

# Visual Basic 2005

---

*A Developer's  
Notebook™*

Matthew MacDonald

O'REILLY®

# The Visual Basic Language

When Visual Basic .NET first appeared, loyal VB developers were shocked to find dramatic changes in their favorite language. Suddenly, common tasks such as instantiating an object and declaring a structure required new syntax, and even basic data types like the array had been transformed into something new. Fortunately, Visual Basic 2005 doesn't have the same shocks in store. The language changes in the latest version of VB are refinements that simplify life *without* making any existing code obsolete. Many of these changes are language features imported from C# (e.g., operator overloading), while others are completely new ingredients that have been built into the latest version of the common language runtime (e.g., generics). In this chapter, you'll learn about all the most useful changes to the VB language.

## Use the My Objects to Program Common Tasks

The new My objects provide easy access to various features that developers often need but don't necessarily know where to find in the sprawling .NET class library. Essentially, the My objects offer one-stop shopping, with access to everything from the Windows registry to the current network connection. Best of all, the My object hierarchy is organized according to use and is easy to navigate using Visual Studio IntelliSense.

### *In this chapter:*

- *Use the My Objects to Program Common Tasks*
- *Get Application Information*
- *Use Strongly Typed Resources*
- *Use Strongly Typed Configuration Settings*
- *Build Typesafe Generic Classes*
- *Make Simple Data Types Nullable*
- *Use Operators with Custom Objects*
- *Split a Class into Multiple Files*
- *Extend the My Namespace*
- *Skip to the Next Iteration of a Loop*
- *Dispose of Objects Automatically*

*Tired of hunting through the extensive .NET class library in search of what you need? With the new My objects, you can quickly find some of the most useful features .NET has to offer.*

## How do I do that?

There are seven first-level My objects. Out of these, three core objects centralize functionality from the .NET Framework and provide computer information. These include:

### My.Computer

This object provides information about the current computer, including its network connection, the mouse and keyboard state, the printer and screen, and the clock. You can also use this object as a jumping-off point to play a sound, find a file, access the registry, or use the Windows clipboard.

### My.Application

This object provides information about the current application and its context, including the assembly and its version, the folder where the application is running, the culture, and the command-line arguments that were used to start the application. You can also use this object to log an application event.

### My.User

This object provides information about the current user. You can use this object to check the user's Windows account and test what groups the user is a member of.

Along with these three objects, there are another two objects that provide *default instances*. Default instances are objects that .NET creates automatically for certain types of classes defined in your application. They include:

### My.Forms

This object provides a default instance of each Windows form in your application. You can use this object to communicate between forms without needing to track form references in another class.

### My.WebServices

This object provides a default proxy-class instance for every web service. For example, if your project uses two web references, you can access a ready-made proxy class for each one through this object.

Finally, there are two other My objects that provide easy access to the configuration settings and resources:

## My.Settings

This object allows you to retrieve custom settings from your application's XML configuration file.

## My.Resources

This object allows you to retrieve *resources*—blocks of binary or text data that are compiled into your application assembly. Resources are typically used to store localized strings, images, and audio files.

---

### WARNING

Note that the My objects are influenced by the project type. For example, when creating a web or console application, you won't be able to use My.Forms.

---

Some of the My classes are defined in the Microsoft.VisualBasic.MyServices namespace, while others, such as the classes used for the My.Settings and My.Resources objects, are created dynamically by Visual Studio 2005 when you modify application settings and add resources to the current project.

To try out the My object, you can use Visual Studio IntelliSense. Just type My, followed by a period, and take a look at the available objects, as shown in Figure 2-1. You can choose one and press the period again to step down another level.

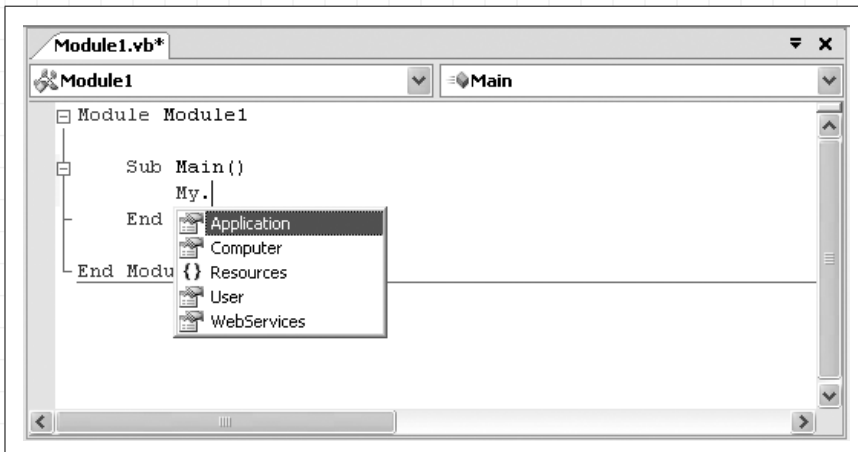


Figure 2-1. Browsing the My objects

To try a simple example that displays some basic information using the `My` object, create a new console project. Then, add this code to the `Main()` routine:

```
Console.WriteLine(My.Computer.Name)
Console.WriteLine(My.Computer.Clock.LocalTime)
Console.WriteLine(My.Application.CurrentDirectory)
Console.WriteLine(My.User.Identity.Name)
```

When you run this code, you'll see some output in the console window, which shows the computer name, current time, application directory, and user:

```
SALESSERVER
2005-10-1 8:08:52 PM
C:\Code\VBNotebook\1.07\MyTest\bin
MATTHEW
```

---

### WARNING

The `My` object also has a “dark side.” Use of the `My` object makes it more difficult to share your solution with non-VB developers, because other languages, such as C#, don't have the same feature.

---

## Where can I learn more?

You can learn more about the `My` object and see examples by looking up the “`My` Object” index entry in the MSDN Help. You can also learn more by examining some of this book's other labs that use the `My` object. Some examples include:

- Using `My.Application` to retrieve details of your program, such as the current version and the command-line parameters used to start it (see the “Get Application Information” lab in this chapter).
- Using `My.Resources` to load images and other resources from the application assembly (see the “Use Strongly Typed Resources” lab in this chapter).
- Using `My.Settings` to retrieve application and user settings (see the “Use Strongly Typed Configuration Settings” lab in this chapter).
- Using `My.Forms` to interact between application windows (see the “Communicate Between Forms” lab in Chapter 3).
- Using `My.Computer` to perform file manipulation and network tasks in Chapters 5 and 6.
- Using `My.User` to authenticate the current user (see the “Test Group Membership of the Current User” lab in Chapter 6).

# Get Application Information

The `My.Application` object provides a wealth of information right at your fingertips. Getting this information is as easy as retrieving a property.

## How do I do that?

The information in the `My.Application` object comes in handy in a variety of situations. Here are two examples:

- You want to get the exact version number. This could be useful if you want to build a dynamic About box, or check with a web service to make sure you have the latest version of an assembly.
- You want to record some diagnostic details. This becomes important if a problem is occurring at a client site and you need to log some general information about the application that's running.

To create a straightforward example, you can use the code in Example 2-1 in a console application. It retrieves all of these details and displays a complete report in a console window.

