Beginning ASP.NET 2.0 in VB 2005

From Novice to Professional

Matthew MacDonald

Apress®

Beginning ASP.NET 2.0 in VB 2005: From Novice to Professional

Copyright © 2006 by Matthew MacDonald

ISBN-13 (pbk): 978-1-59059-621-0

ISBN-10 (pbk): 1-59059-621-8

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

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

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 3

Types, Objects, and Namespaces

Object-oriented programming has been a popular buzzword over the last several years. In fact, one of the few places that object-oriented programming *wasn't* emphasized was in ordinary ASP pages. With .NET, the story changes considerably. Not only does .NET allow you to use objects, it demands it. Almost every ingredient you'll need to use to create a web application is, on some level, really a kind of object.

So how much do you need to know about object-oriented programming to write web pages in .NET? It depends on whether you want to follow existing examples and cut and paste code samples or have a deeper understanding of the way .NET works and gain more control. This book assumes that if you're willing to pick up a thousand-page book, then you're the type of programmer who excels by understanding how and why things work the way they do. It also assumes you're interested in some of the advanced ASP.NET programming tasks that *will* require class-based design, such as creating your own components (see Chapter 24) and designing custom controls (see Chapter 25).

This chapter explains objects from the point of view of the .NET Framework. It won't rehash the typical object-oriented theory, because countless excellent programming books cover the subject. Instead, you'll see the types of objects .NET allows, how they're constructed, and how they fit into the larger framework of namespaces and assemblies.

The Basics of Classes

As a developer, you've probably already created classes or at least heard about them. *Classes* are the code definitions for objects. The nice thing about a class is that you can use it to create as many objects as you need. For example, you might have a class that represents an XML file, which can be used to read some data. If you want to access multiple XML files at once, you can create several instances of your class, as shown in Figure 3-1. These instances are called *objects*.

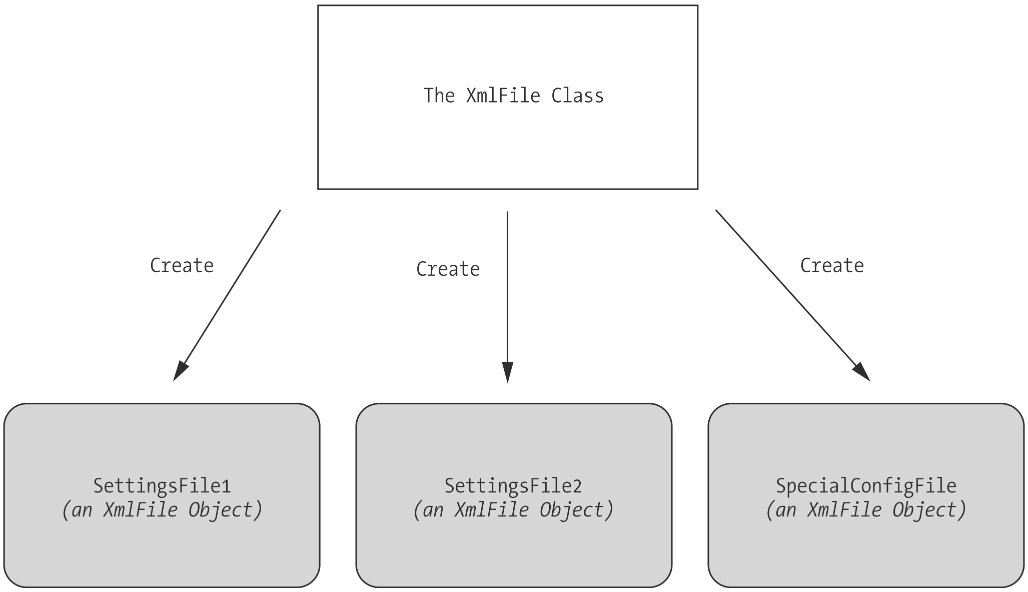


Figure 3-1. *Classes are used to create objects.*

Classes interact with each other with the help of three key ingredients:

- *Properties*: Properties allow you to access an object's data. Some properties may be read-only, so they cannot be modified, while others can be changed. For example, the previous chapter demonstrated how you can use the read-only Length property of a string object to find out how many characters are in a string.

- *Methods*: Methods allow you to perform an action with an object. Unlike properties, methods are used for actions that perform a distinct task or may change the object's state significantly. For example, to open a connection to a database, you might call the Open() method of a Connection object.

- *Events*: Events provide notification that something has happened. If you've ever programmed an ordinary desktop application in Visual Basic, you know how controls can fire events to trigger your code. For example, if a user clicks a button, the Button object fires a Click event, which your code can react to. ASP.NET controls also provide events.

Classes contain their own code and internal set of private data. Classes behave like *black boxes*, which means that when you use an object, you shouldn't waste any time wondering how it works internally or what low-level information it's using. Instead, you need to worry only about the public interface of a class, which is the set of properties, methods, and events available for you to use. Together, these elements are called class *members*.

In ASP.NET, you'll create your own custom classes to represent individual web pages. In addition, you'll create custom classes if you design separate components. For the most part, however, you'll be using prebuilt classes from the .NET class library, rather than programming your own.

Shared and Instance Members

One of the tricks about .NET classes is that you really use them in two ways. You can use some class members without creating an object first. These are called *shared* members, and they're accessed by class name. For example, you can use the shared property DateTime.Now to retrieve a DateTime object that represents the current date and time. You don't need to create a DateTime object first.

On the other hand, the majority of the DateTime members require a valid instance. For example, you can't use the AddDays() method or the Hour property without a valid object. These *instance* members have no meaning without a live object and some valid data to draw on.

The following code snippet uses shared and instance members:

```
' Get the current date using a shared method.
' Note that you use the type name DateTime.
Dim myDate As DateTime = DateTime.Now

' Use an instance method to add a day.
' Note that you need to use the object name myDate.
myDate = myDate.AddDays(1)

' The following code makes no sense.
' It tries to use the instance method AddDays() with the class name DateTime!
myDate = DateTime.AddDays(1)
```

Both properties and methods can be designated as shared. Shared methods are a major part of the .NET Framework, and you will use them frequently in this book. Remember, some classes may consist entirely of shared members (such as the Math class shown in the previous chapter), and some may use only instance members. Other classes, such as DateTime, provide a combination of the two.

The next example, which introduces a basic class, will use only instance members. This is the most common design and a good starting point.

A Simple Class

To create a class, you must define it in a class block:

```
Public Class MyClass
    ' Class code goes here.
End Class
```

You can define as many classes as you need in the same file. However, good coding practices suggest that in most cases you use a single file for each class.

Classes exist in many forms. They may represent an actual thing in the real world (as they do in most programming textbooks), they may represent some programming abstraction (such as a rectangle or color structure), or they may just be a convenient way to group related functionality (like with the Math class). Deciding what a class should represent and breaking down your code into a group of interrelated classes are part of the art of object-oriented programming.

Building a Basic Class

In the next example, you'll see how to construct a .NET class piece by piece. This class will represent a product from the catalog of an e-commerce company. The Product class will store product data, and it will include the built-in functionality needed to generate a block of HTML that displays the product on a web page. When this class is complete, you'll be able to put it to work with a sample ASP.NET test page.

