unknown. You will not be able to access it directly by its name and must rely on the variable used to first instantiate the class. This means you can't explicitly cast the person variable using `As`; you must rely on type inference (see recipe 1-19). Furthermore, an anonymous type *must* have at least one property. Properties for anonymous types are created by using object initializers (see recipe 1-20). The new version of Visual Studio fully supports the use of anonymous types by correctly displaying appropriate IntelliSense, as shown in Figure 1-14.

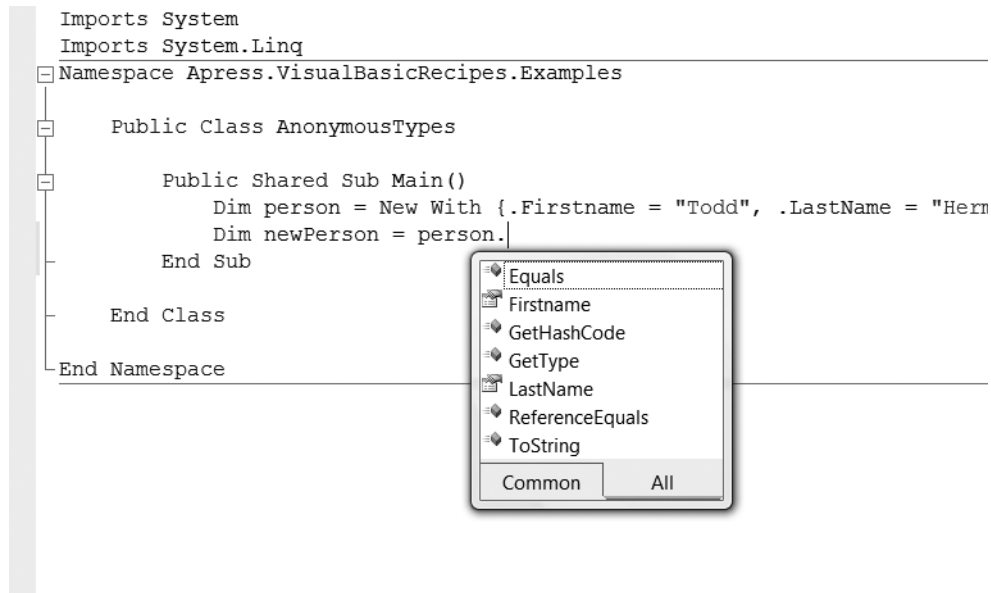


Figure 1-14. IntelliSense support for anonymous types

Anonymous types can also infer property names from object initializers, as in this example:

```
Dim person = New With {DateTime.Now, .FirstName = "Todd", .LastName = "Herman"}
```

In this case, the anonymous type created by the compiler would have the `Now`, `FirstName`, and `LastName` properties.

Anonymous types are a powerful new feature available in VB 9.0 and are used extensively in LINQ queries (see Chapters 6, 7, and 8) for returning strongly typed data.

1-22. Create and Use Extension Methods

Problem

You need to extend the functionality of a class without relying on inheritance or access to the actual class.

Solution

Create the method (a Sub or Function) you want to add, and then apply the `ExtensionAttribute` attribute to it.

How It Works

The key to using extension methods is the attribute `ExtensionAttribute`, which is new to VB 9.0 and located in the `System.Runtime.CompilerServices` namespace. You must apply this attribute to any method that you want to use as an extension method. Furthermore, you can apply the attribute only to methods defined within a `Module`.

An extension method *extends* the functionality of a specific class without actually modifying it. The class being extended is referenced by the first parameter of the extension method. Because of this, all extension methods *must* have at least one parameter, and it *must* refer to the class being extended.

```
<System.Runtime.CompilerServices.Extension(> _
Public Function Reverse(ByVal s As String) As String

    Dim reversed As New Text.StringBuilder(s.Length)
    Dim chars As Char() = s.ToCharArray

    For count As Integer = chars.Length - 1 To 0 Step -1
        reversed.Append(chars(count))
    Next

    Return reversed.ToString

End Function
```

The `Reverse` method is an extension method because it has the `ExtensionAttribute` attribute applied to it. You also know that it extends the `String` class because the first parameter is a `String`. Using an extension method is the same as calling any other method, and the Visual Studio IDE supports this via *IntelliSense*, as shown in Figure 1-15.

