

39

Useful Namespaces

The .NET Framework is a library of classes, interfaces, and types that add extra power to Visual Studio .NET. These features go beyond what is normally provided by a programming language such as Visual Basic.

The .NET Framework is truly enormous. To make it more manageable, Microsoft has broken it into namespaces. The namespaces form a hierarchical catalog that groups related classes and functions in a meaningful way.

For example, the System namespace contains basic classes and methods that an application can use to perform common tasks. The System.Drawing namespace is the part of the System namespace that holds graphical tools. The System.Drawing.Design, System.Drawing.Drawing2D, System.Drawing.Imaging, System.Drawing.Printing, and System.Drawing.Text namespaces further subdivide System.Drawing into finer groupings.

Many of the .NET Framework namespaces are essential for day-to-day programming. For example, many Visual Basic applications need to produce printouts, so they use the System.Drawing.Printing namespace. Different applications draw graphics or images on the screen, so they need to use other System.Drawing namespaces.

Because so much of the .NET Framework is used in everyday programming tasks, this book doesn't strongly differentiate between Visual Basic and .NET Framework functionality. Presumably, the book could have focused solely on the Visual Basic language and ignored the .NET Framework, but it would have been a much less useful book.

Although the book covers many useful .NET Framework features, there's a huge amount that it doesn't cover. The .NET Framework includes hundreds of namespaces that define a huge number of classes, types, enumerated values, and other paraphernalia.

The following sections describe some of the highest-level and most useful namespaces provided by the .NET Framework.

ROOT NAMESPACES

Initially a Windows application includes two root namespaces: Microsoft and System.

NAMESPACES GALORE

Your program may include references to many other namespaces. If you add references to development libraries, your program will have access to their namespaces. For example, you might have Amazon.com, Google, eBay, and other development toolkits installed, and they come with their own namespaces. Later versions of Windows will also provide namespaces that you may want to reference.

Also note that the My namespace provides shortcuts that make common programming tasks easier. For more information on the My namespace, see the section “My” in Chapter 36, “Configuration and Resources,” and Appendix S, “The My Namespace.”

The Microsoft Namespace

The Microsoft root namespace contains Microsoft-specific items. In theory, any vendor can implement .NET languages that translate into Intermediate Language (IL) code. If you were to build such a language, the items in the Microsoft namespace would generally not apply to your language. Items in the System namespace described next would be as useful to users of your language as they are to programmers who use the Microsoft languages, but the items in the Microsoft namespace would probably not be as helpful.

The following table describes the most important second-level namespaces contained in the Microsoft root namespace.

NAMESPACE	CONTAINS
Microsoft.Csharp	Items supporting compilation and code generation for C#.
Microsoft.JScript	Items supporting compilation and code generation for JScript.
Microsoft.VisualBasic	Items supporting compilation and code generation for Visual Basic. Some of the items in this namespace are useful to Visual Basic programmers, mostly for compatibility with previous versions of Visual Basic.
Microsoft.Vsa	Items supporting Visual Studio for Applications (VSA), which lets you include scripting in your application.
Microsoft.WindowsCE	Items supporting Pocket PC and Smartphone applications using the .NET Compact Framework.
Microsoft.Win32	Classes that handle operating system events and that manipulate the System Registry.

The System Namespace

The System namespace contains basic classes used to define fundamental data types. It also defines important event handlers, interfaces, and exceptions.

The following table describes the second-level namespaces contained in the System root namespace.

NAMESPACE	CONTAINS
System.CodeDom	Classes for representing and manipulating source-code documents.
System.Collections	Interfaces and classes for defining various collection classes, lists, queues, hash tables, and dictionaries.
System.ComponentModel	Classes that control design time and runtime behavior of components and controls. Defines several useful code attributes such as Description, DefaultEvent, DefaultProperty, and DefaultValue. Also defines some useful classes such as ComponentResourceManager.
System.Configuration	Classes and interfaces for working with configuration files.
System.Data	Mostly classes for ADO.NET (the .NET version of ADO — ActiveX Data Objects). Sub-namespaces include features for specific kinds of databases and database technologies such as SQL Server, Oracle, OLE DB (Object Linking and Embedding), and so forth.
System.Deployment	Classes that let you programmatically update ClickOnce deployments.
System.Diagnostics	Classes for working with system processes, performance counters, and event logs.
System.DirectoryServices	Classes for working with Active Directory.
System.Drawing	Classes for using GDI+ graphics routines to draw two-dimensional graphics, text, and images.
System.EnterpriseServices	Tools for working with COM+ and building enterprise applications.
System.Globalization	Classes that help with internationalization. Includes tools for customizing an application's language and resources, and for using localized formats such as date, currency, and number formats.
System.IO	Classes for reading and writing streams and files.
System.Linq	Classes for LINQ. See Chapter 21, "LINQ," for more information.

continues

(continued)

NAMESPACE	CONTAINS
System.Management	Classes for system management and monitoring.
System.Media	Classes for playing sounds. For example, you can use the following code to play the system's "hand" sound: <code>System.Media.SystemSounds.Hand.Play()</code> Example program <code>SystemSounds</code> , which is available for download on the book's web site, uses this namespace to play the system sounds.
System.Messaging	Classes for working with message queues to send and receive messages across the network.
System.Net	Classes for working with network protocols.
System.Reflection	Classes for working with loaded types. A program can use these to learn about classes and their capabilities, and to invoke an object's methods.
System.Resources	Classes to create and manage culture-specific resources programmatically.
System.Runtime	Classes for working with metadata for compilers, interop services (interoperating with unmanaged code), marshalling, remoting, and serialization.
System.Security	Classes for security and cryptography.
System.ServiceProcess	Classes that let you implement, install, and control Windows service processes.
System.Text	Classes representing various character encodings. Also contains the <code>StringBuilder</code> class, which lets you build large strings quickly, and classes for working with regular expressions.
System.Threading	Classes for multithreading.
System.Timers	Timer class.
System.Transactions	Classes for working with transactions involving multiple distributed components and multiphase notifications.
System.Web	Classes for web programming and browser/server interactions.
System.Windows.Forms	Classes that define Windows forms controls (including the <code>Form</code> class itself).
System.Xml	Classes that let you manipulate XML files.