Once you've defined the basic skeleton for your class, the next step is to add some basic data members. This example defines three member variables that store information about the product—namely, its name, price, and a URL that points to an image file:

```
Public Class Product
    Private name As String
    Private price As Decimal
    Private imageUrl As String
End Class
```

A local variable exists only until the current method ends. On the other hand, a *member variable* (or *field*) is declared as part of a class. It's available to all the methods in the class; it's created when the object is created; and it lives as long as the containing object lives.

When you declare a member variable, you need to explicitly set its *accessibility*. The accessibility determines whether other parts of your code will be able to read and alter this variable. For example, if ObjectA contains a private variable, ObjectB will not be able to read or modify it. Only ObjectA will have that ability. On the other hand, if ObjectA has a public variable, any other object in your application is free to read and alter the information it contains. Local variables don't support any accessibility keywords, because they can never be made available to any code outside of the current method. Generally, in a simple ASP.NET application, most of your member variables will be private because the majority of your code will be self-contained in a single web page class. As you start creating separate components to reuse functionality, however, accessibility becomes much more important. Table 3-1 explains the access levels you can use.

Table 3-1. *Accessibility Keywords*

Keyword	Accessibility
Public	Can be accessed by any class
Private	Can be accessed only by methods inside the current class
Friend	Can be accessed by methods in any of the classes in the current assembly (the compiled code file)
Protected	Can be accessed by methods in the current class or by any class that inherits from this class

The accessibility keywords don't just apply to member variables. They also apply to methods, properties, and events, all of which will be explored in this chapter.

Tip By convention, all the public pieces of your class (the class name, public events, properties and methods, and so on) should use *Pascal case*. This means the name starts with an initial capital. (The function name DoSomething() is one example of Pascal case.) On the other hand, private member variables can use any case you want. Usually, private members will adopt *camel case*. This means the name starts with an initial lowercase letter. (The variable name myInformation is one example of camel case.)

Creating an Object

When creating an object, you need to specify the New keyword. For example, the following code snippet creates a Product object (that is, an instance of the Product class) and then stores a reference to the object in the saleProduct variable:

```
Dim saleProduct As New Product()

' Optionally you could do this in two steps:
' Dim saleProduct As Product
' saleProduct = new Product()
```

If you omit the keyword, you'll declare the variable, but you won't create the object. Here's an example:

```
Dim saleProduct As Product
```

In this case, your saleProduct variable doesn't point to any object at all. (Technically, it's Nothing, which is the VB keyword that represents a null reference.) If you try to use the saleProduct variable, you'll receive the common "null reference" error, because no object exists for you to use. When you're finished using an object, you can release it by removing all references to the object. In the previous code snippet, there is only one variable that points to the Product object—the saleProduct variable. Here's how you release the saleProduct reference:

```
saleProduct = Nothing
```

In .NET, you almost never need to use this code. That's because objects are automatically released when the appropriate variable goes out of scope. For example, if you declare the saleProduct variable inside a method, the object is released once the method ends.

■ **Tip** Just because an object is released doesn't mean the memory it uses is immediately reclaimed. The CLR uses a background task (called the *garbage collection service*) that periodically scans for released objects and reclaims the memory they hold.

In some cases, you will want to declare an object variable without actually creating an object. For example, you might want to call a function that generates an object for you, and then use the object that's returned from the function.

In order to do this, declare your variable without using the New keyword, and then assign the object to your variable. Here's an example:

```
' Declare but don't create the product.
Dim saleProduct As Product

' Call a function that accepts a numeric product ID parameter,
' and returns a Product object.
' Assign the Product object to the saleProduct variable.
saleProduct = FetchProduct(23)
```

Adding Properties

The simple Product class is essentially useless because your code cannot manipulate it. All its information is private and unreachable. Other classes won't be able to set or read this information.

To overcome this limitation, you could make the member variables public. Unfortunately, that approach could lead to problems because it would give other objects free access to change everything, even allowing them to apply invalid or inconsistent data. Instead, you need to add a "control panel" through which your code can manipulate Product objects in a safe way. You can do this by adding *property accessors*.

Accessors usually have two parts. The Get accessor allows your code to retrieve data from the object. The Set accessor allows your code to set the object's data. In some cases, you might omit one of these parts, such as when you want to create a property that can be examined but not modified.

Accessors are similar to any other type of method in that you can write as much code as you need. For example, your Set accessor could raise an error to alert the client code of invalid data and prevent the change from being applied. Or your Set accessor could change multiple private variables at once, thereby making sure the object's internal state remains consistent. In the Product class example, this sophistication isn't required. Instead, the property accessors just provide straightforward access to the private variables.

Here's a revised version of the Product class that makes all its member variables private and adds three properties to provide access to them:

```
Public Class Product
    Private _name As String
    Private _price As Decimal
    Private _imageUrl As String

    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            _name = value
        End Set
    End Property

    Public Property Price() As Decimal
        Get
            Return _price
        End Get
        Set(ByVal value As Decimal)
            _price = value
        End Set
    End Property
    Public Property ImageUrl() As String
        Get
            Return _imageUrl
        End Get
        Set(ByVal value As String)
            _imageUrl = value
        End Set
    End Property
End Class
```

Property accessors, like any other public piece of a class, should start with an initial capital. Usually, the private variable will have a similar name, but prefixed with an underscore (as in the previous code example), or the m_ prefix (which means "member variable"). Although it's technically possible, it's not recommended to use the same name for a property as for a private variable, because it's too easy to make a mistake and refer to one when you want the other.

The client can now create and configure an instance of the class by using its properties and the familiar dot syntax. For example, if the object variable is named saleProduct, you can set the product name using the saleProduct.Name property. Here's an example:

```
Dim saleProduct As New Product()
saleProduct.Name = "Kitchen Garbage"
saleProduct.Price = 49.99D
saleProduct.ImageUrl = "http://mysite/garbage.png"
```

■ **Note** Visual Basic treats all literal decimal values (hard-coded numbers such as 49.99) as the Double data type. In the preceding code, this doesn't cause a problem because Visual Basic is able to seamlessly convert a Double into a Decimal, which is the required data type for the Product.Price property. However, if you've switched on Option Explicit, this implicit conversion isn't allowed, so you need to replace 49.99 with 49.99D. The *D* character at the end of any number tells Visual Basic to interpret the number as a Decimal data type straight off.

Usually, property accessors come in pairs—that is, every property has both a Get and a Set accessor. But this isn't always the case. You can create properties that can be read but not set (which are called read-only properties), and properties that can be set but not retrieved (called write-only). All you need to do is include either the ReadOnly or the WriteOnly keyword in the property declaration, and then leave out whichever part of the property you don't need. Here's an example:

```
Public ReadOnly Property Price() As Decimal
    Get
        Return _price
    End Get
End Property
```

This technique is particularly handy if you want to create properties that don't correspond directly to a private member variable. For example, you might want to use properties that represent calculated values, or properties that are based on other properties.

Adding a Basic Method

The current Product class consists entirely of data. This type of class is often useful in an application. For example, you might use it to send information about a product from one function to another. However, it's more common to add functionality to your classes along with the data. This functionality takes the form of *methods*.