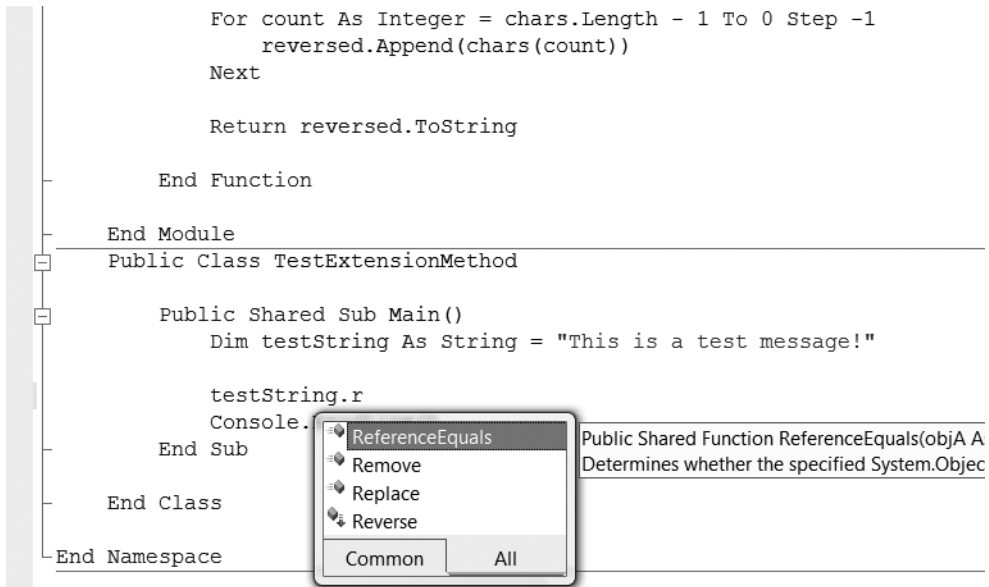


Figure 1-15. *IntelliSense support for extension methods*

In the case of the preceding example, you would create a `String` and then call the `Reverse` method, like this:

```
Dim testString As String = "This is a test message!"
Console.WriteLine(testString.Reverse())
```

This would produce the following result:

```
!egassem tset a si sihT
```

It is perfectly legitimate to call an extension method directly. When used in this manner, the first parameter of the method is used as an actual parameter. For example, you would get the same results if you changed the example to this:

```
Console.WriteLine(Reverse(testString))
```

The preceding example is fairly simple but demonstrates how easy it is to extend the functionality of a class without directly modifying it. What makes extension methods even more powerful is that they can also be used to extend base classes or even interfaces.

Extension methods are a key component of LINQ queries, which are covered in detail in Chapters 6, 7, and 8.

1-23. Create and Use Lambda Expressions

Problem

You need to use an inline function, which is a single-line function that does not require a standard function code block, in an effort to save some development time or support LINQ (discussed in more detail in Chapter 6).

Solution

Create a *lambda expression* using the `Function` keyword, and use it directly or pass it as an argument to a function that requires a delegate.

How It Works

To use a simple function, you typically start by creating the function. The following example takes an `Integer` and multiplies it by itself:

```
Private Shared Function Square(ByVal num As Integer) As Integer
    Return num * num
End Function
```

If you need to pass a function as an argument to some method, you could use a *delegate*. Delegates are used extensively by events and threading (discussed in Chapter 4) and by LINQ (discussed in Chapter 6). You accomplish this by using the `Delegate` keyword and using `AddressOf` to pass a reference to the function, as shown here:

```
Delegate Function CalculateDelegate(ByVal num As Integer) As Integer
```

```
Private Shared Sub Calculate(ByVal num As Integer, ➤
    ByVal calculation As CalculateDelegate)
    Console.WriteLine(calculation(num).ToString)
End Sub
```


The previous delegate and method would be used like this:

```
Call Calculate(5, AddressOf Square)
```

In the previous example, the Calculate method will call the Square function that was passed to it, using the number 5. This will result in the number 25 being written to the console.

Everything discussed earlier is how previous versions of VB .NET handle simple functions and delegates. VB .NET 9.0 supports the same methodology but offers a very powerful alternative for small functions that return a value from a single expression. This alternative is known as the *lambda expression*.

