CHAPTER 5

BUILDING THE BUSINESS LOGIC LAYER

If you have class, you've got it made. If you don't have class, no matter what else you have, it won't make up for it.
—Ann Landers

If you have classes in your application, you've got it made. If you don't have classes, no matter what else you have, it won't make up for it. Classes are central to development in .NET.

In fact, it is difficult to build a .NET application without using classes. If you added a form to your project, you have already created a class. A form is just a class that inherits from the .NET Framework `System.Windows.Forms.Form` class, which gives the class the attributes and behaviors of a form.

In the preceding chapter, you saw how to use classes to build the user interface layer. This chapter shows you how to build classes for the business logic layer.

This chapter covers the fundamentals of creating classes and defining properties and methods. It also details more advanced topics such as using generics and building a base business object class. The Purchase Tracker sample application is used to demonstrate these techniques.

What Does This Chapter Cover?

This chapter demonstrates the following techniques:

- Creating a class
- Documenting the class with XML comments
- Adding exception classes to your class file
- Defining properties

- Defining property accessibility
- Understanding generics
- Handling `Nullable` types
- Defining methods
- Passing parameters `ByVal` or `ByRef`
- Overloading methods
- Marking methods as obsolete
- Creating a base business object class
- Overriding base class members

This chapter covers the basics of how to create a class and then builds on those basics to detail some of the new Visual Basic 2005 features such as XML comments and generics. If you have already been doing object-oriented programming in Visual Basic, you already know the basics. But if you want to "build along" as you read through this book, work through the basics before moving on to the more advanced features later in this chapter.

Creating a Class

A **class** describes the things in your application, such as customers or products. Each piece of data associated with the class is defined as a **property** of the class. Each set of functionality associated with the class is defined as a **method** of the class.

For example, the Purchase Tracker sample application works with products. The products are described by a `Product` class. Each attribute of the products, such as name, number, description, price, and so on, is represented in the class as a property. Each process that must be performed for the products, such as retrieving, saving, and so on, is defined in the class as a method.

A single item, such as an individual product (a ring or sword, for example), is represented by an **object** created from the class. Because an object is an instance of a class, the act of creating an object from the class is called **instantiation**.

A common metaphor is to think of the class as the blueprint, and the object as the building constructed from the blueprint. Any number of buildings can be created from the same blueprint. Another metaphor is a cookie cutter. The class is the cookie cutter, and the objects are all the cookies created from the cookie cutter.

NOTE: The terms **class** and **object** are sometimes used interchangeably. Technically, however, the class is the data and logic that you define at design time. The objects are instances of the class created at runtime.

The phrase "business objects" is really a misnomer, because they are really "business classes." In this book, the term "business object classes" is sometimes used to distinguish the difference.

If you are building a nontrivial application, build it as a set of layers, as described in Chapter 2, "Designing Software." Implement each layer as a separate project in a solution, as described in Chapter 3, "Building Projects." This gives you separately compiled components, one for each layer.

You build each layer as a set of classes. The user interface layer is comprised of a set of form classes (as shown in Chapter 4, "Building the User Interface Layer") following the user interface design. The business logic layer (as described in this chapter) includes the set of classes you build following the implementation design. The data layer (detailed in Chapter 8, "Building the Data Access Layer") contains classes that provide the interaction between the database and the business logic layer.

When you construct the business logic layer, it is important to define the pertinent set of classes. For each class, you define the appropriate properties and methods. This ensures that the correct set of information and logic is encapsulated in each class, making it easier to work with and maintain the class.

NOTE: Even if you don't go through the design phase, you need to think through the application to identify the appropriate set of classes for your business logic layer.

You normally define one class for each key thing involved with the application. For example, the Purchase Tracker sample application has products, customers, and purchases. The products map to a `Product` class, with properties to manage product information and methods to retrieve, save, and perform any other required processing on product information. The customers map to a `Customer` class, and so on. For more information on defining classes for your application, see Chapter 2.

You can also define classes for other implementation logic. For example, you could create a class to manage application logging or security. These implementation-based classes were also discussed in Chapter 2.

This section details the process of creating a class. You can use these techniques to create each class needed by your application.