### Example 2-1. Retrieving information from `My.Application`

```
' Find out what parameters were used to start the application.
Console.WriteLine("Command line parameters: ")
For Each Arg As String In My.Application.CommandLineArgs
    Console.WriteLine(Arg & " ")
Next
Console.WriteLine()
Console.WriteLine()

' Find out some information about the assembly where this code is located.
' This information comes from metadata (attributes in your code).
Console.WriteLine("Company: " & My.Application.Info.CompanyName)
Console.WriteLine("Description: " & My.Application.Info.Description)
Console.WriteLine("Located in: " & My.Application.Info.DirectoryPath)
Console.WriteLine("Copyright: " & My.Application.Info.Copyright)
Console.WriteLine("Trademark: " & My.Application.Info.Trademark)
Console.WriteLine("Name: " & My.Application.Info.AssemblyName)
Console.WriteLine("Product: " & My.Application.Info.ProductName)
Console.WriteLine("Title: " & My.Application.Info.Title)
Console.WriteLine("Version: " & My.Application.Info.Version.ToString())
Console.WriteLine()
```

*Using the `My.Application` object, you can get information about the current version of your application, where it's located, and what parameters were used to start it.*

---

## TIP

Visual Studio 2005 includes a Quick Console window that acts as a lightweight version of the normal command-line window. In some cases, this window is a little buggy. If you have trouble running a sample console application and seeing its output, just disable this feature. To do so, select Tools → Options, make sure the “Show all settings” checkbox is checked, and select the Debugging → General tab. Then turn off “Redirect all console output to the Quick Console window.”

---

Before you test this code, it makes sense to set up your environment to ensure that you will see meaningful data. For example, you might want to tell Visual Studio to supply some command-line parameters when it launches the application. To do this, double-click the My Project icon in the Solution Explorer. Then, choose the Debug tab and look for the “Command line parameters” text box. For example, you could add three parameters by specifying the command line `/a /b /c`.

If you want to set information such as the assembly author, product, version, and so on, you need to add special attributes to the `AssemblyInfo.vb` file, which isn’t shown in the Solution Explorer. To access it, you need to select Solution → Show All Files. You’ll find the `AssemblyInfo.vb` file under the My Projects node. Here’s a typical set of tags that you might enter:

```
<Assembly: AssemblyVersion("1.0.0.0")>
<Assembly: AssemblyCompany("Prosetech")>
<Assembly: AssemblyDescription("Utility that tests My.Application")>
<Assembly: AssemblyCopyright("(C) Matthew MacDonald")>
<Assembly: AssemblyTrademark("(R) Prosetech")>
<Assembly: AssemblyTitle("Test App")>
<Assembly: AssemblyProduct("Test App")>
```

All of this information is embedded in your compiled assembly as metadata.

Now you can run the test application. Here’s an example of the output you’ll see:

```
Command line parameters: /a /b /c

Company: Prosetech
Description: Utility that tests My.Application
Located in: C:\Code\VBNotebook\1.08\ApplicationInfo\bin
Copyright: (C) Matthew MacDonald
Trademark: (R) Prosetech
Name: ApplicationInfo.exe
Product: Test App
Title: Test App
Version: 1.0.0.0
```

*New in VB 2005 is the ability to add application information in a special dialog box. To use this feature, double-click the My Project item in the Solution Explorer, select the Assembly tab, and click the Assembly Information button.*

## What about...

...getting more detailed diagnostic information? The `My.Computer.Info` object also provides a dash of diagnostic details with two useful properties. `LoadedAssemblies` provides a collection with all the assemblies that are currently loaded (and available to your application). You can also examine their version and publisher information. `StackTrace` provides a snapshot of the current stack, which reflects where you are in your code. For example, if your `Main()` method calls a method named `A()` that then calls method `B()`, you'll see three of your methods on the stack—`B()`, `A()`, and `Main()`—in reverse order.

Here's the code you can add to start looking at this information:

```
Console.WriteLine("Currently loaded assemblies")
For Each Assm As System.Reflection.Assembly In _
    My.Application.Info.LoadedAssemblies
    Console.WriteLine(Assm.GetName().Name)
Next
Console.WriteLine()

Console.WriteLine("Current stack trace: " & My.Application.Info.StackTrace)
Console.WriteLine()
```

## Use Strongly Typed Resources

In addition to code, .NET assemblies can also contain *resources*—embedded binary data such as images and hardcoded strings. Even though .NET has supported a system of resources since Version 1.0, Visual Studio hasn't included integrated design-time support. As a result, developers who need to store image data usually add it to a control that supports it at design time, such as a `PictureBox` or `ImageList`. These controls insert the picture data into the application resource file automatically.

In Visual Studio 2005, it's dramatically easier to add information to the resources file and update it afterward. Even better, you can access this information in a strongly typed fashion from anywhere in your code.

## How do I do that?

In order to try using a strongly typed resource of an image in this lab, you need to create a new Windows application before continuing.

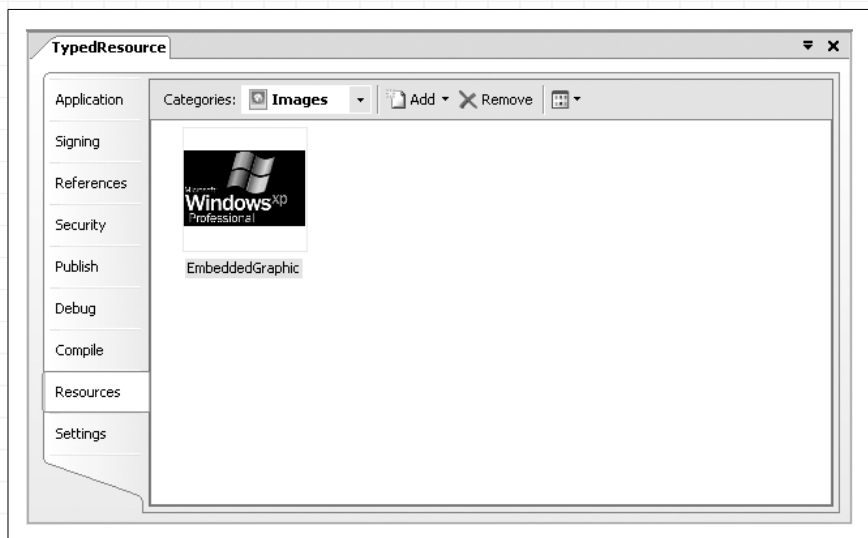
To add a resource, start by double-clicking the `My Project` node in the Solution Explorer. This opens up the application designer, where you can configure a host of application-related settings. Next, click the `Resources`

*Strongly typed resources let you embed static data such as images into your compiled assemblies, and access it easily in your code.*

tab. In the Categories drop-down listbox, select the type of resources you want to see (strings, images, audio, and so on). The string view shows a grid of settings. The image view is a little different—by default, it shows a thumbnail of each picture.

To add a new picture, select the Images category from the drop-down list and then select Add → Existing File from the toolbar. Browse to an image file, select it, and click OK. If you don't have an image file handy, try using one from the *Windows* directory, such as *winnt256.bmp* (which is included with most versions of Windows).