Methods are simply procedures that are built into your class. When a method is called on an object, your code responds to do something useful, such as return some calculated data. In this example, we'll add a GetHtml() method to the Product class. This method will return a string representing a formatted block of HTML based on the current data in the Product object. You could then take this block of HTML and place it on a web page to represent the product:

```
Public Class Product
    ' (Variables and properties omitted for clarity.)

    Public Function GetHtml() As String
        Dim htmlString As String
        htmlString = "<h1>" & Name & "</h1><br />"
        htmlString &= "<h3>Costs: " & Price.ToString() & "</h3><br />"
        htmlString &= "<img src=" & ImageUrl & ">"
        Return htmlString
    End Function
End Class
```

All the GetHtml() method does is read the private data and format it in some attractive way. This really targets the class as a user interface class rather than as a pure data class or "business object."

Adding a Constructor

Currently, the Product class has a problem. Ideally, classes should ensure that instances are always in a valid state. However, unless you explicitly set all the appropriate properties, the Product object won't correspond to a valid product. This could cause an error if you try to use a method that relies on some of the data that hasn't been supplied. To solve this problem, you need to equip your class with one or more *constructors*.

A constructor is a method that automatically runs when an instance is created. In VB, the constructor is always a method with the name New().

The next code example shows a new version of the Product class. It adds a constructor that requires the product price and name as arguments.

```
Public Class Product
    ' (Additional class code omitted for clarity.)

    Public Sub New(ByVal name As String, ByVal price As Decimal)
        _name = name
        _price = price
    End Sub
End Class
```

Here's an example of the code you need to create an object based on the new Product class, using its constructor:

```
Dim saleProduct As New Product("Kitchen Garbage", 49.99D)
```

The preceding code is much leaner than the code that was required to create and initialize an instance of the previous Product class. With the help of the constructor, you can create a Product object and configure it with the basic data it needs in a single line.

If you don't create a constructor, .NET supplies a default public constructor that does nothing. If you create at least one constructor, .NET will not supply a default constructor. Thus, in the preceding example, the Product class has exactly one constructor, which is the one that is explicitly defined in code. To create a Product object, you *must* use this constructor. This restriction prevents a client from creating an object without specifying the bare minimum amount of data that's required:

```
' This will not be allowed, because there is
' no zero-argument constructor.
Dim saleProduct As New Product()
```

Most of the classes you use will have constructors that require parameters. As with ordinary methods, constructors can be overloaded with multiple versions, each providing a different set of parameters. When creating an object, you can choose the constructor that suits you best based on the information that you have available. The .NET Framework classes use overloaded constructors extensively.

Adding a Basic Event

Classes can also use events to notify your code. To define an event in VB .NET, you use the Event keyword, followed by the name of the event, and a list of parameters that the event will use. Once you've defined the event, you can fire it anytime using the RaiseEvent statement.

As an illustration, the Product class example has been enhanced with a NameChanged event that occurs whenever the Name is modified through the property accessor. This event won't fire if code inside the class changes the underlying private name variable without going through the property accessor.

```
Public Class Product
    ' (Additional class code omitted for clarity.)

    ' Define the event.
    Public Event NameChanged()

    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(value As String)
            _name = value

            ' Fire the event to all listeners.
            RaiseEvent NameChanged()
        End Set
    End Property
End Class
```

ASP.NET uses an *event-driven* programming model, so you'll soon become used to writing code that reacts to events. But unless you're creating your own components, you won't need to fire your own custom events. For an example where custom events are useful, refer to Chapter 25, which discusses how you can build your own controls.

Handling an Event

It's quite possible that you'll create dozens of ASP.NET applications without once defining a custom event. However, you'll be hard-pressed to write a single ASP.NET web page without *handling* an event. To handle an event, you first create a method called an *event handler*. The event handler contains the code that should be executed when the event occurs. Then, you connect the event handler to the event.

To handle the NameChanged event, you need to create an event handler. Usually, this event handler will be placed in another class, one that needs to respond to the change. The event handler needs to have the same signature as the event it's handling. In the Product example, the event has no parameters, so the event handler would look like the simple subroutine shown here:

```
Private Sub ChangeDetected()
    ' This code executes in response to the NameChanged event.
End Sub
```

The next step is to hook up the event handler to the event. There are two ways to connect an event handler. The first option is to connect an event handler at runtime using the AddHandler statement. Here's an example:

```
Dim saleProduct As New Product()

' This connects the saleProduct.NameChanged event to an event handling
' method called ChangeDetected.
AddHandler saleProduct.NameChanged, AddressOf ChangeDetected

' Now the event will occur in response to this code:
saleProduct.Name = "Kitchen Garbage"
```

You'll notice that this code is quite similar to the delegate example in the previous chapter. In fact, events use delegates behind the scenes to keep track of the event handlers they need to notify.

■**Tip** You can also detach an event handler using the RemoveHandler statement.

Declarative Event Handling

Instead of connecting events programmatically (as described in the previous section), you can connect them *declaratively*. This approach is often more convenient because it requires less code.

The first step is to declare the event-firing object at the class level. Here's an example that takes this step with the saleProduct variable:

```
Public Class EventTester

    Private saleProduct As New Product()

    Private Sub ChangeDetected()
        ' This code executes in response to the NameChanged event.
    End Sub

    ...
End Class
```

By declaring saleProduct at the class level (rather than inside a method), you make it available for declarative event handling. Note that it isn't necessary to use the New keyword to create the object immediately, as this example does. Instead, you could create the object somewhere else in your code, as long as it's *defined* at the class level.

Once you have this basic design, the next step is to hook up your event handler. In this case, you won't use the AddHandler statement. Instead, you'll use the WithEvents keyword and the Handles clause.

The WithEvents keyword is added to the declaration of the object that raises the event, like so:

```
Private WithEvents saleProduct As New Product()
```

The Handles clause is added to the end the declaration for your event handler. It specifies the event you want to handle:

```
Private Sub ChangeDetected() Handles saleProduct.NameChanged
```

Here's the complete code:

```
Public Class EventTester

    Private WithEvents saleProduct As New Product()

    Private Sub ChangeDetected() Handles saleProduct.NameChanged
        ' This code executes in response to the NameChanged event.
    End Sub

    ...
End Class
```

The difference between the declarative and the programmatic approaches to event handling is just skin deep. When you use WithEvents and Handles, the VB compiler will generate the necessary AddHandler statements to link up your event handler automatically.

This approach is often more convenient than the programmatic approach, but it doesn't give you quite the same flexibility. For example, you won't have the ability to attach and detach event handlers while your program is running, which is useful in some specialized scenarios.

Visual Studio uses declarative event handling, so you'll see this technique in the web form examples in the next few chapters. However, it's worth noting that you don't need to add the WithEvents and Handles keywords yourself. Instead, Visual Studio adds the necessary code to connect all the event handlers you create.

■**Note** In traditional Visual Basic programming, events were connected to event handlers based on the method name. In .NET, this clumsy system is abandoned. Your event handler can have any name you want, and it can even be used to handle more than one event, provided they pass the same type of information in their parameters.