You can find more detailed information on these namespaces on Microsoft’s web pages. The URL for a namespace’s web page is “msdn.microsoft.com/” followed by the namespace followed by “.aspx” as in:

```
msdn.microsoft.com/system.codedom.aspx
msdn.microsoft.com/system.reflection.aspx
msdn.microsoft.com/system.windows.forms.aspx
```

ADVANCED EXAMPLES

Several chapters in this book cover pieces of the .NET Framework namespaces. For example, Chapter 36 describes many of the most useful tools provided by the System.Globalization and System.Resources namespaces. Similarly, Chapters 30 through 34 explain many of the most useful drawing tools provided by the System.Drawing namespace.

Other parts of the .NET Framework namespaces are quite specialized, and you may never need to use them. For example, many developers can use fairly standard installation techniques, so they will never need to use the System.Deployment classes to programmatically update ClickOnce deployments.

A few namespaces bear some special mention here, however. They are quite useful in many situations but they tend to stand separately rather than fitting nicely into one of the book’s major parts such as IDE, Object-Oriented Programming, or Graphics.

The following sections give a few examples that demonstrate some of the more useful of these namespaces.

Regular Expressions

A *regular expression* is a series of symbols that represents a class of strings. A program can use regular expression tools to determine whether a string matches a regular expression or to extract pieces of a string that match an expression. For example, a program can use regular expressions to see if a string has the format of a valid phone number, Social Security number, ZIP code or other postal code, e-mail address, and so forth.

The following regular expression represents a 7- or 10-digit phone number in the United States:

```
^([2-9]\d{2}-)?[2-9]\d{2}-\d{4}$
```

The following table describes the pieces of this expression.

SUBEXPRESSION	MEANING
^	(The caret symbol.) Matches the beginning of the string.
[2-9]	Matches the characters 2 through 9 (United States phone numbers cannot begin with 0 or 1).
\d	Matches any digit 0 through 9.
{2}	Repeats the previous group ([0-9]) exactly two times.

continues

(continued)

SUBEXPRESSION	MEANING
-	Matches a dash.
([2-9]\d{2}-)?	The parentheses group the items inside. The ? matches the previous item exactly zero or one times. Thus the subexpression matches three digits and a dash, all repeated zero or one times.
[2-9]\d{2}-	Matches one digit 2 through 9 followed by two digits 0 through 9 followed by a dash.
\d{4}	Matches any digit exactly four times.
\$	Matches the end of the string.

Taken together, this regular expression matches strings of the form NXX-XXXX and NXX-NXX-XXXX where N is a digit 2 through 9 and X is any digit.

A complete discussion of regular expressions is outside the scope of this book. Search the online help or the Microsoft web site to learn about the rules for building regular expressions. The web page msdn.microsoft.com/az24scfc.aspx provides useful links to information about regular expression language elements. Another useful page is www.regexlib.com/RETester.aspx, which provides a regular expression tester and a library of useful regular expressions.

As you read the rest of this section and when visiting regular expression web sites, be aware that there are a couple different types of regular expression languages, which won't all work with every regular expression class.

The following code shows how a program can validate a text field against a regular expression. When the user changes the text in the txtTestExp control, its Changed event handler creates a new Regex object, passing its constructor the regular expression held in the txtRegExp text box. It then calls the Regex object's IsMatch method to see if the text matches the regular expression. If the text matches, the program sets the txtTestExp control's background color to white. If the text doesn't match the expression, the program makes the control's background yellow to indicate an error.



```
Private Sub txtTestExp_TextChanged() Handles txtTestExp.TextChanged
    Dim reg_exp As New Regex(txtRegExp.Text)
    If reg_exp.IsMatch(txtTestExp.Text) Then
        txtTestExp.BackColor = Color.White
    Else
        txtTestExp.BackColor = Color.Yellow
    End If
End Sub
```

code snippet RegExValidate

The following example uses a Regex object's Matches method to retrieve a collection of Match objects that describe the places where a string matches a regular expression. It then loops through the collection, highlighting the matches in a Rich Text Box.



```
Private Sub btnGo_Click() Handles btnGo.Click
    Dim reg_exp As New Regex(txtPattern.Text)
    Dim matches As MatchCollection
    matches = reg_exp.Matches(txtTestString.Text)

    rchResults.Text = txtTestString.Text
    For Each a_match As Match In matches
        rchResults.Select(a_match.Index, a_match.Length)
        rchResults.SelectionBackColor = Color.Black
        rchResults.SelectionColor = Color.White
    Next a_match
End Sub
```

code snippet RegExHighlight

In this example, the regular expression is `(in|or)`, so the program finds matches where the string contains `in` or `or`.

The following code uses a `Regex` object to make replacements in a string. It creates a `Regex` object, passing its constructor the `IgnoreCase` option to tell the object to ignore capitalization in the string. It then calls the object's `Replace` method, passing it the string to modify and the pattern that it should use to make the replacement.



```
Dim reg_exp As New Regex(txtPattern.Text, RegexOptions.IgnoreCase)
lblResult.Text = reg_exp.Replace(Me.txtTestString.Text,
    txtReplacementPattern.Text)
```

code snippet RegExReplace

The `Regex` class can perform much more complicated matches. For example, you can use it to find fields within each line in a multiline string and then build a string containing the fields reformatted or reordered. See the online help for more details.