By default, the resource name has the same name as the file, but you can rename it after adding it. In this example, rename the image to EmbeddedGraphic (as shown in Figure 2-2).



**Figure 2-2.** Adding a picture as a strongly typed resource

*The resources class is added in the My Project directory and is given the name Resources.Designer.vb. To see it, you need to choose Project → Show All Files. Of course, you should never change this file by hand.*

Using a resource is easy. All resources are compiled dynamically into a strongly typed resource class, which you can access through `My.Resources`. To try out this resource, add a `PictureBox` control to your Windows form (and keep the default name `PictureBox1`). Then, add the following code to show the image when the form loads:

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
    PictureBox1.Image = My.Resources.EmbeddedGraphic  
  
End Sub
```

If you run the code, you'll see the image appear on the form. To make sure the image is being extracted from the assembly, try compiling the application and then deleting the image file (the code will still work seamlessly).

When you add a resource in this way, Visual Studio copies the resource to the *Resources* subdirectory of your application. You can see this directory, along with all the resources it contains, in the Solution Explorer. When you compile your application, all the resources are embedded in the assembly. However, there's a distinct advantage to maintaining them in a separate directory. This way, you can easily update a resource by replacing the file and recompiling the application. You don't need to modify any code. This is a tremendous benefit if you need to update a number of images or other resources at once.

You can also attach a resource to various controls using the Properties window. For example, when you click the ellipsis (...) in the Properties window next to the Image property for the PictureBox control, a designer appears that lists all the pictures that are available in the application's resources.

*Another advantage of resources is that you can use the same images in multiple controls on multiple different forms, without needing to add more than one copy of the same file.*

## What about...

...the ImageList? If you're a Windows developer, you're probably familiar with the ImageList control, which groups together multiple images (usually small bitmaps) for use in other controls, such as menus, toolbars, trees, and lists. The ImageList doesn't use typed resources. Instead, it uses a custom serialization scheme. You'll find that although the ImageList provides design-time support and programmatic access to the images it contains, this access isn't strongly typed.

## Use Strongly Typed Configuration Settings

Applications commonly need configuration settings to nail down details like file locations, database connection strings, and user preferences. Rather than hardcoding these settings (or inventing your own mechanism to store them), .NET lets you add them to an application-specific configuration file. This allows you to adjust values on a whim by editing a text file without recompiling your application.

In Visual Studio 2005, configuration settings are even easier to use. That's because they're automatically compiled into a custom class that provides strongly typed access to them. That means you can retrieve

*Use error-proof configuration settings by the application designer.*

settings using properties, with the help of IntelliSense, instead of relying on string-based lookups. Even better, .NET enhances this model with the ability to use updatable, user-specific settings to track preferences and other information. You'll see both of these techniques at work in this lab.

## How do I do that?

Every custom configuration setting is defined with a unique string name. In previous versions of .NET, you could retrieve the value of a configuration setting by looking up the value by its string name in a collection. However, if you use the wrong name, you wouldn't realize your error until you run the code and it fails with a runtime exception.

In Visual Studio 2005, the story is much improved. To add a new configuration setting, double-click the My Project node in the Solution Explorer. This opens up the application designer where you can configure a host of application-related settings. Next, click the Settings tab, which shows a list of custom configuration settings where you can define new settings and their values.

To add a custom configuration setting to your application, enter a new setting name at the bottom of the list. Then specify the data type, scope, and the actual content of the setting. For example, to add a setting with a file path, you might use the name `UserDataFilePath`, the type `String`, the scope `Application` (you'll learn more about this shortly), and the value `c:\MyFiles`. Figure 2-3 shows this setting.

*In a web application, configuration settings are placed in the web.config file. In other applications, application settings are recorded to a configuration file that takes the name of the application, plus the extension .config, as in MyApp.exe.config.*

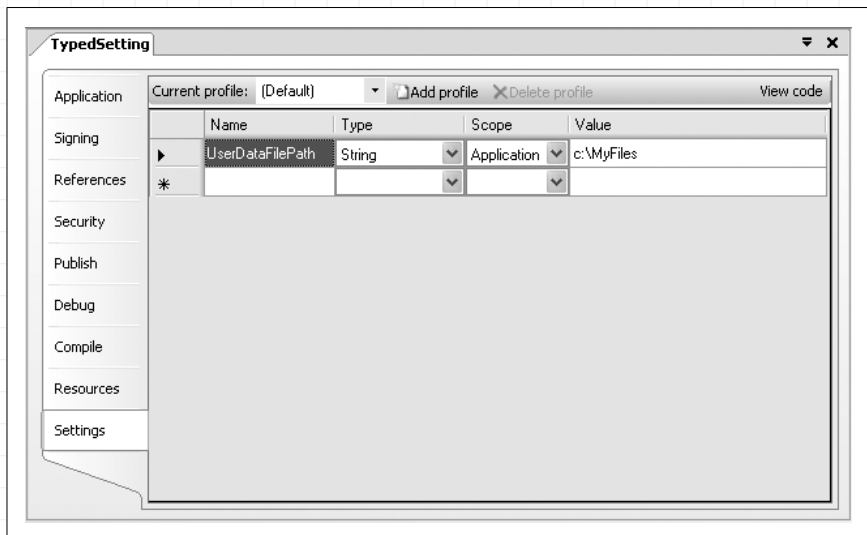


Figure 2-3. Defining a strongly typed application setting

When you add the setting, Visual Studio .NET inserts the following information into the application configuration file:

```
<configuration>
  <!-- Other settings are defined here. -->
  <applicationSettings>
    <WindowsApplication1.MySettings>
      <setting name="UserDataFilePath" serializeAs="String">
        <value>c:\MyFiles</value>
      </setting>
    </WindowsApplication1.MySettings>
  </applicationSettings>
</configuration>
```

At the same time behind the scenes, Visual Studio compiles a class that includes information about your custom configuration setting. Then, you can access the setting by name anywhere in your code through the `My.Settings` object. For example, here's code that retrieves the setting named `UserDataFilePath`:

```
Dim path As String
path = My.Settings.UserDataFilePath
```

In .NET 2.0, configuration settings don't need to be strings. You can also use other serializable data types, including integers, decimals, dates, and times (just choose the appropriate data type from the Types drop-down list). These data types are serialized to text in the configuration file, but you can retrieve them through `My.Settings` as their native data type, with no parsing required!

*The application settings class is added in the My Project directory and is named Settings.Designer.vb. To see it, select Project → Show All Files.*

## What about...

...updating settings? The `UserDataFilePath` example uses an *application-scoped* setting, which can be read at runtime but can't be modified. If you need to change an application-scoped setting, you have to modify the configuration file by hand (or use the settings list in Visual Studio).

Your other choice is to create *user-scoped* settings. To do this, just choose `User` from the Scope drop-down list in the settings list. With a user-scoped setting, the value you set in Visual Studio is stored as the default in the configuration file in the application directory. However, when you change these settings, a new *user.config* file is created for the current user and saved in a user-specific directory (with a name in the form `c:\Documents and Settings\[UserName]\Local Settings\Application Data\[ApplicationName]\[UniqueDirectory]`).

The only trick pertaining to user-specific settings is that you *must* call `My.Settings.Save()` to store your changes. Otherwise, changes will only

persist until the application is closed. Typically, you'll call `My.Settings.Save()` when your application ends.

To try out a user-scoped setting, change the scope of the `UserDataFilePath` setting from `Application` to `User`. Then, create a form that has a text box (named `txtFilePath`) and two buttons, one for retrieving the user data (`cmdRefresh`) and one for changing it (`cmdUpdate`). Here are the event handlers you'll use:

```
Private Sub cmdRefresh_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles cmdRefresh.Click  
    txtFilePath.Text = My.Settings.UserDataFilePath  
End Sub
```

```
Private Sub cmdUpdate_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles cmdUpdate.Click  
    My.Settings.UserDataFilePath = txtFilePath.Text  
End Sub
```

Finally, to make sure your changes are there the next time you run the application, tell .NET to create or update the `user.config` file when the form closes with this code:

```
Private Sub Form1_FormClosed(ByVal sender As Object, _  
    ByVal e As System.Windows.Forms.FormClosedEventArgs) _  
    Handles Me.FormClosed  
    My.Settings.Save()  
End Sub
```

This rounds out a simple test form. You can run this application and try alternately retrieving the current setting and storing a new one. If you're interested, you can then track down the `user.config` file that has the changed settings for the current user.

*Need to create a class that's flexible enough to work with any type of object, but able to restrict the objects it accepts in any given instance? With generics, VB has the perfect solution.*

## Build Typesafe Generic Classes

Programmers often face a difficult choice. On one hand, it's keenly important to build solutions that are as generic as possible, so that they can be reused in different scenarios. For example, why build a `CustomerCollection` class that accepts only objects of type `Customer` when you can build a generic `Collection` class that can be configured to accept objects of any type? On the other hand, performance and type safety considerations can make a generic solution less desirable. If you use a generic .NET `Collection` class to store `Customer` objects, for example, how can you be sure that someone won't accidentally insert another type of object into the collection, causing an insidious problem later on?

Visual Basic 2005 and .NET 2.0 provide a solution called *generics*. Generics are classes that are *parameterized by type*. In other words,

generics allow you to create a class template that supports any type. When you instantiate that class, you specify the type you want to use, and from that point on, your object is “locked in” to the type you chose.

## How do I do that?

An example of where the use of generics makes great sense is the `System.Collections.ArrayList` class. `ArrayList` is an all-purpose, dynamically self-sizing collection. It can hold ordinary .NET objects or your own custom objects. In order to support this, `ArrayList` treats everything as the base `Object` type.

The problem is that there’s no way to impose any restrictions on how `ArrayList` works. For example, if you want to use `ArrayList` to store a collection of `Customer` objects, you have no way to be sure that a faulty piece of code won’t accidentally insert strings, integers, or some other type of object, causing future headaches. For this reason, developers often create their own strongly typed collection classes—in fact, the .NET class library is filled with dozens of them.

Generics can solve this problem. For example, using generics you can declare a class that works with any type using the `Of` keyword:

```
Public Class GenericList(Of ItemType)
    ' (Code goes here)
End Class
```

In this case, you are creating a new class named `GenericList` that can work with any type of object. However, the client needs to specify what type should be used. In your class code, you refer to that type as `ItemType`. Of course, `ItemType` isn’t really a type—it’s just a placeholder for the type that you’ll choose when you instantiate a `GenericList` object.

Example 2-2 shows the complete code for a simple typesafe `ArrayList`.

### Example 2-2. A typesafe collection using generics

```
Public Class GenericList(Of ItemType)
    Inherits CollectionBase

    Public Function Add(ByVal value As ItemType) As Integer
        Return List.Add(value)
    End Function

    Public Sub Remove(ByVal value As ItemType)
        List.Remove(value)
    End Sub
```

### Example 2-2. A typesafe collection using generics (continued)

```
Public ReadOnly Property Item(ByVal index As Integer) As ItemType
    Get
        ' The appropriate item is retrieved from the List object and
        ' explicitly cast to the appropriate type, and then returned.
        Return CType(List.Item(index), ItemType)
    End Get
End Property
End Class
```

The `GenericList` class wraps an ordinary `ArrayList`, which is provided through the `List` property of the `CollectionBase` class it inherits from. However, the `GenericList` class works differently than an `ArrayList` by providing strongly typed `Add()` and `Remove()` methods, which use the `ItemType` placeholder.

Here's an example of how you might use the `GenericList` class to create an `ArrayList` collection that only supports strings:

```
' Create the GenericList instance, and choose a type (in this case, string).
Dim List As New GenericList(Of String)

' Add two strings.
List.Add("blue")
List.Add("green")

' The next statement will fail because it has the wrong type.
' There is no automatic way to convert a GUID to a string.
' In fact, this line won't ever run, because the compiler
' notices the problem and refuses to build the application.
List.Add(Guid.NewGuid())
```

There's no limit to how many ways you can parameterize a class. In the `GenericList` example, there's only one type parameter. However, you could easily create a class that works with two or three types of objects, and allows you to make all of these types generic. To use this approach, just separate each parameter type with a comma (between the brackets at the beginning of a class).

For example, consider the following `GenericHashTable` class, which allows you to define the type of the items the collection will store (`ItemType`), as well as the type of the keys you will use to index those items (`KeyType`):

```
Public Class GenericHashTable(Of ItemType, KeyType)
    Inherits DictionaryBase
    ' (Code goes here.)
End Class
```

Another important feature in generics is the ability to apply *constraints* to parameters. Constraints restrict the types allowed for a given generic class. For example, suppose you want to create a class that supports only types that implement a particular interface. To do so, first declare the type or types the class accepts and then use the `As` keyword to specify the base class that the type must derive from, or the interface that the type must implement.

Here's an example that restricts the items stored in a `GenericList` to serializable items. This feature would be useful if, for example, you wanted to add a method to the `GenericList` that required serialization, such as a method that writes all the items in the list to a stream:

```
Public Class SerializableList(Of ItemType As ISerializable)
    Inherits CollectionBase
    ' (Code goes here.)
End Class
```

Similarly, here's a collection that can contain any type of object, provided it's derived from the `System.Windows.Forms.Control` class. The end result is a collection that's limited to controls, like the one exposed by the `Forms.Controls` property on a window:

```
Public Class ControlCollection(Of ItemType As Control)
    Inherits CollectionBase
    ' (Code goes here.)
End Class
```

Sometimes, your generic class might need the ability to create the parameter class. For example, the `GenericList` example might need the ability to create an instance of the item you want to store in the collection. In this case, you need to use the `New` constraint. The `New` constraint allows only parameter types that have a public zero-argument constructor, and aren't marked `MustInherit`. This ensures that your code can create instances of the parameter type. Here's a collection that imposes the `New` constraint:

```
Public Class GenericList(Of ItemType As New)
    Inherits CollectionBase
    ' (Code goes here.)
End Class
```

It's also worth noting that you can define as many constraints as you want, as long as you group the list of constraints in curly braces, as shown here:

```
Public Class GenericList(Of ItemType As {ISerializable, New})
    Inherits CollectionBase
    ' (Code goes here.)
End Class
```

*Generics are built into the Common Language Runtime. That means they are supported in all first-class .NET languages, including C#.*

Constraints are enforced by the compiler, so if you violate a constraint rule when using a generic class, you won't be able to compile your application.

## What about...

...using generics with other code structures? Generics don't just work with classes. They can also be used in structures, interfaces, delegates, and even methods. For more information, look for the index entry "generics" in the MSDN Help. For more in-depth examples of advanced generic techniques, you can refer to a Microsoft whitepaper at [http://www.msdn.net/library/en-us/dnvs05/html/vb2005\\_generics.asp](http://www.msdn.net/library/en-us/dnvs05/html/vb2005_generics.asp).

Incidentally, the .NET Framework designers are well aware of the usefulness of generic collections, and they've already created several for you to use out of the box. You'll find them in the new `System.Collections.Generic` namespace. They include:

- `List` (a basic collection like the `GenericList` example)
- `Dictionary` (a name-value collection that indexes each item with a key)
- `LinkedList` (a linked list, where each item points to the next item in the chain)
- `Queue` (a first-in-first-out collection)
- `Stack` (a last-in-first-out collection)
- `SortedList` (a name-value collection that's kept in perpetually sorted order)

Most of these types duplicate one of the types in the `System.Collections` namespace. The old collections remain for backward compatibility.

*Do you need to represent data that may or may not be present? VB .NET's new nullable types fill the gap.*

## Make Simple Data Types Nullable

With the new support for generics that's found in the .NET Framework, a number of new features become possible. One of these features—generic strongly typed collections—was demonstrated in the previous lab, "Build Typesafe Generic Classes." Now you'll see another way that generics can solve common problems, this time by using the new nullable data types.

## How do I do that?

A null value (identified in Visual Basic by the keyword `Nothing`), is a special flag that indicates no data is present. Most developers are familiar with null object references, which indicate that the object has been defined but not created. For example, in the following code, the `FileStream` contains a null reference because it hasn't been instantiated with the `New` keyword:

```
Dim fs As FileStream
If fs Is Nothing
    ' This is always true because the FileStream hasn't
    ' been created yet.
    Console.WriteLine("Object contains a null reference.")
End If
```

Core data types like integers and strings *can't* contain null values. Numeric variables are automatically initialized to 0. Boolean variables are `False`. String variables are set to an empty string (""), automatically. In fact, even if you explicitly set a simple data type variable to `Nothing` in your code, it will automatically revert to the empty value (0, `False`, or ""), as the following code demonstrates:

```
Dim j As Integer = Nothing
If j = 0 Then
    ' This is always true because there is an
    ' implicit conversion between Nothing and 0 for integers.
    Console.WriteLine("Non-nullable integer j = " & j)
End If
```

This design sometimes causes problems, because there's no way to distinguish between an empty value and a value that was never supplied in the first place. For example, imagine you create code that needs to retrieve the number of times the user has placed an order from a text file. Later on, you examine this value. The problem occurs if this value is 0. Quite simply, you have no way to know whether this is valid data (the user placed no orders), or it represents missing information (the setting couldn't be retrieved or the current user isn't a registered customer).

Thanks to generics, .NET 2.0 has a solution—a `System.Nullable` class that can wrap any other data type. When you create an instance of `Nullable` you specify the data type. If you don't set a value, this instance contains a null reference. You can test whether this is true by testing the `Nullable.HasValue()` method, and you can retrieve the underlying object through the `Nullable.Value` property.

Here's the sample code you need to create a nullable integer:

```
Dim i As Nullable(Of Integer)
If Not i.HasValue Then
    ' This is true, because no value has been assigned.
    Console.WriteLine("i is a null value")
End If

' Assign a value. Note that you must assign directly to i, not i.Value.
' The i.Value property is read-only, and it always reflects the
' currently assigned object, if it is not Nothing.
i = 100
If i.HasValue Then
    ' This is true, because a value (100) is now present.
    Console.WriteLine("Nullable integer i = " & i.Value)
End If
```

## What about...

...using `Nullable` with full-fledged reference objects? Although you don't need this ability (because reference types can contain a null reference), it still gives you some advantages. Namely, you can use the slightly more readable `HasValue()` method instead of testing for `Nothing`. Best of all, you can make this change seamlessly, because the `Nullable` class has the remarkable ability to allow implicit conversions between `Nullable` and the type it wraps.

## Where can I learn more?

To learn more about `Nullable` and how it's implemented, look up the "Nullable class" index entry in the MSDN Help.

*Tired of using clumsy syntax like `ObjA.Subtract(ObjB)` to perform simple operations on your custom objects? With VB's support for operator overloading, you can manipulate your objects as easily as ordinary numbers.*

## Use Operators with Custom Objects

Every VB programmer is familiar with the arithmetic operators for addition (+), subtraction (-), division (/), and multiplication (\*). Ordinarily, these operators are reserved for .NET numeric types, and have no meaning when used with other objects. However, in VB .NET 2.0 you can build objects that support all of these operators, as well as the operators used for logical operations and implicit conversion). This technique won't make sense for business objects, but it is extremely handy if you need to model mathematical structures such as vectors, matrixes, complex numbers, or—as demonstrated in the following example—fractions.

## How do I do that?

To overload an operator in Visual Basic 2005, you need to create a special operator method in your class (or structure). This method must be declared with the keywords `Public Shared Operator`, followed by the symbol for the operator (e.g., `+`).

---

### TIP

To *overload* an operator simply means to define what an operator does when used with a specific type of object. In other words, when you overload the `+` operator for a `Fraction` class, you tell .NET what to do when your code adds two `Fraction` objects together.

---

For example, here's an operator method that adds support for the addition (`+`) operator:

```
Public Shared Operator+(objA As MyClass, objB As MyClass) As MyClass
    ' (Code goes here.)
End Operator
```

Every operator method accepts two parameters, which represent the values on either side of the operator. Depending on the class and the operator, order may be important (as it is for division).

Once you've defined an operator, the VB compiler will call your code when it executes a statement that uses the operator with your class. For example, the compiler changes code like this:

```
ObjC = ObjA + ObjB
```

into this:

```
ObjC = MyClass.Operator+(ObjA, ObjB)
```

Example 2-3 shows how you can overload the Visual Basic arithmetic operators used to handle `Fraction` objects. A `Fraction` consists of two portions: a numerator and a denominator (known colloquially as "the top part and the bottom part"). The `Fraction` code overloads the `+`, `-`, `*`, and `/` operators, allowing you to perform fractional calculations without converting your numbers to decimals and losing precision.

### Example 2-3. Overloading arithmetic operators in the `Fraction` class

```
Public Structure Fraction
```

```
    ' The two parts of a fraction.
    Public Denominator As Integer
    Public Numerator As Integer
```

```
    Public Sub New(ByVal numerator As Integer, ByVal denominator As Integer)
```

### Example 2-3. Overloading arithmetic operators in the Fraction class (continued)

```
Me.Numerator = numerator
Me.Denominator = denominator
End Sub

Public Shared Operator +(ByVal x As Fraction, ByVal y As Fraction) _
As Fraction
Return Normalize(x.Numerator * y.Denominator + _
y.Numerator * x.Denominator, x.Denominator * y.Denominator)
End Operator