Testing the Product Class

To learn a little more about how the Product class works, it helps to create a simple web page. This web page will create a Product object, get its HTML representation, and then display it in the web page. To try this example, you'll need to use the three files that are provided with the online samples in the Chapter03 directory:

- *Product.vb*: This file contains the code for the Product class. It's in the App_Code subdirectory, which allows ASP.NET to compile it automatically (a trick you'll learn more about in Chapter 5).
- *Garbage.jpg*: This is the image that the Product class will use.
- *Default.aspx*: This file contains the web page code that uses the Product class.

The easiest way to test this example is to use Visual Studio, because it includes an integrated web server. Without Visual Studio, you would need to create a virtual directory for this application using IIS, which is much more awkward.

Here are the steps you need to perform the test:

1. Start Visual Studio.

2. Select File ➤ Open ➤ Web Site from the menu.

3. In the Open Web Site dialog box, browse to the Chapter03 directory, select it, and click Open. This loads your project into Visual Studio.

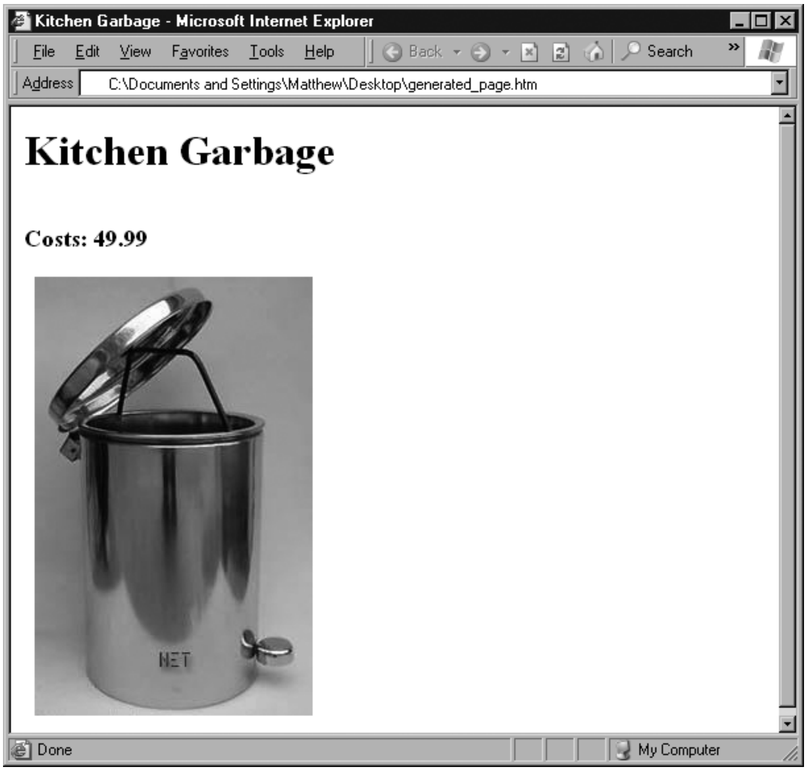
4. Choose Debug ➤ Start Without Debugging to launch the website. Visual Studio will open a new window with your default browser and navigate to the Default.aspx page.

When the Default.aspx page executes, it creates a new Product object, configures it, and uses the GetHtml() method. The HTML is written to the web page using the Response.Write() method. Here's the code:

```
<%@ Page Language="VB" %>
<script runat="server">
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
        Dim saleProduct As New Product("Kitchen Garbage", 49.99D)
        saleProduct.ImageUrl = "garbage.jpg"
        Response.Write(saleProduct.GetHtml())
    End Sub
</script>

<html>
    <head runat="server">
        <title>Product Test</title>
    </head>
</html>
```

The <script> block holds a subroutine named Page_Load. This subroutine is triggered when the page is first created. Once this code is finished, the HTML is sent to the client. Figure 3-2 shows the web page you'll see.


Kitchen Garbage - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search
Address C:\Documents and Settings\Matthew\Desktop\generated_page.htm
Done
My Computer

Kitchen Garbage

Costs: 49.99

Figure 3-2. *Output generated by a Product object*

Interestingly, the GetHtml() method is similar to how an ASP.NET web control works, but on a much cruder level. To use an ASP.NET control, you create an object (explicitly or implicitly) and configure some properties. Then ASP.NET automatically creates a web page by examining all these objects and requesting their associated HTML (by calling a hidden GetHtml() method or by doing something conceptually similar).[1] It then sends the completed page to the user. The end result is that you work with objects, instead of dealing directly with raw HTML code.

When using a web control, you see only the public interface made up of properties, methods, and events. However, understanding how class code actually works will help you master advanced development.

Now that you've seen the basics of classes and a demonstration of how you can use a class, it's time to introduce a little more theory about .NET objects and revisit the basic data types introduced in the previous chapter.

1. Actually, the ASP.NET engine calls a method named Render() in every web control.

Value Types and Reference Types

In Chapter 2, you learned how simple data types such as dates and integers are actually objects created from the class library. This allows some impressive tricks, such as built-in string handling and date calculation. However, simple data types differ from more complex objects in one important way. Simple data types are *value types,* while classes are *reference types.*

This means a variable for a simple data type contains the actual information you put in it (such as the number 7). On the other hand, object variables actually store a reference that points to a location in memory where the full object is stored. In most cases, .NET masks you from this underlying reality, and in many programming tasks you won't notice the difference. However, in three cases you will notice that object variables act a little differently than ordinary data types: in assignment operations, in equality testing, and when passing parameters.

Assignment Operations

When you assign a simple data variable to another simple data variable, the contents of the variable are copied:

```
integerA = integerB   ' integerA now has a copy of the contents of integerB.
                      ' There are two duplicate integers in memory.
```

Reference types work a little differently. Reference types tend to deal with larger amounts of data. Copying the entire contents of a reference type object could slow down an application, particularly if you are performing multiple assignments. For that reason, when you assign a reference type you copy the reference that *points* to the object, not the full object content:

```
' Create a new Product object.
Dim productVariable1 As New Product()

' Declare a second variable.
Dim productVariable2 As Product
productVariable2 = productVariable1
' productVariable1 and productVariable2 now both point to the same thing.
' There is one object and two ways to access it.
```

The consequences of this behavior are far ranging. This example modifies the Product object using productVariable2:

```
productVariable2.Price = 25.99
```

You'll find that productVariable1.Price is set to 25.99. Of course, this only makes sense because productVariable1 and productVariable2 are two variables that point to the same in-memory object.

If you really do want to copy an object (not a reference), you need to create a new object, and then initialize its information to match the first object. Some objects provide a Clone() method that allows you to easily copy the object. One example is the DataSet, which is used to store information from a database.

Equality Testing

A similar distinction between reference types and value types appears when you compare two variables. When you compare value types (such as integers), you're comparing the contents.

```
If integerA = integerB Then
    ' This is true as long as the integers have the same content.
End If
```

When you compare reference type variables, you're actually testing whether they're the same instance. In other words, you're testing whether the references are pointing to the same object in memory, not if their contents match. VB emphasizes this difference by forcing you to use the Is keyword to compare reference types. Using the equals (=) sign will generate a compile-time error.

```
If productVariable1 Is productVariable2 Then
    ' This is True if both productVariable1 and productVariable2
    ' point to the same thing.
    ' This is False if they are separate objects, even if they have
    ' identical content.
End If
```

Passing Parameters by Reference and by Value

You can use two types of method parameters. The standard type is *pass-by-value*. When you use pass-by-value parameters, the method receives a copy of the parameter data. That means that if the method modifies the parameter, this change won't affect the calling code. By default, all parameters are pass-by-value. (Visual Studio also inserts the ByVal keyword automatically to make that fact explicit.)