XML

Extensible Markup Language (XML) is a simple language for storing data in a text format. It encloses data within tags that delimit the data. You can give those tags any names that you want. For example, the following text shows an XML file containing three `Employee` records:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Employees>
  <Employee>
    <FirstName>Albert</FirstName>
    <LastName>Anders</LastName>
    <EmployeeId>11111</EmployeeId>
  </Employee>
  <Employee>
    <FirstName>Betty</FirstName>
    <LastName>Beach</LastName>
    <EmployeeId>22222</EmployeeId>
  </Employee>
```

```

    <Employee>
      <FirstName>Chuck</FirstName>
      <LastName>Cinder</LastName>
      <EmployeeId>33333</EmployeeId>
    </Employee>
  </Employees>

```

The System.Xml namespace contains classes for reading, writing, and manipulating XML data. Different classes let you process XML files in different ways. For example, the XmlDocument class lets you represent an XML document completely within memory. Using this class, you can perform complex manipulations of an XML file, adding and removing elements, searching for elements with particular attributes, and merging XML documents.

The XmlTextReader and XmlTextWriter classes let you read and write XML data in a fast, forward-only fashion. These classes can be more efficient than XmlDocument when you must quickly build or scan very large XML files that might not easily fit in memory all at once.

The following code shows one way a program can use the System.Xml namespace to generate the previous employee XML file:



Available for
download on
Wrox.com

```

Private Sub btnGo_Click() Handles btnGo.Click
    Dim xml_text_writer As _
        New XmlTextWriter("employees.xml", System.Text.Encoding.UTF8)

    ' Use indentation to make the result look nice.
    xml_text_writer.Formatting = Formatting.Indented
    xml_text_writer.Indentation = 4

    ' Write the XML declaration.
    xml_text_writer.WriteStartDocument(True)

    ' Start the Employees node.
    xml_text_writer.WriteStartElement("Employees")

    ' Write some Employee elements.
    MakeEmployee(xml_text_writer, "Albert", "Anders", 11111)
    MakeEmployee(xml_text_writer, "Betty", "Beach", 22222)
    MakeEmployee(xml_text_writer, "Chuck", "Cinder", 33333)

    ' End the Employees node.
    xml_text_writer.WriteEndElement()

    ' End the document.
    xml_text_writer.WriteEndDocument()

    ' Close the XmlTextWriter.
    xml_text_writer.Close()
End Sub

' Add an Employee node to the document.

```



```

Private Sub MakeEmployee(ByVal xml_text_writer As XmlTextWriter,
    ByVal first_name As String, ByVal last_name As String,
    ByVal emp_id As Integer)
    ' Start the Employee element.
    xml_text_writer.WriteStartElement("Employee")

    ' Write the FirstName.
    xml_text_writer.WriteStartElement("FirstName")
    xml_text_writer.WriteString(first_name)
    xml_text_writer.WriteEndElement()

    ' Write the LastName.
    xml_text_writer.WriteStartElement("LastName")
    xml_text_writer.WriteString(last_name)
    xml_text_writer.WriteEndElement()

    ' Write the EmployeeId.
    xml_text_writer.WriteStartElement("EmployeeId")
    xml_text_writer.WriteString(emp_id.ToString)
    xml_text_writer.WriteEndElement()

    ' Close the Employee element.
    xml_text_writer.WriteEndElement()
End Sub

```

code snippet BuildMemoryXml

The code starts by creating an `XmlTextWriter` object. This class provides methods for efficiently writing items into an XML file. The code sets the writer's `Formatting` and `Indentation` properties to make the object indent the resulting XML file nicely. If you don't set these properties, the file comes out all run together on a single line. That's fine for programs that process XML files but makes the file hard for humans to read.

The program calls the `WriteStartDocument` method to write the file's XML declaration, including the XML version, encoding, and standalone attribute. It calls `WriteStartElement` to write the starting `<Employees>` XML tag and then calls subroutine `MakeEmployee` to generate three Employee items. It calls the `WriteEndElement` method to write the `</Employees>` end tag, and calls `WriteEndDocument` to end the document. The program then closes the `XmlTextWriter` to close the file.

Subroutine `MakeEmployee` writes a starting `<Employee>` element into the file. It then uses the `WriteStartElement`, `WriteString`, and `WriteEndElement` methods to add the employee's `FirstName`, `LastName`, and `EmployeeId` elements to the document. The routine finishes by calling `WriteEndElement` to create the `</Employee>` end tag.

Other classes within the `System.Xml` namespace let you load and manipulate XML data in memory, read XML data in a fast forward-only manner, and search XML documents for elements matching certain criteria. XML is quickly becoming a common language that allows unrelated applications to communicate with each other. Using the XML tools provided by the `System.Xml` namespace, your application can read, write, and manipulate XML data, too.

Cryptography

The System.Security namespace includes objects for performing various cryptographic operations. The four main scenarios supported by these objects include the following:

- **Secret-key encryption** — This technique encrypts data so you cannot read it unless you know the secret key. This is also called *symmetric cryptography*.
- **Public-key encryption** — This technique encrypts data using a public key that everyone knows. Only the person with a secret private key can read the data. This is useful if you want to be the only one able to read messages anyone sends to you. This is also called *asymmetric cryptography*.
- **Signing** — This technique signs data to guarantee that it really came from a specific party. For example, you can sign an executable program to prove that it's really your program and not a virus substituted by some hacker.
- **Hashing** — This technique maps a piece of data such as a document into a hash value so it's very unlikely that two different documents will map to the same hash value. If you know a document's hash value, you can later hash the document again and compare the values. If the calculated value matches the previously known value, it is very unlikely that anyone has modified the file since the first hashing.

The example described later in this section encrypts and decrypts files. The basic idea is to create a CryptoStream object attached to a file stream opened for writing. As you write data into the CryptoStream, it encrypts or decrypts the data and sends the result to the output file stream.

Although the classes provided by Visual Studio are easier to use than the routines contained in the underlying cryptography API, the details are still somewhat involved. To encrypt and decrypt files, you must first select an encryption algorithm. You need to pick key and block sizes that are supported by the corresponding encryption provider.

To use an encryption provider, you must pass it a key and initialization vector (IV). Each of these is a series of bytes that the encryption provider uses to initialize its internal state before it encrypts or decrypts files.

If you want to control the encryption with a textual password, you must convert it into a series of bytes that you can use for the key and initialization vector. You can do that with a PasswordDeriveBytes object, but that object also requires the name of the hashing algorithm that it should use to convert the password into the key and initialization vector bytes.

Working through the following example should make this less confusing. Example program AesFile, which is available for download on the book's web site, uses the AES (Advanced Encryption Standard) algorithm to encrypt and decrypt files. The program uses the SHA384 hashing algorithm to convert a text password into key and initialization vector bytes. (For information on AES, see en.wikipedia.org/wiki/Advanced_Encryption_Standard. For information on SHA384, see en.wikipedia.org/wiki/Sha_hash.)



Available for
download on
Wrox.com