Public Shared Operator -(ByVal x As Fraction, ByVal y As Fraction) _
As Fraction
Return Normalize(x.Numerator * y.Denominator - _
y.Numerator * x.Denominator, x.Denominator * y.Denominator)
End Operator

Public Shared Operator *(ByVal x As Fraction, ByVal y As Fraction) _
As Fraction
Return Normalize(x.Numerator * y.Numerator, _
x.Denominator * y.Denominator)
End Operator

Public Shared Operator /(ByVal x As Fraction, ByVal y As Fraction) _
As Fraction
Return Normalize(x.Numerator * y.Denominator, _
x.Denominator * y.Numerator)
End Operator

' Reduce a fraction.
Private Shared Function Normalize(ByVal numerator As Integer, _
ByVal denominator As Integer) As Fraction
If (numerator <> 0) And (denominator <> 0) Then
' Fix signs.
If denominator < 0 Then
denominator *= -1
numerator *= -1
End If

Dim divisor As Integer = GCD(numerator, denominator)
numerator \= divisor
denominator \= divisor
End If

Return New Fraction(numerator, denominator)
End Function

' Return the greatest common divisor using Euclid's algorithm.
Private Shared Function GCD(ByVal x As Integer, ByVal y As Integer) _
As Integer
Dim temp As Integer

x = Math.Abs(x)
y = Math.Abs(y)
```

### Example 2-3. Overloading arithmetic operators in the Fraction class (continued)

```
Do While (y <> 0)
    temp = x Mod y
    x = y
    y = temp
Loop

Return x
End Function

' Convert the fraction to decimal form.
Public Function GetDouble() As Double
    Return CType(Me.Numerator, Double) / _
        CType(Me.Denominator, Double)
End Function

' Get a string representation of the fraction.
Public Overrides Function ToString() As String
    Return Me.Numerator.ToString & "/" & Me.Denominator.ToString
End Function

End Structure
```

The console code shown in Example 2-4 puts the fraction class through a quirk-and-dirty test. Thanks to operator overloading, the number remains in fractional form, and precision is never lost.

### Example 2-4. Testing the Fraction class

```
Module FractionTest

Sub Main()
    Dim f1 As New Fraction(2, 3)
    Dim f2 As New Fraction(1, 4)

    Console.WriteLine("f1 = " & f1.ToString())
    Console.WriteLine("f2 = " & f2.ToString())

    Dim f3 As Fraction
    f3 = f1 + f2      ' f3 is now 11/12
    Console.WriteLine("f1 + f2 = " & f3.ToString())

    f3 = f1 / f2     ' f3 is now 8/3
    Console.WriteLine("f1 / f2 = " & f3.ToString())

    f3 = f1 - f2     ' f3 is now 5/12
    Console.WriteLine("f1 - f2 = " & f3.ToString())

    f3 = f1 * f2     ' f3 is now 1/6
    Console.WriteLine("f1 * f2 = " & f3.ToString())
End Sub

End Module
```

When you run this application, here's the output you'll see:

```
f1 = 2/3
f2 = 1/4
f1 + f2 = 11/12
f1 / f2 = 8/3
f1 - f2 = 5/12
f1 * f2 = 1/6
```

Usually, the parameters and the return value of an operator method use the same type. However, there's no reason you can't create more than one version of an operator method so your object can be used in expressions with different types.

## What about...

...using operator overloading with other types? There are a number of classes that are natural candidates for operator overloading. Here are some good examples:

- Mathematical classes that model vectors, matrixes, complex numbers, or tensors.
- Money classes that round calculations to the nearest penny, and support different currency types.
- Measurement classes that have irregular units, like inches and feet.

## Where can I learn more?

For more of the language details behind operator overloading and all the operators that you can overload, refer to the "Operator procedures" index entry in the MSDN Help.

*Have your classes grown too large to manage in one file? With the new `Partial` keyword, you can split a class into separate files.*

## Split a Class into Multiple Files

If you've cracked open a .NET 2.0 Windows Forms class, you'll have noticed that all the automatically generated code is missing! To understand where it's gone, you need to learn about a new feature called *partial classes*, which allow you to split classes into several pieces.

## How do I do that?

Using the new `Partial` keyword, you can split a single class into as many pieces as you want. You simply define the same class in more than one place. Here's an example that defines a class named `SampleClass` in two pieces:

```
Partial Public Class SampleClass
    Public Sub MethodA()
        Console.WriteLine("Method A called.")
    End Sub
End Class
```

```
Partial Public Class SampleClass
    Public Sub MethodB()
        Console.WriteLine("Method B called.")
    End Sub
End Class
```

In this example, the two declarations are in the same file, one after the other. However, there's no reason that you can't put the two `SampleClass` pieces in different source code files in the same project. (The only restrictions are that you can't define the two pieces in separate assemblies or in separate namespaces.)

When you build the application containing the previous code, Visual Studio will track down each piece of `SampleClass` and assemble it into a complete, compiled class with two methods, `MethodA()` and `MethodB()`. You can use both methods, as shown here:

```
Dim Obj As New SampleClass()
Obj.MethodA()
Obj.MethodB()
```

Partial classes don't offer you much help in solving programming problems, but they can be useful in breaking up extremely large, unwieldy classes. Of course, the existence of large classes in your application could be a sign that you haven't properly factored your problem, in which case you should really break your class down into separate, not partial, classes. One of the key roles of partial classes in .NET is to hide the designer code that is automatically generated by Visual Studio, whose visibility in previous versions has been a source of annoyance to some VB programmers.

For example, when you build a .NET Windows form in Visual Basic 2005, your event handling code is placed in the source code file for the form, but the designer code that creates and configures each control and connects its event handlers is nowhere to be seen. In order to see this code, you need to select `Project → Show All Files` from the Visual Studio menu. When you do, the file that contains the missing half of the class appears in the Solution Explorer as a separate file. Given a form named `Form1`, you'll actually wind up with a `Form1.vb` file that contains your code and a `Form1.Designer.vb` file that contains the automatically generated part.

## What about...

...using the `Partial` keyword with structures? That works, but you can't create partial interfaces, enumerations, or any other .NET programming construct.

## Where can I learn more?

To get more details on partial classes, refer to the index entry "Partial keyword" in the MSDN Help.

*Do you use the My objects so much you'd like to customize them yourself? VB 7.005 lets you plug in your own classes.*

## Extend the My Namespace

The My objects aren't defined in a single place. Some come from classes defined in the `Microsoft.VisualBasic.MyServices` namespace, while others are generated dynamically as you add forms, web services, configuration settings, and embedded resources to your project. However, as a developer you can participate in the My namespace and extend it with your own ingredients (e.g., useful calculations and tasks that are specific to your application).

## How do I do that?

To plug a new class into the My object hierarchy, simply use a `Namespace` block with the name `My`. For example, you could add this code to create a new `BusinessFunctions` class that contains a company-specific function for generating custom identifiers (by joining the customer name to a new GUID):