Lambda expressions are inline functions that are based on a form of calculus with the same name. The basic concept is to take the entire function and compress it into a single line. To do this with the Square function shown earlier, you would create a statement that looks similar to this:

```
Function(num) num * num
```

The statement starts with the Function keyword that includes the list of required parameters surrounded by parentheses. This is immediately followed by the expression that must be a single line that returns some value. The previous example can be simplified by deleting the Square function and changing the execution statement to the following:

```
Call Calculate(5, Function(num) num * num)
```

This works because lambda expressions are, at their core, delegates. The compiler creates an anonymous type (see recipe 1-21) that is instantiated and used by the receiving method. Figure 1-16 shows the generated anonymous delegate as shown in the MSIL Disassembler tool (Ildasm.exe).

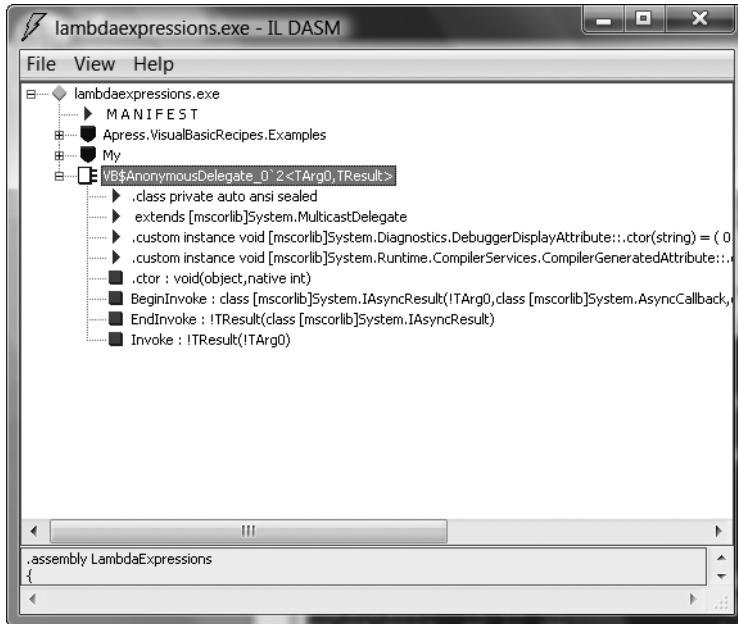


Figure 1-16. MSIL Disassembler tool view of an anonymous delegate

Lambda expressions can also be stored in a variable so it can be reused or more easily contained and passed to some method. Since VB 9.0 supports anonymous types and type inference (see recipe 1-19), you can leverage these features when using lambda expressions. For example, look at the following statement:

```
Dim calc = Function(num As Integer) num * num
```

In the previous statement, `calc` will be inferred as an anonymous delegate that meets the signature specified by the lambda expression. If you do not explicitly type the `num` parameter, then the data type for `calc` cannot be accurately inferred, resulting in an anonymous delegate whose parameters and return types are `Objects`.

To make storing lambda expressions even easier, .NET 3.5 includes the `System.Func` generic delegate. The `Func` delegate has five signatures that all include the data type of the returned value but vary depending on the number of arguments supported, which ranges from 0 to 5. With this in mind, you can change the previous example to use the `Func` delegate like this:

```
Dim calc As Func(Of Integer, Integer) = Function(num) num * num
```

The previous examples are all very basic in an attempt to simply explain the concepts of lambda expressions. The following example is a little more advanced and provides a more in-depth look at the power of lambda expressions:

```
Public Shared Sub Main()

    ' An array of numbers to be squared
    Dim numList() As Integer = {1, 2, 3, 4, 5, 6, 7, 8, 9}

    Console.WriteLine("Lambda Test: Square an array of numbers")
    Call Calculate(numList, Function(num) num * num)
    Console.ReadLine()

End Sub

' A method that executes the supplied function for each number
' in the supplied array.
Private Shared Sub Calculate(ByVal nums() As Integer, ➡
    ByVal calculation As Func(Of Integer, Integer))

    For Each num In nums
        ' Execute the lambda expression supplied and display the
        ' results to the console.
        Console.WriteLine(calculation(num).ToString)
    Next

End Sub
```

In this example, an array of `Integers` and a lambda expression to square numbers are passed to the `Calculate` method. The method loops through each `Integer` in the array and executes the provided lambda expression. The results would look similar to this:

Lambda Test: Square an array of numbers

1
4
9
16
25
36
49
64
81

Note LINQ (discussed in further detail in Chapter 6) relies heavily on extension methods (see recipe 1-22) that accept lambda expressions (in the form of a `Func`) as arguments.