```
' Encrypt or decrypt a file, saving the results
' in another file.
Private Sub CryptFile(ByVal password As String, ByVal in_file As String,
    ByVal out_file As String, ByVal encrypt As Boolean)
    ' Create input and output file streams.
    Dim in_stream As New FileStream(in_file, FileMode.Open, FileAccess.Read)
    Dim out_stream As New FileStream(out_file, FileMode.Create, FileAccess.Write)

    ' Make an AES service provider.
    Dim aes_provider As New AesCryptoServiceProvider()

    ' Find a valid key size for this provider.
    Dim key_size_bits As Integer = 0
    For i As Integer = 1024 To 1 Step -1
        If aes_provider.ValidKeySize(i) Then
            key_size_bits = i
            Exit For
        End If
    Next i
    Debug.Assert(key_size_bits > 0)

    ' Get the block size for this provider.
    Dim block_size_bits As Integer = aes_provider.BlockSize

    ' Generate the key and initialization vector.
    Dim key As Byte() = Nothing
    Dim iv As Byte() = Nothing
    Dim salt As Byte() = {&H0, &H0, &H1, &H2, &H3, &H4, &H5,
        &H6, &Hf1, &Hf0, &HEE, &H21, &H22, &H45}
    MakeKeyAndIV(password, salt, key_size_bits, block_size_bits, key, iv)
    ' Make the encryptor or decryptor.
    Dim crypto_transform As ICryptoTransform
    If encrypt Then
        crypto_transform = aes_provider.CreateEncryptor(key, iv)
    Else
        crypto_transform = aes_provider.CreateDecryptor(key, iv)
    End If

    ' Attach a crypto stream to the output stream.
    Dim crypto_stream As New CryptoStream(out_stream, crypto_transform,
        CryptoStreamMode.Write)

    ' Encrypt or decrypt the file.
    Const BLOCK_SIZE As Integer = 1024
    Dim buffer(BLOCK_SIZE) As Byte
    Dim bytes_read As Integer
    Do
        ' Read some bytes.
        bytes_read = in_stream.Read(buffer, 0, BLOCK_SIZE)
        If bytes_read = 0 Then Exit Do

        ' Write the bytes into the CryptoStream.
        crypto_stream.Write(buffer, 0, bytes_read)
    
```

```
Loop

' Close the streams.
crypto_stream.Close()
in_stream.Close()
out_stream.Close()
End Sub

' Use the password to generate key bytes.
Private Sub MakeKeyAndIV(ByVal password As String, ByVal salt() As Byte,
    ByVal key_size_bits As Integer, ByVal block_size_bits As Integer,
    ByRef key As Byte(), ByRef iv As Byte())
    Dim derive_bytes As New Rfc2898DeriveBytes(txtPassword.Text, salt, 1000)