Adding a Class to a Project

There are many ways to add a class to a project. As discussed in the preceding chapter, adding a form project item actually adds two class files to the project. It adds a class in one file with a .vb extension and a partial class in another file with a .designer.vb extension.

When building the business logic layer, you normally add one class project item for each business object class (such as `Product` and `Customer`) and one for each implementation class (such as `Logging` and `Security`).

NOTE: You may want to define standard implementation classes in a separate utility component instead of in the business object component. That way, it can more readily become a part of your reusable framework, as discussed in Chapter 2.

To add a class to a project:

1. Right-click the project in Solution Explorer and select **Add | New Item** from the context menu, *or* select **Project | Add New Item** from the main menu bar.
   Alternatively, you could select **Add | Class** from the context menu, *or* select **Project | Add Class** from the main menu bar.
2. Select the **Class** template, name the class, and click the **Add** button.
   If you created your own class template using the steps in Chapter 3, you can use your template here.
   Use standard naming conventions for your class name. The most common standard is to name the class using the singular name of the business entity or implementation feature represented by the class. For products the class name would be `Product`, for logging the class name would be `Logging`, and so on.
   Visual Studio creates the class file with a .vb extension, adds it to Solution Explorer, and then displays the class in the Code Editor.

When Visual Studio creates the class file, it automatically generates the class declaration as follows:

```
Public Class Product

End Class
```

You can add any number of classes to your projects as needed by your application. Regardless of the class's purpose or location, the basic process of building a class is the same.

NOTE: Throughout this chapter, classes are created in the business object Class Library project. You can use these same techniques to add classes to other parts of your application. For example, your Windows Application project may require classes to manage user interface features such as standard grid processing. Or you may create a utility or general library component that requires classes.

Building Along

For the Purchase Tracker sample application:

- Visual Studio created a default class for you when you created the business object Class Library project (**PTBO**). In *Solution Explorer*, change the name of this default class from Class1 to **Product**. When you change the name of a class in Solution Explorer, Visual Studio changes the class name in the Code Editor accordingly.

NOTE: When you change the class name in Solution Explorer, Visual Studio modifies the class name in your code only if the class name matched the name defined in Solution Explorer. For example, say you don't modify the class name in Solution Explorer, but you instead change the name directly in the class file. If you later change the name in Solution Explorer, it does not change the name you entered in the class file.

Documenting the Class

It is always a good idea to add documentation for a class immediately after adding the class. By adding the documentation right away, you focus on the class's purpose, which helps you keep the class encapsulated. It is also much easier to document each class as you go along instead of facing the large task of going back later and documenting all the classes.

To document the class:

1. Open the class in the Code Editor.
2. Move the insertion point immediately before the word `Public` in the `Public Class` statement.
3. Type three comment markers, defined in Visual Basic as apostrophes (`'''`), and press the Enter key.
   The XML comments feature automatically creates the structure of your class documentation as follows:

```
''' <summary>
'''
''' </summary>
''' <remarks></remarks>
Public Class Product

End Class
```

NOTE: If you type the three comment markers in the empty line above the class definition instead of on the same line as the class definition, you don't need to press the Enter key to generate the documentation structure.

4. Type a summary of the class's purpose between the `summary` tags and any remarks between the `remark` tags.
   Your documentation may be similar to this:

```
''' <summary>
''' Provides product management features such as
''' retrieving product data and saving product changes
''' </summary>
''' <remarks>Use this class to work with products
''' </remarks>
```

Use the `summary` tags to describe the class and the `remarks` tags to add supplemental information. The `summary` is the most important tag because it is the one used by Visual Studio.

When you provide a summary of the class using XML comments, your class displays documentation about itself in appropriate places within Visual Studio, such as in the List Members box, shown in Figure 5.1. Open the List Members box by typing a part of the class name in the Code Editor and pressing Ctrl+Spacebar *or* by selecting **Edit | Intellisense | List Members** from the main menu bar *or* by clicking the **Display an Object Member List** icon on the Text Editor toolbar.