```
Namespace My

    Public Class BusinessFunctions
        Public Shared Function GenerateNewCustomerID( _
            ByVal name As String) As String
            Return name & "_" & Guid.NewGuid.ToString()
        End Function
    End Class

End Namespace
```

Once you've created the `BusinessFunctions` object in the right place, you can make use of it in your application just like any other My object. For example, to display a new customer ID:

```
Console.WriteLine(My.BusinessFunctions.GenerateNewCustomerID("matthew"))
```

Note that the My classes you add need to use shared methods and properties. That's because the My object won't be instantiated automatically. As a result, if you use ordinary instance members, you'll need to create the My object on your own, and you won't be able to manipulate it with the same syntax. Another solution is to create a module in the My namespace, because all the methods and properties in a module are always shared.

You can also extend some of the existing My objects thanks to partial classes. For example, using this feature you could add new information to the My.Computer object or new routines to the My.Application object. In this case, the approach is slightly different. My.Computer exposes an instance of the MyComputer object. My.Application exposes an instance of the MyApplication object. Thus, to add to either of these classes, you need to create a partial class with the appropriate name, and add the instance members you need. You should also declare this class with the accessibility keyword Friend in order to match the existing class.

Here's an example you can use to extend My.Application with a method that checks for update versions:

```
Namespace My
    Partial Friend Class MyApplication
        Public Function IsNewVersionAvailable() As Boolean
            ' Usually, you would read the latest available version number
            ' from a web service or some other resource.
            ' Here, it's hardcoded.
            Dim LatestVersion As New Version(1, 2, 1, 1)
            Return Application.Info.Version.CompareTo(LatestVersion)
        End Function
    End Class
End Namespace
```

And now you can use this method:

```
If My.Application.IsNewVersionAvailable()
    Console.WriteLine("A newer version is available.")
Else
    Console.WriteLine("This is the latest version.")
End If
```

## What about...

...using your My extensions in multiple applications? There's no reason you can't treat My classes in the same way that you treat any other useful class that you want to reuse in multiple applications. In other words,

*Shared members are members that are always available through the class name, even if you haven't created an object. If you use shared variables, there will be one copy of that variable, which is global to your whole application.*

you can create a class library project, add some My extensions, and compile it to a DLL. You can then reference that DLL in other applications.

Of course, despite what Microsoft enthusiasts may tell you, extending the My namespace in that way has two potentially dangerous drawbacks:

- It becomes more awkward to share your component with other languages. For example, C# does not provide a My feature. Although you could still use a custom My object in a C# application, it wouldn't plug in as neatly.
- When you use the My namespace, you circumvent one of the great benefits of namespaces—avoiding naming conflicts. For example, consider two companies who create components for logging. If you use the recommended .NET namespace standard (CompanyName.ApplicationName.ClassName), there's little chance these two components will have the same fully qualified names. One might be Acme.SuperLogger.Logger while the other is ComponentTech.LogMagic.Logger. However, if they both extend a My object, it's quite possible that they would both use the same name (like My.Application.Logger). As a result, you wouldn't be able to use both of them in the same application.

*VB's new Continue keyword gives you a quick way to step out of a tangled block of code in a loop and head straight into the next iteration.*

## Skip to the Next Iteration of a Loop

The Visual Basic language provides a handful of common *flow control* statements, which let you direct the execution of your code. For example, you can use Return to step out of a function, or Exit to back out of a loop. However, before VB 2005, there wasn't any way to skip to the next iteration of a loop.

### How do I do that?

The Continue statement is one of those language details that seems like a minor frill at first, but quickly proves itself to be a major convenience. The Continue statement exists in three versions: Continue For, Continue Do, and Continue While, each of which is used with a different type of loop (For ... Next, Do ... Loop, or While ... End While).

To see how the Continue statement works consider the following code:

```
For i = 1 to 1000
    If i Mod 5 = 0 Then
        ' (Task A code.)
```

```

        Continue For
    End If
    ' (Task B code.)
Next

```

This code loops 1,000 times, incrementing a counter *i*. Whenever *i* is divisible by five, the task A code executes. Then, the Continue For statement is executed, the counter is incremented, and execution resumes at the beginning of the loop, skipping the code in task B.

In this example, the continue statement isn't really required, because you could rewrite the code easily enough as follows:

```

For i = 1 to 1000
    If i Mod 5 = 0 Then
        ' (Task A code.)
    Else
        ' (Task B code.)
    End If
Next

```

However, this isn't nearly as possible if you need to perform several different tests. To see the real benefit of the Continue statement, you need to consider a more complex (and realistic) example.

Example 2-5 demonstrates a loop that scans through an array of words. Each word is analyzed, and the program decides whether the word is made up of letters, numeric characters, or the space character. If the program matches one test (for example, the letter test), it needs to continue to the next word without performing the next test. To accomplish this without using the Continue statement, you need to use nested loops, an approach that creates awkward code.

#### **Example 2-5.** Analyzing a string without using the Continue statement

```

' Define a sentence.
Dim Sentence As String = "The final number is 433."

' Split the sentence into an array of words.
Dim Delimiters() As Char = {" ", ".", ",", ""}
Dim Words() As String = Sentence.Split(Delimiters)

' Examine each word.
For Each Word As String In Words
    ' Check if the word is blank.
    If Word <> "" Then
        Console.WriteLine("'" + Word + "'" & vbTab & "= ")

        ' Check if the word is made up of letters.
        Dim AllLetters As Boolean = True
        For Each Character As Char In Word
            If Not Char.IsLetter(Character) Then

```

**Example 2-5.** Analyzing a string without using the Continue statement (continued)

```
        AllLetters = False
    End If
Next
If AllLetters Then
    Console.WriteLine("word")
Else
    ' If the word isn't made up of letters,
    ' check if the word is made up of numbers.
    Dim AllNumbers As Boolean = True
    For Each Character As Char In Word
        If Not Char.IsDigit(Character) Then
            AllNumbers = False
        End If
    Next
    If AllNumbers Then
        Console.WriteLine("number")
    Else
        ' If the word isn't made up of letters or numbers,
        ' assume it's something else.
        Console.WriteLine("mixed")
    End If
End If
End If
Next
```

Now, consider the rewritten version shown in Example 2-6 that uses the Continue statement to clarify what's going on.

**Example 2-6.** Analyzing a string using the Continue statement

```
' Examine each word.
For Each Word As String In Words
    ' Check if the word is blank.
    If Word = "" Then Continue For
    Console.Write("'" + Word + "'" & vbTab & "= ")

    ' Check if the word is made up of letters.
    Dim AllLetters As Boolean = True
    For Each Character As Char In Word
        If Not Char.IsLetter(Character) Then
            AllLetters = False
        End If
    Next
    If AllLetters Then
        Console.WriteLine("word")
        Continue For
    End If

    ' If the word isn't made up of letters,
    ' check if the word is made up of numbers.
    Dim AllNumbers As Boolean = True
    For Each Character As Char In Word
```

### Example 2-6. Analyzing a string using the Continue statement (continued)

```
    If Not Char.IsDigit(Character) Then
        AllNumbers = False
    End If
Next
If AllNumbers Then
    Console.WriteLine("number")
    Continue For
End If