    key = derive_bytes.GetBytes(key_size_bits \ 8)
    iv = derive_bytes.GetBytes(block_size_bits \ 8)
End Sub
```

code snippet AesFile

Subroutine `CryptFile` encrypts or decrypts a file, saving the result in a new file. It takes as parameters a password string, the names of the input and output files, and a Boolean indicating whether it should perform encryption or decryption.

The routine starts by opening the input and output files. It then makes an `AesCryptoServiceProvider` object to provide the encryption and decryption algorithms using AES. The program must find a key length that is supported by the encryption service provider. This code counts backward from 1,024 until it finds a value that the provider's `ValidKeySize` method approves. On my computer, the largest key size the provider supports is 192 bits.

The AES algorithm encrypts data in blocks. The program uses the provider's `BlockSize` property to see how big those blocks are. The program must generate an initialization vector that has this same size.

The program calls the `MakeKeyAndIV` subroutine. This routine, which is described shortly, converts a text password into arrays of bytes for use as the key and initialization vector. The salt array contains a series of random bytes to make guessing the password harder for an attacker. The `Rfc2898DeriveBytes` class used by subroutine `MakeKeyAndIV` can generate a random salt for the program, but this example uses a salt array written into the code to make reading the code easier.

After obtaining the key and initialization vector, the program makes an object to perform the encryption or decryption transformation, depending on whether the subroutine's `encrypt` parameter is `True` or `False`. The program uses the encryption provider's `CreateEncryptor` or `CreateDecryptor` method, passing it the key and initialization vector.

Now, the program makes a `CryptoStream` object attached to its output file stream. It passes the object's constructor and output file stream, the cryptographic transformation object, and a flag indicating that the program will write to the stream.

At this point, the program has set the stage and can finally begin processing data. It allocates a buffer to hold data and then enters a `Do` loop. In the loop, it reads data from the input file into the buffer. If it reads no bytes, the program has reached the end of the input file, so it exits the loop. If it reads some bytes, the program writes them into the `CryptoStream`. The `CryptoStream` uses

its cryptographic transformation object to encrypt or decrypt the data and sends the result to its attached output file stream.

When it has finished processing the input file, the subroutine closes its streams.

Subroutine MakeKeyAndIV uses a text password to generate arrays of bytes to use as a key and initialization vector. It begins by creating an Rfc2898DeriveBytes object, passing to its constructor the password text, the salt, and the number of iterations the object should use to generate the random bytes. The salt can be any array of bytes as long as it's the same when encrypting and decrypting the file. The salt makes it harder for an attacker to build a dictionary of key and initialization vector values for every possible password string.

Having built the PasswordDeriveBytes object, the subroutine calls its GetBytes method to get the proper number of bytes for the key and initialization vector.

HOW EASY WAS THAT?

Previous editions of this book used the triple DES (Data Encryption Standard) algorithm to encrypt and decrypt files. However, DES is an old standard and cryptographers now recommend using AES instead.

The only thing I had to do to update this example was change the single statement that created the cryptographic service provider to

```
Dim aes_provider As New AesCryptoServiceProvider()
```

For clarity I also renamed the provider variable from `des_provider` to `aes_provider` but the update really only required changing a single statement. Setting up and using the cryptographic library takes a bit of work but the pieces are fairly interchangeable so switching algorithms is easy.

(Both the original program DesFile and new program AesFile are available for download on the book's web site.)

The following code uses the CryptFile subroutine to encrypt and then decrypt a file. First it calls CryptFile, passing it a password, input and output file names, and the value True to indicate that the routine should encrypt the file. Next, the code calls CryptFile again, this time to decrypt the encrypted file.

```
' Encrypt the file.
CryptFile(txtPassword.Text, txtPlaintextFile.Text, txtCyphertextFile.Text, True)

' Decrypt the file.
CryptFile(txtPassword.Text, txtCyphertextFile.Text, txtDecypheredFile.Text, False)
```

The DesFile example program, which is available for download on the book's web site, demonstrates the CryptFile subroutine. Enter some text and a password, and then click the left > button to encrypt the file. Click the right > button to decrypt the encrypted file and see if it matches the original text.

If you change the password by even a single character, the decryption returns gibberish. Figure 39-1 shows the program trying to decrypt a message incorrectly. Before the program tried to decrypt the file, I added an “s” to the end of the password. The result is completely unreadable.

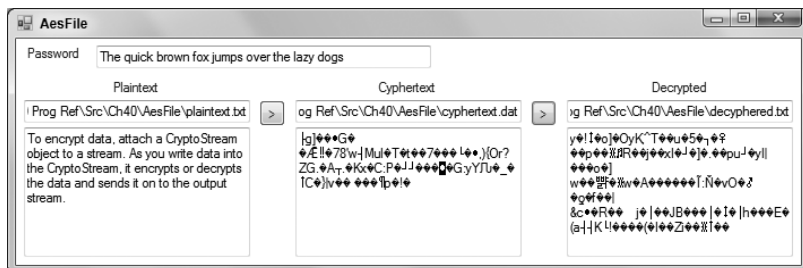


FIGURE 39-1: Changing even a single character in the password makes decryption produce an unintelligible result.

See the online help for information about the other main cryptographic operations (secret-key encryption, public-key encryption, signing, and hashing). Other books may also provide additional insights into cryptography. For example, the book *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition* (Schneier, Wiley Publishing, Inc., 1996) provides an excellent overview of modern cryptography and describes many important algorithms in detail. *Practical Cryptography* (Ferguson and Schneier, Wiley, 2003) provides a higher level executive summary of the algorithms and how to use them without covering implementation details. *Cryptography for Dummies* (Cobb, For Dummies, 2004) provides another high-level introduction to basic cryptographic concepts such as hashing and public key encryption.

Reflection

Reflection lets a program learn about itself and other programming entities. It includes objects that tell the program about assemblies, modules, and types.

Example program `ReflectionFormProperties` uses the following code to examine the program’s form and display a list of its properties, their types, and their values:



```
Private Sub Form1_Load() Handles MyBase.Load
    ' Make column headers.
    lvwProperties.View = View.Details
    lvwProperties.Columns.Clear()
    lvwProperties.Columns.Add("Property", 10,
        HorizontalAlignment.Left)
    lvwProperties.Columns.Add("Type", 10,
        HorizontalAlignment.Left)
    lvwProperties.Columns.Add("Value", 10,
        HorizontalAlignment.Left)

    ' List the properties.
    Dim property_value As Object
    Dim properties_info As PropertyInfo() =
        GetType(Form1).GetProperties()
```