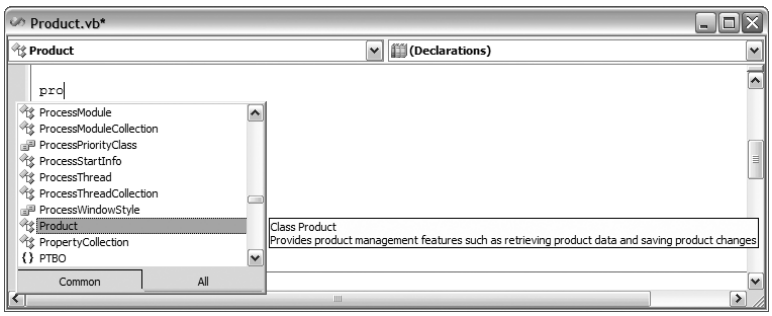

Figure 5.1 The documentation provided in the List Members box is the summary defined in the XML documentation for the class.

Using XML comments to document your classes makes it easier for you and other developers to work with your classes.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add documentation for the **Product** class using XML comments.

In the Code Editor, view the comments by typing `pro` and then pressing Ctrl+Spacebar to display the List Members box, as shown in Figure 5.1.

Organizing the Code Structure

Code that is organized is much easier to maintain, because you can quickly find the code that needs to be changed. When you standardize this organizational structure, any member of the team can quickly locate and modify any class code, because the code structure of each class is the same.

A class is normally composed of a set of properties and public and private methods. Public methods are methods that can be called from outside of the class, and private methods are those used only within the class.

You define the organizational structure of a class using regions, as described in Chapter 3. If you built a class template using the information in Chapter 3 and then used that template when creating your class, these regions are already defined in your code. If not, you can create the regions as follows:

```
Public Class Product

#Region " Properties"

#End Region

#Region " Public Methods"

#End Region

#Region " Private Methods"

#End Region

End Class
```

Add any other regions as needed to define the standard code structure for your classes. For example, if your class has a constructor, you can add a Constructor region as defined later in this section.

NOTE: Once you define the standard set of regions to use in your classes, define a custom class template, as described in Chapter 3. Provide this template to all the members of your team. This ensures that all the classes of your application follow your standard structure.

As you develop the code for the application, properties are added to the Properties region, public methods are added to the Public Methods region, and so on. To further aid in the organization, you can insert the properties and methods within each region in alphabetical order.

NOTE: If you don't use a region, delete it. For example, if your class has no private methods, delete the Private Methods region. This makes it clear that the class has no private methods without opening the region to see that it has no private methods.

You can also put regions within regions. So, you can put each method in its own region within the Private Methods or Public Methods region. This makes it easier to focus on the code, because you can close all methods except for the one you are working with.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add the regions as defined in this section.

In the Code Editor, open and close the regions to see how they hide and show their contents.

Instantiating an Object

Once a class is defined, you can create objects from the class. This is called object instantiation. You then use the object to access the properties and methods defined in the class.

To create an object from a class:

1. Declare an object variable.
   For example:

   ```
   Dim prod as Product
   ```

2. Create a new instance of the class, and assign the object variable to reference that new instance.
   For example:

   ```
   prod = New Product
   ```

NOTE FOR VB6 DEVELOPERS: The `Set` statement is no longer required when assigning an object reference to an object variable.

3. Access the properties and methods for the object by using the object variable and a period (`.`).
   For example:

```
prod.ProductName
```

Alternatively, you can accomplish the first two steps in one code line as follows:

```
Dim prod as New Product
```

This line declares the object variable and assigns it to reference a new object from the `Product` class.

NOTE FOR VB6 DEVELOPERS: In classic versions of Visual Basic, it was highly recommended that you not put the `New` keyword on the declaration because of how instances were created. This is no longer the case, so the preceding syntax is generally recommended.

Defining the Constructor

A **constructor** is a built-in method in a class that the .NET runtime executes when an object is first instantiated (created). You add code in the constructor to perform any initialization operations for a new object.

You define a constructor by creating a `New` method in the class as follows:

```
#Region " Constructors"
    Public Sub New()

    End Sub
#End Region
```

The constructor executes when you create an object from the class. For example:

```
Dim prod as New Product
```

When the .NET runtime executes this line of code, it calls the `New` method in the `Product` class and runs any code in your constructor.