The second type of parameter is *pass-by-reference*. With pass-by-reference, the method accesses the parameter value directly. If a method changes the value of a pass-by-reference parameter, the original variable is also modified.

To get a better understanding of the difference, consider the following code, which shows a method that uses a parameter named number. This code uses the ByVal keyword to indicate that number should be passed by value:

```
Private Sub ProcessNumber(ByVal number As Integer)
    number *= 2
End Sub
```

Here's how you can call ProcessNumber():

```
Dim num As Integer = 10
ProcessNumber(num)        ' When this call completes, Num will still be 10.
```

Here's what happens. When this code calls ProcessNumber() it passes a copy of the num variable. This copy is multiplied by two. However, the variable in the calling code isn't affected at all.

This behavior changes when you use the ByRef keyword, as shown here:

```
Private Sub ProcessNumber(ByRef number As Integer)
    number *= 2
End Sub
```

Now when the method modifies this parameter (multiplying it by 2), the calling code is also affected:

```
Dim num As Integer = 10
ProcessNumber(num)        ' Once this call completes, Num will be 20.
```

The difference between ByVal and ByRef is straightforward when you're using value types, such as integers. However, if you use reference types, such as a Product object or an array, you won't see this behavior. The reason is because the entire object isn't passed in the parameter. Instead, it's just the *reference* that's transmitted. This is much more efficient for large objects (it saves having to copy a large block of memory), but it doesn't always lead to the behavior you expect.

To understand the difference, consider this method:

```
Private Sub ProcessProduct(ByVal prod As Product)
    prod.Price *= 2
End Sub
```

This code accepts a Product object and increases the price by a factor of 2. Because the Product object is passed by value, you might reasonably expect that the ProcessProduct() method receives a copy of the Product object. However, this isn't the case. Instead, the

ProcessProduct() method gets a copy of the *reference*. However, this new reference still points to the same in-memory Product object. That means that the change shown in this example will affect the calling code.

Reviewing .NET Types

So far, the discussion has focused on simple data types and classes. The .NET class library is actually composed of *types,* which is a catchall term that includes several object-like relatives:

Classes: This is the most common type in .NET Framework. Strings and arrays are two examples of .NET classes, although you can easily create your own.

Structures: Structures, like classes, can include properties, methods, and events. Unlike classes, they are value types, which alters the way they behave with assignment and comparison operations. Structures also lack some of the more advanced class features (such as inheritance) and are generally simpler and smaller. Integers, dates, and characters are all structures.

Enumerations: An enumeration defines a set of integer constants with descriptive names. Enumerations were introduced in the previous chapter.

Delegates: A delegate is a method pointer that allows you to invoke a method indirectly. Delegates are the foundation for .NET event handling and were introduced in the previous chapter.

Interfaces: They define contracts to which a class must adhere. Interfaces are an advanced technique of object-oriented programming, and they're useful when standardizing how objects interact. You'll learn about interfaces with custom control programming in Chapter 25.

WOULD THE REAL REFERENCE TYPES PLEASE STAND UP?

Occasionally, a class can override its behavior to act more like a value type. For example, the String type is a full-featured class, not a simple value type. (This is required to make strings efficient, because they can contain a variable amount of data.) However, the String type overrides its equality and assignment operations so that these operations work like those of a simple value type. This makes the String type work in the way that programmers intuitively expect. Arrays, on the other hand, are reference types through and through. If you assign one array variable to another, you copy the reference, not the array (although the Array class also provides a Clone() method that returns a duplicate array to allow true copying).

The following table sets the record straight and explains a few common types.

Common Reference and Value Types

Data Type	Nature	Behavior
Int32, Decimal, Single, Double, and all other basic numeric types	Value Type	Equality and assignment operations work with the variable contents, not a reference.
DateTime, TimeSpan	Value Type	Equality and assignment operations work with the variable contents, not a reference.
Char, Byte, and Boolean	Value Type	Equality and assignment operations work with the variable contents, not a reference.
String	Reference Type	Equality and assignment operations appear to work with the variable contents, not a reference.
Array	Reference Type	Equality and assignment operations work with the reference, not the contents.

Understanding Namespaces and Assemblies

Whether you realize it at first, every piece of code in .NET exists inside a .NET type (typically a class). In turn, every type exists inside a namespace. Figure 3-3 shows this arrangement for your own code and the DateTime class. Keep in mind that this is an extreme simplification—the System namespace alone is stocked with several hundred classes. This diagram is designed only to show you the layers of organization.

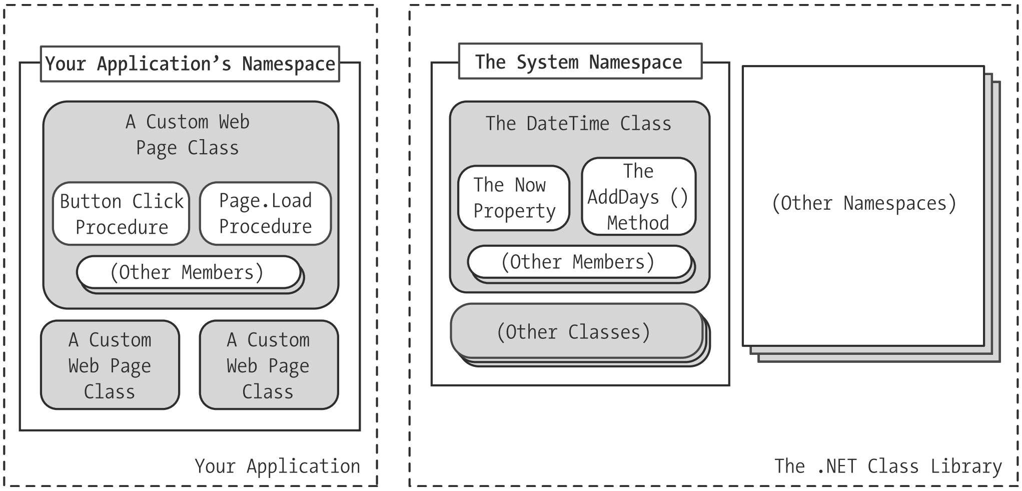
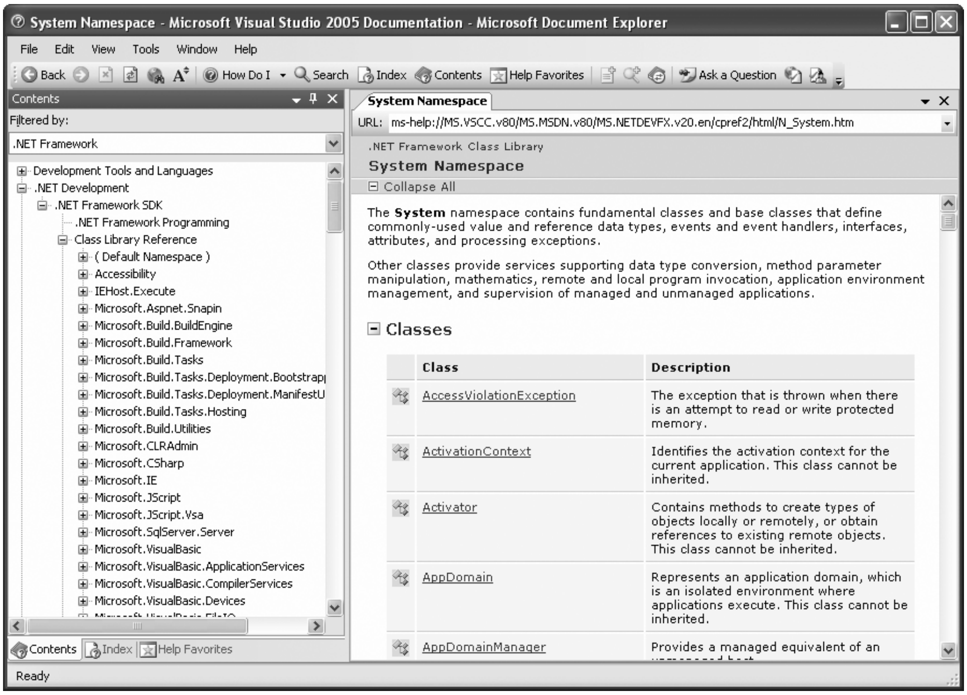