```

lvwProperties.Items.Clear()
For i As Integer = 0 To properties_info.Length - 1
    With properties_info(i)
        If .GetIndexParameters().Length = 0 Then
            property_value = .GetValue(Me, Nothing)
            If property_value Is Nothing Then
                ListViewMakeRow(lvwProperties,
                                .Name,
                                .PropertyType.ToString,
                                " < Nothing > ")
            Else
                ListViewMakeRow(lvwProperties,
                                .Name,
                                .PropertyType.ToString,
                                property_value.ToString)
            End If
        Else
            ListViewMakeRow(lvwProperties,
                            .Name,
                            .PropertyType.ToString,
                            " < array > ")
        End If
    End With
Next i

' Size the columns to fit the data.
lvwProperties.Columns(0).Width = -2
lvwProperties.Columns(1).Width = -2
lvwProperties.Columns(2).Width = -2
End Sub

' Make a ListView row.
Private Sub ListViewMakeRow(ByVal lvw As ListView,
    ByVal item_title As String, ByVal ParamArray subitem_titles() As String)
    ' Make the item.
    Dim new_item As ListViewItem = lvw.Items.Add(item_title)

    ' Make the subitems.
    For i As Integer = subitem_titles.GetLowerBound(0) To _
        subitem_titles.GetUpperBound(0)
        new_item.SubItems.Add(subitem_titles(i))
    Next i
End Sub

```

code snippet ReflectionFormProperties

The program starts by formatting the ListView control named lvwProperties. Next, it defines an array of PropertyInfo objects named properties_info. It uses GetType to get type information about the Form1 class and then uses the type's GetProperties method to get information about the properties. The program then loops through the PropertyInfo objects.

If the object's `GetIndexParameters` array contains no entries, the property is not an array. In that case, the program uses the `PropertyInfo` object's `GetValue` method to get the property's value. The code then displays the property's name, type, and value.

If the `PropertyInfo` object's `GetIndexParameters` array contains entries, the property is an array. In that case, the program displays the property's name and type, and the string `<array>`.

The subroutine finishes by sizing the `ListView` control's columns and then making the form fit the columns.

The helper subroutine `ListViewMakeRow` adds a row of values to the `ListView` control. It adds a new item to the control and then adds subitems to the item. The item appears in the control's first column and the subitems appear in the other columns.

Using reflection to learn about your application is interesting, but not always necessary. After all, if you build an object, you probably know what its properties are.

Reflection can also tell you a lot about other applications. The `ReflectionGetResources` example program uses the following code to learn about another application. This program reads the assembly information in a file (example `ReflectionHasResources` is a resource-only DLL that this program can examine) and lists the embedded resources that it contains. The user can then select a resource to view it.



Available for
download on
Wrox.com

```
Private m_TargetAssembly As Assembly

' List the target assembly's resources.
Private Sub btnList_Click() Handles btnList.Click
    ' Get the target assembly.
    m_TargetAssembly = Assembly.LoadFile(txtFile.Text)

    ' List the target's manifest resource names.
    lstResourceFiles.Items.Clear()
    For Each str As String In m_TargetAssembly.GetManifestResourceNames()
        lstResourceFiles.Items.Add(str)
    Next str
End Sub

' List this file's resources.
Private Sub lstResourceFiles_SelectedIndexChanged()
    Handles lstResourceFiles.SelectedIndexChanged
        lstResources.Items.Clear()

        Dim resource_reader As ResourceReader
        resource_reader = New ResourceReader(
            m_TargetAssembly.GetManifestResourceStream(lstResourceFiles.Text))
        Dim dict_enumerator As IDictionaryEnumerator =
            resource_reader.GetEnumerator()
        While dict_enumerator.MoveNext()
            lstResources.Items.Add(New ResourceInfo(
                dict_enumerator.Key,
                dict_enumerator.Value))
        End While
        resource_reader.Close()
End Sub
```



```

' Display the selected resource.
Private Sub lstResources_SelectedIndexChanged() _
    Handles lstResources.SelectedIndexChanged
    lblString.Text = ""
    picImage.Image = Nothing
    Me.Cursor = Cursors.WaitCursor
    Refresh()

    Dim resource_info As ResourceInfo =
        DirectCast(lstResources.SelectedItem, ResourceInfo)
    Select Case resource_info.Value.GetType.Name
        Case "Bitmap"
            picImage.Image = CType(resource_info.Value, Bitmap)
            lblString.Text = ""
        Case "String"
            picImage.Image = Nothing
            lblString.Text = CType(resource_info.Value, String)
        Case Else
            ' Try to play it as audio.
            Try
                My.Computer.Audio.Play(resource_info.Value,
                    AudioPlayMode.WaitToComplete)
            Catch ex As Exception
                MessageBox.Show(resource_info.Key &
                    " has an unknown resource type",
                    "Unknown Resource Type", MessageBoxButtons.OK)
            End Try
        End Select