NOTE FOR VB6 DEVELOPERS: There is no longer an `Initialize` event for a class. Any code you would have put into the `Initialize` event can go into the constructor.

You can pass data into the constructor by defining parameters. The constructor is then called a **parameterized constructor**. For example, this constructor defines a `productID` as a parameter:

```
#Region " Constructors"
     Public Sub New(ByVal productID as Integer)

     End Sub
#End Region
```

You pass the parameter to the constructor when creating the instance of the object:

```
Dim prod as New Product(1)
```

or

```
Dim prod as Product
prod = New Product(1)
```

Multiple constructors can be defined for a class. For example, you could define a constructor with no parameters and one with a parameter. The .NET runtime knows which constructor to call based on the parameters passed to the constructor. Defining one method (in this case, `New`) with two different signatures is called **overloading** and is described in detail later in this chapter.

In many cases, you don't need any specialized code to be executed when the object is instantiated, so you don't need to create a constructor. If you don't create one, the runtime executes an empty constructor for you.

In other cases, you may want a more formal object creation pattern. The most common formal pattern for object creation is called the Factory pattern. A **pattern** is a reusable solution for a recurring problem. The **Factory pattern** defines a standard solution for creating objects. Instead of creating an instance of a class using the `New` keyword, the Factory pattern defines a method that creates and returns an instance of the class.

This makes the process of creating object instances more explicit. See the "Additional Reading" section at the end of Chapter 1, "Introduction to OO in .NET," for more information on patterns.

NOTE: The Factory pattern has two implementations: one using a factor *class* and the other using a factory *method*. This example uses a factory method.

If you elect to apply the Factory method, you no longer use this style of code to create an object:

```
Dim prod as Product
prod = New Product
```

You instead use code like this:

```
Dim prod as Product
prod = Product.Create()
```

The `Create` method, and the syntax used to call it, are defined in detail later in this chapter.

NOTE: If you select to use a Factory method pattern, other code in the application should use the Factory method and not directly instantiate an object using the `New` keyword. To prevent any code from creating an instance of your object without using the Factory method, define an empty constructor and use the `Private` keyword. For example:

```
#Region " Constructors"
    Private Sub New()

    End Sub
#End Region
```

Use constructors or a Factory pattern method to define any code that must be executed when first creating an object from the class. Don't bother creating a constructor if you have no initialization code.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add a region for a constructor.
- Add a **private** constructor with no code in the constructor. This ensures that code outside of the class cannot create objects from the class. Instead, a Factory pattern method is added in a later "Building Along" activity.
- Open the **ProductWin** form in the Code Editor.
- In the **Load** event for the **ProductWin** form, add code to create an object from the **Product** class:

```
Dim prod As Product
prod = New Product
```

  Visual Studio underlines part of the second code line and displays an error: "'PTBO.Product.Private Sub New()' is not accessible in this context because it is 'Private'".

NOTE: If Visual Studio displays an error on the first line stating "Type 'Product' is not defined," either you did not set a reference from the Windows Application project (PTWin) to the business object Class Library project (PTBO), or you did not import the PTBO namespace. See the "Referencing Projects in a Solution" section in Chapter 3 for details.

- In the **Product** class, comment out the private constructor by inserting comment markers before each of the two code lines of the constructor. The lines of code added in the preceding step are now syntactically correct.

Leave the private constructor commented out for now. This allows you to try out some of the upcoming features. The private constructor will be uncommented when the Factory pattern method is added in a later "Building Along" activity.

Defining the Destructor

A **destructor** is a built-in method in a class that the .NET runtime executes when an object is destroyed. You define a destructor by creating a `Finalize` method in the class. But using a destructor is not recommended, because it does not necessarily execute when you expect it to, and it may not execute at all.

It may seem that the destructor should execute when you specify that you no longer need the object. For example, the following code defines that you no longer need the specified object reference:

```
prod = Nothing
```

But this code does not destroy the object. It just releases the object, making it available for destruction.

Your code does not define when an object is destroyed—the .NET garbage collector does. The **garbage collector** is a memory manager that manages the allocation and release of memory for all .NET applications. The garbage collector performs garbage collection when it needs to. It then releases the memory allocated to a managed object and destroys the object if that object is no longer used. But because you cannot predict when the garbage collector will perform garbage collection, you don't know exactly when your object will be destroyed. Therefore, you cannot know when your destructor will be executed. (See the "Additional Reading" section for more information on the garbage collector.)