Figure 3-3. *A look at two namespaces*

Namespaces can organize all the different types in the class library. Without namespaces, these types would all be grouped into a single long and messy list. This sort of organization is practical for a small set of information, but it would be impractical for the thousands of types included with .NET.

Many of the chapters in this book introduce new .NET classes and namespaces. For example, in the chapters on web controls, you'll learn how to use the objects in the System.Web.UI namespace. In the chapters about web services, you'll study the types in the System.Web.Services namespace. For databases, you'll turn to the System.Data namespace. In fact, you've already learned a little about one namespace: the basic System namespace that contains all the simple data types explained in the previous chapter.

To continue your exploration after you've finished the book, you'll need to turn to the MSDN reference, which painstakingly documents the properties, methods, and events of every class in every namespace (see Figure 3-4). If you have Visual Studio installed, you can view the MSDN Help by selecting Start ➤ Programs ➤ Microsoft Visual Studio 2005 ➤ Microsoft Visual Studio 2005 Documentation (the exact path depends on the version of Visual Studio you've installed). You can find class reference information, grouped by namespace, under the .NET Development ➤ .NET Framework SDK ➤ Class Library Reference node.


System Namespace - Microsoft Visual Studio 2005 Documentation - Microsoft Document Explorer
File Edit View Tools Window Help
Back How Do I Search Index Contents Help Favorites Ask a Question
Contents
Filtered by:
.NET Framework
System Namespace
Collapse All
Contents Index Help Favorites
Ready

URL: ms-help://MS.VSCC.v80/MS.MSDN.v80/MS.NETDEVFX.v20.en/cpref2/html/N_System.htm

- Development Tools and Languages
- .NET Development
  - .NET Framework SDK
    - .NET Framework Programming
    - Class Library Reference
      - (Default Namespace)
      - Accessibility
      - IEHost.Execute
      - Microsoft.Aspnet.Snapin
      - Microsoft.Build.BuildEngine
      - Microsoft.Build.Framework
      - Microsoft.Build.Tasks
      - Microsoft.Build.Tasks.Deployment.Bootstrap
      - Microsoft.Build.Tasks.Deployment.ManifestU
      - Microsoft.Build.Tasks.Hosting
      - Microsoft.Build.Utilities
      - Microsoft.CLRAdmin
      - Microsoft.CSharp
      - Microsoft.IE
      - Microsoft.JScript
      - Microsoft.JScript.Vsa
      - Microsoft.SqlServer.Server
      - Microsoft.VisualBasic
      - Microsoft.VisualBasic.ApplicationServices
      - Microsoft.VisualBasic.CompilerServices
      - Microsoft.VisualBasic.Devices

.NET Framework Class Library

System Namespace

The **System** namespace contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.

Other classes provide services supporting data type conversion, method parameter manipulation, mathematics, remote and local program invocation, application environment management, and supervision of managed and unmanaged applications.

Classes

	Class	Description
	AccessViolationException	The exception that is thrown when there is an attempt to read or write protected memory.
	ActivationContext	Identifies the activation context for the current application. This class cannot be inherited.
	Activator	Contains methods to create types of objects locally or remotely, or obtain references to existing remote objects. This class cannot be inherited.
	AppDomain	Represents an application domain, which is an isolated environment where applications execute. This class cannot be inherited.
	AppDomainManager	Provides a managed equivalent of an

Figure 3-4. *The MSDN Class Library reference*

Using Namespaces

Often when you write ASP.NET code, you'll just use the namespace that Visual Studio creates automatically. If, however, you want to organize your code into multiple namespaces, you can define the namespace using a simple block structure, as shown here:

```
Namespace MyCompany

    Namespace MyApp

        Public Class Product
            ' Code goes here.
        End Class

    End Namespace

End Namespace
```

In the preceding example, the Product class is in the namespace MyCompany.MyApp. Code inside this namespace can access the Product class by name. Code outside it needs to use the fully qualified name, as in MyCompany.MyApp.Product. This ensures that you can use the components from various third-party developers without worrying about a name collision. If those developers follow the recommended naming standards, their classes will always be in a namespace that uses the name of their company and software product. The fully qualified name of a class will then almost certainly be unique.

Namespaces don't take an accessibility keyword and can be nested as many layers deep as you need. Nesting is purely cosmetic—for instance, in the previous example, no special relationship exists between the MyCompany namespace and the MyApp namespace. In fact, you could create the namespace MyCompany.MyApp using this syntax without using nesting:

```
Namespace MyCompany.MyApp

    Public Class Product
        ' Code goes here.
    End Class

End Namespace
```

You can declare the same namespace in various code files. In fact, more than one project can even use the same namespace. Namespaces are really nothing more than convenient, logical containers that help you organize your classes.

Tip If you're using Visual Studio, all your code will automatically be placed in a projectwide namespace. By default, this namespace has the same name as your project. For more information, refer to Chapter 4, which tackles Visual Studio in detail.

Importing Namespaces

Having to type long, fully qualified names is certain to tire your fingers and create overly verbose code. To simplify matters, it's standard practice to import the namespaces you want to use. When you import a namespace, you don't need to type the fully qualified type names. Instead, you can use any types in that namespace as though they were defined locally.

To import a namespace, you use the Imports statement. These statements must appear as the first lines in your code file, outside of any namespaces or classes:

```
Imports MyCompany.MyApp
```

Consider the situation without importing a namespace:

```
Dim salesProduct As New MyCompany.MyApp.Product()
```

It's much more manageable when you import the MyCompany.MyApp namespace. Once you do, you can use this shortened syntax instead:

```
Dim salesProduct As New Product()
```

Importing namespaces is really just a convenience. It has no effect on the performance of your application. In fact, whether you use namespace imports, the compiled IL code will look the same. That's because the language compiler will translate your relative class references into fully qualified class names when it generates an EXE or DLL file.