    Me.Cursor = Cursors.Default
End Sub

Private Class ResourceInfo
    Public Key As Object
    Public Value As Object
    Public Sub New(ByVal new_key As Object, ByVal new_value As Object)
        Key = new_key
        Value = new_value
    End Sub
    Public Overrides Function ToString() As String
        Return Key.ToString & " (" & Value.ToString & ")"
    End Function
End Class

```

code snippet ReflectionGetResources

The user enters the name of the assembly to load the txtFile text box. For example, this can be the name of a .NET executable program.

When the user clicks the List button, the btnList_Click event handler uses the Assembly class's shared LoadFile method to load an Assembly object representing the indicated assembly. It then loops through the array of strings returned by the Assembly object's GetManifestResourceNames method, adding the resource file names to the ListBox named lstResourceFiles.

When the user selects a resource file from the list, the `IstResourceFiles_SelectedIndexChanged` event handler displays a list of resources in the file. It uses the Assembly object's `GetManifestResourceStream` method to get a stream for the resources. It uses the stream to make a `ResourceReader` object and then enumerates the items found by the `ResourceReader`. It saves each object in a new `ResourceInfo` object (this class is described shortly) and adds it to the `IstResources` list.

When the user selects a resource from `IstResources`, its `SelectedIndexChanged` event handler retrieves the selected `ResourceInfo` object, converts its `Value` property into an appropriate data type, and displays the result. The `ResourceInfo` class stores `Key` and `Value` information for a resource enumerated by a `ResourceReader` object's enumerator. It provides an overloaded `ToString` that the `IstResources` list uses to represent the items.

This is admittedly a fairly complex example, but it performs the fairly remarkable feat of pulling resources out of another compile application.

Reflection can provide a lot of information about applications, modules, types, methods, properties, events, parameters, and so forth. It lets a program discover and invoke methods at runtime and build types at runtime.

An application also uses reflection indirectly when it performs such actions as serialization, which uses reflection to learn how to serialize and deserialize objects.

Reflection is a very advanced and somewhat arcane topic, but it is extremely powerful.

TPL

The *Task Parallel Library*, or *TPL*, is a set of tools that make building parallel programs easier. It provides a set of relatively simple method calls that launch multiple routines simultaneously on whatever processors are available.

Not long ago, only supercomputers contained multiple processing units, so only they could truly perform more than one task at the same time. Desktop operating systems switched rapidly back and forth between applications so it appeared as if the computer was performing a lot of tasks simultaneously, but in fact it was only doing one thing at a time.

More recently multi-processor computers are becoming quite common and relatively inexpensive. Practically any computer vendor sells computers with two or four processors. Soon it's likely that you'll be able to buy affordable computers with 8, 16, or possibly even dozens of processors.

The operating system can use some of this extra computing power transparently to make your system run more quickly, but if you have a computationally intensive application that hogs the processor, you must take special action if you want to get the full benefits of all of your processors.

To improve performance, you can launch multiple *threads of execution* to perform different tasks. If you run the threads on separate processors, they can do their work at the same time.

Unfortunately, writing safe and effective multi-threaded applications can be tricky. If you do it wrong, the threads will interfere with each other, possibly making the application crash or even take longer than it would on a single thread.

TPL is intended to make writing safe and effective multi-threaded applications easier. The TPL methods are lightweight and don't add too much overhead to an application so, if you need to perform several tasks at once and you have multiple processors available, your program will probably run faster. TPL overhead is fairly low, so even if you run the program on a single-processor system, you don't pay a huge penalty for trying to use multiple threads.

Getting Started

TPL is part of the `System.Threading` namespace. To make working with the namespace easier, you can add the following Imports statement at the top of your program files.

```
Imports System.Threading.Tasks
```

Now you're ready to use TPL. The following sections describe some of the most useful TPL methods: `Parallel.Invoke`, `Parallel.For`, and `Parallel.ForEach`.

Parallel.Invoke

The `Parallel` class contains methods for launching parallel threads. The `Parallel.Invoke` takes as parameters a series of `System.Action` objects that give it information about the tasks it should launch.

The `System.Action` class is actually just a named delegate representing a subroutine that takes no parameters so you can use the address of any subroutine.

Example program `ParallelInvoke` shown in Figure 39-2 demonstrates the `Parallel.Invoke` method. As you can see, the parallel version was significantly faster on my dual-core system.

The following code shows how program `ParallelInvoke` uses `Parallel.Invoke`:

```
Parallel.Invoke(
    AddressOf Fibonacci0,
    AddressOf Fibonacci1,
    AddressOf Fibonacci2,
    AddressOf Fibonacci3)
```

The four Fibonacci routines simply evaluate the Fibonacci number for various values. For example, the `Fibonacci0` function shown in the following code gets the first text box's value stored in the `Numbers` array, calls the `Fibonacci` function, and saves the result in the `Results` array.

```
Private Sub Fibonacci0()
    Results(0) = Fibonacci(Numbers(0))
End Sub
```

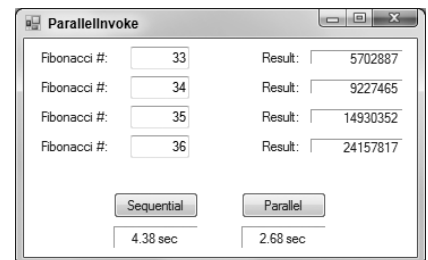


FIGURE 39-2: `Parallel.Invoke` runs several subroutines on multiple threads.

FIBONACCI FUN

The Fibonacci sequence is defined recursively by $\text{Fibonacci}(0) = 1$, $\text{Fibonacci}(1) = 1$, and for larger values of N $\text{Fibonacci}(N) = \text{Fibonacci}(N - 1) + \text{Fibonacci}(N - 2)$. The first 10 values are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and 89.

The function grows fairly quickly so, as you can see in Figure 39-2, $\text{Fibonacci}(36) = 24,157,817$.

More importantly for this example is the fact that the recursive definition is an inefficient way to calculate Fibonacci numbers. To see why, consider that calculating $\text{Fibonacci}(N)$ requires calculating $\text{Fibonacci}(N - 1)$ and $\text{Fibonacci}(N - 2)$. But calculating $\text{Fibonacci}(N - 1)$ also requires calculating $\text{Fibonacci}(N - 2)$, so that value is calculated twice. During the course of a calculation, intermediate values are calculated a huge number of times so the code takes a while and the example has some nice long routines to parallelize.

The following code shows the Fibonacci function.