' If the word isn't made up of letters or numbers,
' assume it's something else.
Console.WriteLine("mixed")
Next
```

## What about...

...using Continue in a nested loop? It's possible. If you nest a For loop inside a Do loop, you can use Continue For to skip to the next iteration of the inner loop, or Continue Do to skip to the next iteration of the outer loop. This technique also works in reverse (with a Do loop inside a For loop), but it doesn't work if you nest a loop inside another loop of the same type. In this case, there's no unambiguous way to refer to the outer loop, and so your Continue statement always refers to the inner loop.

## Where can I learn more?

For the language lowdown on Continue, refer to the index entry "continue statement" in the MSDN Help.

## Dispose of Objects Automatically

In .NET, it's keenly important to make sure objects that use unmanaged resources (e.g., file handles, database connections, and graphics contexts) release these resources as soon as possible. Toward this end, such objects should always implement the IDisposable interface, and provide a Dispose() method that you can call to release their resources immediately.

The only problem with this technique is that you must always remember to call the Dispose() method (or another method that calls Dispose(), such as a Close() method). VB 2005 provides a new safeguard you can apply to make sure Dispose() is always called: the Using statement.

*Worried that you'll have objects floating around in memory, tying up resources until the garbage collector tracks them down? With the Using statement, you can make sure disposable objects meet with a timely demise.*

## How do I do that?

You use the `Using` statement in a block structure. In the first line, when you declare the `Using` block, you specify the disposable object you are using. Often, you'll also create the object at the same time using the `New` keyword. Then, you write the code that uses the disposable object inside the `Using` block. Here's an example with a snippet of code that creates a new file and writes some data to the file:

```
Using NewFile As New System.IO.StreamWriter("c:\MyFile.txt")
    NewFile.WriteLine("This is line 1")
    NewFile.WriteLine("This is line 2")
End Using
```

```
' The file is closed automatically.
' The NewFile object is no longer available here.
```

In this example, as soon as the execution leaves the `Using` block, the `Dispose()` method is called on the `NewFile` object, releasing the file handle.

## What about...

...errors that occur inside a `Using` block? Thankfully, .NET makes sure it disposes of the resource no matter how you exit the `Using` block, even if an unhandled exception occurs.

The `Using` statement makes sense with all kinds of disposable objects, such as:

- Files (including `FileStream`, `StreamReader`, and `StreamWriter`)
- Database connections (including `SqlConnection`, `OracleConnection`, and `OleDbConnection`)
- Network connections (including `TcpClient`, `UdpClient`, `NetworkStream`, `FtpWebResponse`, `HttpWebResponse`)
- Graphics (including `Image`, `Bitmap`, `Metafile`, `Graphics`)

## Where can I learn more?

For the language lowdown, refer to the index entry "Using block" in the MSDN Help.

# Safeguard Properties with Split Accessibility

Most properties consist of a *property get procedure* (which allows you to retrieve the property value) and a *property set procedure* (which allows you to set a new value for the property). In previous versions of Visual Basic, the declared access level of both procedures needed to be the same. In VB 2005, you can protect a property by assigning to the set procedure a lower access level than you give to the get procedure.

*In the past, there was no way to create a property that everyone could read but only your application could update. VB 2005 finally loosens the rules and gives you more flexibility.*

## How do I do that?

VB recognizes three levels of accessibility. Arranged from most to least permissive, these are:

- Public (available to all classes in all assemblies)
- Friend (available to all code in all the classes in the current assembly)
- Private (only available to code in the same class)

Imagine you are creating a DLL component that's going to be used by another application. You might decide to create a property called `Status` that the client application needs to read, and so you declare the property `Public`:

```
Public Class ComponetClass

    Private _Status As Integer
    Public Property Status() As Integer
        Get
            Return _Status
        End Get
        Set(ByVal value As Integer)
            _Status = value
        End Set
    End Property

End Class
```

The problem here is that the access level assigned to the `Status` property allows the client to change it, which doesn't make sense. You could make `Status` a read-only property (in other words, omit the property set procedure altogether), but that wouldn't allow other classes that are part of your applications and located in your component assembly to change it.

The solution is to give the property set procedure the `Friend` accessibility level. Here's what the code should look like, with the only change highlighted:

```
Public Property Status() As Integer
    Get
        Return _Status
    End Get
    Friend Set(ByVal value As Integer)
        _Status = value
    End Set
End Property
```

## What about...

...read-only and write-only properties? Split accessibility doesn't help you if you need to make a read-only property (such as a calculated value) or a write-only value (such as a password that shouldn't remain accessible). To create a read-only property, add the `ReadOnly` keyword to the property declaration (right after the accessibility keyword), and remove the property set procedure. To create a write-only property, remove the property get procedure and add the `WriteOnly` keyword. These keywords are nothing new—they've been available since Visual Basic .NET 1.0.

*With short-circuiting, you can combine multiple conditions to write more compact code.*

## Evaluate Conditions Separately with Short-Circuit Logic

In previous versions of VB, there were two logical operators: `And` and `Or`. Visual Basic 2005 introduces two new operators that supplement these: `AndAlso` and `OrElse`. These operators work in the same way as `And` and `Or`, except they have support for short-circuiting, which allows you to evaluate just one part of a long conditional statement.

## How do I do that?

A common programming scenario is the need to evaluate several conditions in a row. Often, this involves checking that an object is not null, and then examining one of its properties. In order to handle this scenario, you need to use nested `If` blocks, as shown here:

```
If MyObject Is Nothing Then
    If MyObject.Value > 10 Then
        ' (Do something.)
    End If
End If
```

It would be nice to combine both of these conditions into a single line, as follows:

```
If MyObject Is Nothing And MyObject.Value > 10 Then
    ' (Do something.)
End If
```

Unfortunately, this won't work because VB always evaluates both conditions. In other words, even if `MyObject` is `Nothing`, VB will evaluate the second condition and attempt to retrieve the `MyObject.Value` property, which will cause a `NullReferenceException`.

Visual Basic 2005 solves this problem with the `AndAlso` and `OrElse` keywords. When you use these keywords, Visual Basic won't evaluate the second condition if the first condition is false. Here's the corrected code:

```
If MyObject Is Nothing AndAlso MyObject.Value > 10 Then
    ' (Do something.)
End If
```

## What about...

...other language refinements? In this chapter, you've had a tour of the most important VB language innovations. However, it's worth pointing out a few of the less significant ones that I haven't included in this chapter:

- The `IsNot` keyword allows you to simplify awkward syntax slightly. Using it, you can replace syntax like `If Not x Is Nothing` with the equivalent statement `If x IsNot Nothing`.
- The `TryCast()` function allows you to shave a few milliseconds off type casting code. It works like `CType()` or `DirectCast()`, with one exception—if the object can't be converted to the requested type a null reference is returned instead. Thus, instead of checking an object's type and then casting it, you can use `TryCast()` right away and then check if you have an actual object instance.
- Unsigned integers allow you to store numeric values that can't be negative. That restriction saves on memory storage, allowing you to accommodate larger numbers. Unsigned numbers have always been in the .NET Framework, but now VB 2005 includes keywords for them (`UInteger`, `ULong`, and `UShort`).