Assemblies

You might wonder what gives you the ability to use the class library namespaces in a .NET program. Are they hardwired directly into the language? The truth is that all .NET classes are contained in *assemblies*. Assemblies are the physical files that contain compiled code. Typically, assembly files have the extension .exe if they are stand-alone applications, or .dll if they're reusable components.

Tip The .dll extension is also used for code that needs to be executed (or *hosted*) by another type of program. When your web application is compiled, it's turned into a DLL file, because your code doesn't represent a stand-alone application. Instead, the ASP.NET engine executes it when a web request is received.

A strict relationship doesn't exist between assemblies and namespaces. An assembly can contain multiple namespaces. Conversely, more than one assembly file can contain classes in the same namespace. Technically, namespaces are a *logical* way to group classes. Assemblies, however, are a *physical* package for distributing code.

The .NET classes are actually contained in a number of assemblies. For example, the basic types in the System namespace come from the mscorlib.dll assembly. Many ASP.NET types are found in the System.Web.dll assembly. In addition, you might want to use other, third-party assemblies. Often, assemblies and namespaces have the same names. For example, you'll find the namespace System.Web in the assembly file System.Web.dll. However, this is a convenience, not a requirement.

When compiling an application, you need to tell the language compiler what assemblies the application uses. By default, a wide range of .NET assemblies is automatically supported by the compiler. (Technically, these default assemblies are defined in a web.config file that applies settings for the entire computer and is found in a directory like c:\Windows\Microsoft.NET\Framework\v2.0.40607\Config, depending on the version of the .NET Framework you have installed.) If you need to use additional assemblies, you need to define them in a configuration file for your website. Visual Studio makes this process seamless, letting you add assembly references to the configuration file with a couple of quick mouse clicks.

Advanced Class Programming

Part of the art of object-oriented programming is determining class relations. For example, you could create a Product object that contains a ProductFamily object or a Car object that contains four Wheel objects. To create this sort of class relationship, all you need to do is define the appropriate variable or properties in the class. This type of relationship is called *containment*.

For example, the following code shows a ProductCatalog class, which holds an array of Product objects:

```
Public Class ProductCatalog
    Private _products() As Product

    ' (Other class code goes here.)
End Class
```

In ASP.NET programming, you'll find special classes called *collections* that have no purpose other than to group various objects. Some collections also allow you to sort and retrieve objects using a unique name. In the previous chapter, you saw an example with the ArrayList, which provides a dynamically resizable array. Here's how you might use the ArrayList to modify the ProductCatalog class:

```
Public Class ProductCatalog
    Private _products As New ArrayList()

    ' (Other class code goes here.)
End Class
```

This approach has benefits and disadvantages. It makes it easier to add and remove items from the list, but it also removes a useful level of error checking, because the ArrayList supports any type of object. You'll learn more about this issue later in this chapter (in the "Generics" section).

In addition, classes can have a different type of relationship known as *inheritance.*

Inheritance

Inheritance is a form of code reuse. It allows one class to acquire and extend the functionality of another class. For example, you could create a class called TaxableProduct that inherits (or *derives*) from Product. The TaxableProduct class would gain all the same methods, properties, and events of the Product class. You could then add additional members that relate to taxation:

```
Public Class TaxableProduct
    Inherits Product

    Private _taxRate As Decimal = 1.15

    Public ReadOnly Property TotalPrice() As Decimal
        Get
            ' The code can access the Price property because it's
            ' a public member of base Product class.
            Return (Price * __taxRate)
            ' The code cannot access the private _price variable, however,
            ' because it's a private member of the base class.
        End Get
    End Property

End Class
```

This technique appears much more useful than it really is. In an ordinary application, most classes use containment and other relationships instead of inheritance, which can complicate life needlessly without delivering many benefits. Dan Appleman, a renowned .NET programmer, once described inheritance as "the coolest feature you'll almost never use."

In all honesty, you'll see inheritance at work in ASP.NET in one place. Inheritance allows you to create a custom class that inherits the features of a class in the .NET class library. For example, when you create a custom web form, you actually inherit from a basic Page class to gain the standard set of features. Similarly, when you create a custom web service, you inherit from the WebService class. You'll see this type of inheritance throughout this book.

There are many more subtleties of class-based programming with inheritance. For example, you can override parts of a base class, prevent classes from being inherited, or create a class that must be used for inheritance and can't be directly created. However, these topics aren't covered in this book, and they aren't required to build ASP.NET applications.

Shared Members

The beginning of this chapter introduced the idea of shared properties and methods, which can be used without a live object. Shared members are often used to provide helper functionality (such as conversion routines, validation tests, or miscellaneous pieces of information) that you'll want to access without being forced to create an object. The .NET class library uses shared members heavily (as with the System.Math class explored in the previous chapter).

Shared members have a wide variety of possible uses. Sometimes they provide basic conversions and utility functions that support a class. To create a shared property or method, you just need to use the Shared keyword right after the accessibility keyword.

The following example shows a TaxableProduct class that contains a shared TaxRate property and private variable. This means there is one copy of the tax rate information, and it applies to all TaxableProduct objects.

```
Public Class TaxableProduct
    Inherits Product

    ' (Other class code omitted for clarity.)

    Private Shared _TaxRate As Decimal = 1.15D

    ' TaxRate is shared, which means that you can call
    ' TaxableProduct.TaxRate, even without an object.
    Public Shared Property TaxRate() As Decimal
        Get
            Return _TaxRate
        End Get
        Set(value As Decimal)
            _TaxRate = value
        End Set
    End Property

End Class
```

You can now get or set the tax rate information directly from the class, without needing to create an object first:

```
' Change the TaxRate. This will affect all TotalPrice calculations for any
' TaxableProduct object.
TaxableProduct.TaxRate = 1.24D
```

Shared data isn't tied to the lifetime of an object. In fact, it's available throughout the life of the entire application. This means shared members are the closest thing .NET programmers have to global data.

A shared member can't access an instance member. To access a nonshared member, it needs an actual instance of your object.

Casting Objects

Objects can be converted with the same syntax that's used for simple data types. However, an object can be converted only into three things: itself, an interface that it supports, or a base class from which it inherits.

This process is rarely referred to as conversion, because it doesn't actually alter the object. Instead, it changes the way your code "sees" the object—and it thereby changes what your code can and cannot do with the object.

For example, you could cast a TaxableProduct object to a Product object. This operation is allowed because TaxableProduct derives from Product and, thus, every TaxableProduct object *is* a genuine Product object.

```
Dim myTaxableProduct As New TaxableProduct()

Dim myProduct As Product
myProduct = myTaxableProduct
```

When you perform this operation, you don't actually change the TaxableProduct object. The same object remains floating as a blob of binary data somewhere in memory. What you change is the way you access that object. From this point on, when you use the myProduct variable, you're treating the TaxableProduct as an ordinary Product. In other words, you're only able to access a subset of its features—those methods and properties that are defined in the Product class. You won't be able to access the TotalPrice property. Although that information is still stored in memory, it's inaccessible to your code—unless you cast the reference back to a TaxableProduct or access it through the myTaxableProduct reference instead.