```
Private Function Fibonacci(ByVal N As Long) As Long
    If N <= 1 Then Return 1

    Return Fibonacci(N - 1) + Fibonacci(N - 2)
End Function
```

Before calling `ParallelInvoke`, this example stores the values N in an array. Each of the routines looks only at its array entry and places its result in its own separate variable, which is also stored in an array. That means the routines never read or write each other's values. This is important because parallel routines that work with the same variables may interfere with each other.

For example, suppose one routine sets a variable's value to 10 and another routine sets the same variable's value to 20. If the routines are running at the same time, you can't tell which routine gets there first, so you don't know what value the variable holds at the end.

The following list summarizes some of the details you need to consider when working with multiple threads:

- Two threads trying to access the same variables can interfere with each other.
- Two threads trying to lock several shared resources can form a deadlock where neither can continue until the other finishes.
- Parallel threads cannot directly access the user interface thread, so they cannot safely use control properties.
- Some classes are not "thread-safe" so you cannot safely use them in multiple threads at the same time.

As long as separate threads use only their own variables and don't try to interact with the user interface thread, `Parallel.Invoke` is remarkably easy to use.

Parallel.For

The `Parallel.For` method lets you invoke a single subroutine while passing it a series of numeric values.

Example program `ParallelFor` performs calculations similar to those performed by program `Parallel`. Invoke except it uses the `Parallel.For` method.

This version calls the `FindFibonacci` subroutine shown in the following code:

```
Private Sub FindFibonacci(ByVal index As Integer)
    Results(index) = Fibonacci(Numbers(index))
End Sub
```

The `index` parameter tells which entry in the `Numbers` array is the Fibonacci number that the routine should calculate. The `Numbers` array holds the numbers entered in the text boxes shown in Figure 39-2. The code calls the `Fibonacci` function and saves the results in the `Results` array.

The following code shows how the program calls subroutine `FindFibonacci` sequentially, passing it the values 0 through 3:

```
For i As Integer = 0 To 3
    FindFibonacci(i)
Next i
```

The program uses the following code to make the same subroutine calls in parallel:

```
Parallel.For(0, 3, AddressOf FindFibonacci)
```

The results of the two calculations are the same but the parallel version takes only about 41% as long on my dual-core system.

`Parallel.For` is most useful when you need to call a routine many times with different numeric inputs.

Parallel.ForEach

As you may be able to guess, the `Parallel.ForEach` method is similar to `Parallel.For` except it passes a series of objects from a collection into the subroutine instead of a series of sequential values.

Example program `ParallelForEach` performs the same Fibonacci calculations as the previous examples except it uses the `Parallel.ForEach` method. The following code shows the `FiboInfo` class that it passes in the parallel subroutine calls:

```
Private Class FiboInfo
    Public N As Long
    Public Result As Long
End Class
```

The following code shows the new `FindFibonacci` subroutine. The `FiboInfo` parameter both tells the routine which Fibonacci number to calculate and holds the result.

```
Private Sub FindFibonacci(ByVal fibo_info As FiboInfo)
    fibo_info.Result = Fibonacci(fibo_info.N)
End Sub
```

The following code shows the key parallel pieces of example program `ParallelForEach`. This code first initializes the `fibo_info` array and then uses `Parallel.ForEach` to pass its values to different calls to subroutine `FindFibonacci`.

```
Dim fibo_info() As FiboInfo = {
    New FiboInfo() With {.N = CLng(txtNum0.Text)},
    New FiboInfo() With {.N = CLng(txtNum1.Text)},
    New FiboInfo() With {.N = CLng(txtNum2.Text)},
    New FiboInfo() With {.N = CLng(txtNum3.Text)}
}

Parallel.ForEach(fibo_info, AddressOf FindFibonacci)
```

Once again, the results are the same as in the previous examples, but the parallel version takes less time than the sequential version.

There are still plenty of TPL details that I don't have room to cover here. The library provides other classes and methods for executing tasks in parallel and there are many ways you can coordinate among different threads. For more information on TPL, search the Web for articles such as these two of mine posted by DevX.com:

- Getting Started with the .NET Task Parallel Library (www.devx.com/dotnet/Article/39204)
- Getting Started with the .NET Task Parallel Library: Multi-Core Case Studies (www.devx.com/dotnet/Article/39219)

These articles contain more detailed information and other examples.

SUMMARY

The .NET Framework defines hundreds of namespaces, and this chapter described only a few. It provided a brief overview of some of the most important System namespaces and gave more detailed examples that demonstrated regular expressions, XML, cryptography, reflection, and TPL.

Even in these somewhat specialized areas, the examples can cover only a tiny fraction of the capabilities of the namespaces; however, the examples should give you an idea of the types of features that these namespaces can add to your application. If you need to do something similar, they will hopefully inspire you to do more in-depth research so that you can take full advantage of these powerful tools.

The chapters in this book cover a wide variety of Visual Basic programming topics. In the first part of the book, Chapters 1 through 7 describe the Visual Studio integrated development environment and many of the tools that you use to build Visual Basic programs. In the second part of the book, Chapters 8 through 24 explained basic topics of Visual Basic programming (such as the language

itself, using standard controls, and drag and drop). In the third part of the book, Chapters 25 through 29 describe object-oriented concepts (such as class and structure declaration, namespaces, and generics). In the fourth part of the book, Chapters 30 through 35 cover graphical topics (such as how to draw shapes and text, image manipulation, printing, and report generation). In the fifth part of the book, Chapters 36 through 40 explain ways a program can interact with its environment by using techniques such as configuration files, the Registry, streams, and file-system objects.

The rest of this book contains appendices that provide a categorized reference for Visual Basic .NET. You can use them to review quickly the syntax of a particular command, select from among several overloaded versions of a routine, or refresh your memory of what a particular class can do.