■**Note** One of the reasons casting is used is to facilitate more reusable code. For example, you might design an application that uses the Product object. That application is actually able to handle any Product-derived class. Your application doesn't need to distinguish between all the different derived classes (TaxableProduct, NonTaxableProduct, PromoProduct, and so on); it can work seamlessly with all of them.

The following example creates a TaxableProduct object, casts it to a Product reference, and then checks whether the object can be safely cast back into a TaxableProduct (it can). You'll notice that the actual conversion uses the CType() function introduced in the previous chapter.

```
' Define two empty variables (don't use the New keyword).
Dim theProduct As Product
Dim theTaxableProduct As TaxableProduct

' This works, because TaxableProduct derives from Product.
theProduct = New TaxableProduct()

' This will be True.
If TypeOf theProduct Is TaxableProduct Then
    ' Convert theObject, and assign to theTaxableProduct.
    theTaxableProduct = CType(theProduct, TaxableProduct)
End If

Dim totalPrice As Decimal

' This works.
totalPrice = theTaxableProduct.TotalPrice

' This won't work, even though theTaxableProduct and theProduct are the same
' object. The compiler sees that theProduct is declared as a Product variable,
' and the Product class doesn't provide a TotalPrice property.
totalPrice = theProduct.TotalPrice
```

At this point, it might seem that being able to convert objects is a fairly specialized technique that will be required only when you're using inheritance. This isn't always true. Object conversions are also required when you use some particularly flexible classes.

One example is the ArrayList class introduced in the previous chapter. The ArrayList is designed in such a way that it can store any type of object. To have this ability, it treats all objects in the same way—as instances of the root System.Object class. (All classes in .NET inherit from System.Object at some point, even if this relationship isn't explicitly defined in the class code.) The end result is that when you retrieve an object from an ArrayList collection, you need to cast it from a System.Object to its real type, as shown here:

```
' Create the ArrayList.
Dim products As New ArrayList()

' Add several Product objects.
products.Add(product1)
products.Add(product2)
products.Add(product3)

' Retrieve the first item, with casting.
Dim retrievedProduct As Product = CType(products(0), Product)

' This works.
Respose.Write(retrievedProduct.GetHtml())

' Retrieve the first item, as an object. This doesn't require casting,
' but you won't be able to use any of the Product methods or properties.
Dim retrievedObject As Object = products(0)

' This generates a compile error. There is no Object.GetHtml() method.
Respose.Write(retrievedObject.GetHtml())
```

As you can see, if you don't perform the casting, you won't be able to use the methods and properties of the object you retrieve. You'll find many cases like this in .NET code, where your code is handed one of several possible object types and it's up to you to cast the object to the correct type in order to use its full functionality.

■**Note** Occasionally, you might run into a custom method that "converts" an object to another data type. For example, you can use the ToString() method in many objects to get a string that's based on that object. However, this process isn't really a conversion—instead, you're generating a *representation* of your object. This representation probably doesn't preserve all the data of your original object, and usually the conversion is one-way only.

Partial Classes

Partial classes give you the ability to split a single class into more than one source code (.vb) file. For example, if the Product class becomes particularly long and intricate, you might decide to break it into two pieces, as shown here:

```
' This part is stored in file Product1.vb.
Public Partial Class Product
    Private _name As String
    Private _price As Decimal
    Private _imageUrl As String

    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            _name = value
        End Set
    End Property

    Public Property Price() As Decimal
        Get
            Return _price
        End Get
        Set(ByVal value As Decimal)
            _price = value
        End Set
    End Property

    Public Property ImageUrl() As String
        Get
            Return _imageUrl
        End Get
        Set(ByVal value As String)
            _imageUrl = value
        End Set
    End Property
    Public Sub New(name As String, price As Decimal)
        _name = name
        _price = price
    End Sub
End Class

' This part is stored in file Product2.vb.
Public Partial Class Product
    Public Function GetHtml() As String
        Dim htmlString As String
        htmlString = "<h1>" & _name & "</h1><br />"
        htmlString &= "<h3>Costs: " & _price.ToString() & "</h3><br />"
        htmlString &= "<img src=" & _imageUrl & ">"
        Return htmlString
    End Function
End Class
```

A partial class behaves the same as a normal class. This means every method, property, and variable you've defined in the class is accessible everywhere, no matter which source file contains it. When you compile the application, the compiler tracks down each piece of the Product class and assembles it into a complete unit. It doesn't matter what you name the source code files, so long as you keep the class name consistent.

Partial classes don't offer much in the way of solving programming problems, but they can be useful if you have extremely large, unwieldy classes. The real purpose of partial classes in .NET is to hide automatically generated designer code by placing it in a separate file from your code. Visual Studio uses this technique when you create web pages for a web application and forms for a Windows application.

■ **Tip** Technically, you only need to use the Partial keyword on *one* of the class declarations. As long as the compiler finds one partial definition, it assumes all the others are partial as well.

Generics

Generics are a more subtle and powerful feature than partial classes. Generics allow you to create classes that are parameterized by type. In other words, you 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.

To understand how this works, it's easiest to consider some of the .NET classes that support generics. In the previous chapter, you learned how the ArrayList class allows you to create a dynamically sized collection that expands as you add items and shrinks as you remove them. The ArrayList has one weakness, however—it supports any type of object. This makes it extremely flexible, but it also means you can inadvertently run into an error. For example, imagine you use an ArrayList to track a catalog of products. You intend to use the ArrayList to store Product objects, but there's nothing to stop a piece of misbehaving code from inserting strings, integers, or any arbitrary object in the ArrayList. Here's an example:

```
' Create the ArrayList.
Dim products As New ArrayList()

' Add several Product objects.
products.Add(product1)
products.Add(product2)
products.Add(product3)

' Notice how you can still add other types to the ArrayList.
products.Add("This string doesn't belong here.")
```

The solution is a new List collection class. Like the ArrayList, the List class is flexible enough to store different objects in different scenarios. But because it uses generics, you must lock it into a specific type whenever you instantiate a List object. To do this, you specify the class you want to use in parentheses, preceded by the word "Of." So if you want to create a collection of products, you need this code statement:

```
' Create the List for storing Product objects.
Dim products As New List(Of Product)()
```

Now you can add only Product objects to the collection:

```
' Add several Product objects.
products.Add(product1)
products.Add(product2)
products.Add(product3)

' This line fails. In fact, it won't even compile.
products.Add("This string can't be inserted.")
```

You can find the List class, and many more collections that use generics, in the System.Collections.Generic namespace. (The original ArrayList resides in the System. Collections namespace.)

■**Note** Now that you've seen the advantage of the List class, you might wonder why .NET includes the ArrayList at all. In truth, the ArrayList is still useful if you really do need to store different types of objects in one place (which isn't terribly common). However, the real answer is that generics weren't implemented in .NET until version 2.0, so many existing classes don't use them because of backward compatibility.

You can also create your own classes that are parameterized by type, such as the List collection. Creating classes that use generics is beyond the scope of this book, but you can find a solid overview in the Visual Studio Help. Look for the "generics [Visual Basic]" index entry.

The Last Word

At its simplest, object-oriented programming is the idea that your code should be organized into separate classes. If followed carefully, this approach leads to code that's easier to alter, enhance, debug, and reuse. Now that you know the basics of object-oriented programming, you can take a tour of the premier ASP.NET development tool: Visual Studio 2005.