In most cases, you don't need to write any cleanup code that executes when an object is destroyed, so you don't need to care about this. You can just allow the garbage collector to destroy your object when it gets around to it.

But if your object works with unmanaged resources, you do need to write some cleanup code. **Unmanaged resources** are system resources that are not directly managed by the .NET runtime, such as database connections, window handles, open files, network connections, and graphic resources. You need to explicitly release unmanaged resources when your object is released.

Because the execution of the `Finalize` method is unpredictable, do not put the code to release unmanaged resources in the destructor. Create a `Dispose` method instead, and explicitly call `Dispose` when you release the object. Add code in the `Dispose` method to perform any cleanup activities required for your object—primarily, releasing any unmanaged resources used by your object. And since you are explicitly calling `Dispose`, you control when the unmanaged resources are released.

NOTE FOR VB6 DEVELOPERS: There is no longer a `Terminate` event for a class. Any code you would have put into the `Terminate` event can go in the `Dispose` method.

Define a `Dispose` method by implementing the .NET Framework `IDisposable` interface. Using the `IDisposable` interface to define your `Dispose` method ensures that you have a standardized programmatic interface for disposing of your objects.

To implement a `Dispose` method, do the following:

1. Open the class in the Code Editor.
2. Add the `Implements` statement to the class definition to implement the `IDisposable` interface:

```
Public Class Product
    Implements IDisposable
```

3. Press the Enter key after the name of the interface.
   The Code Editor automatically adds the signatures for all the properties and methods defined in the interface and related code to the class. This generated code uses the recommended design pattern for proper cleanup of any unmanaged resources that your application uses.

Visual Studio adds two sets of generated code as part of the `IDisposable` interface implementation, defining two `Dispose` methods. The first part of the generated code provides a `Dispose` method that you can customize to your requirements:

```
Private disposedValue As Boolean = False  ' To detect redundant
➥calls

' IDisposable
Protected Overridable Sub Dispose(ByVal disposing As Boolean)
    If Not Me.disposedValue Then
        If disposing Then
            ' TODO: free unmanaged resources when explicitly called
        End If
```

```
            ' TODO: free shared unmanaged resources
        End If
        Me.disposedValue = True
    End Sub
```

The `disposedValue` variable keeps track of whether the object has already been disposed so that it won't dispose it again.

The parameter passed to this customizable `Dispose` method defines whether `Dispose` is called from the `IDisposable` interface `Dispose` method (shown next). If so, the code should free any unmanaged resources, so put your cleanup code here, by the first TODO Task List comment. If you have shared resources (those defined without a specific instance), put that cleanup code by the second TODO Task List comment.

When all the unmanaged resources are freed, this code sets the `disposedValue` property so that it won't dispose again.

The second part of the generated code defines the implemented interface:

```
#Region " IDisposable Support "
    ' This code added by Visual Basic to correctly implement the
    ' disposable pattern.
    Public Sub Dispose() Implements IDisposable.Dispose
        ' Do not change this code.  Put cleanup code in
        ' Dispose(ByVal disposing As Boolean) above.
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
#End Region
```

This code is meant to be left unchanged. This interface `Dispose` method first calls the customizable `Dispose` method to perform the cleanup. It then calls the `SuppressFinalize` method on the garbage collector. `SuppressFinalize` tells the garbage collector that it does not need to perform further cleanup on the object because the object was already cleaned up by the customized `Dispose` method.

NOTE: Some dispose patterns suggest that you also override the `Finalize` method. In the `Finalize` method, you call the customizable `Dispose` method and pass in `False`. Overriding the `Finalize` method is not recommended

due to the performance and complexity costs of using a finalizer. See the *Framework Design Guidelines* book, listed in the "Additional Reading" section, for more information.

Unlike the `Finalize` destructor, the `Dispose` method is *not* called automatically. The code that created the object must manually call `Dispose` explicitly when destroying the object as follows:

```
prod.Dispose()
prod = Nothing
```

If your code creates an object using the `Using` statement, as defined in the preceding chapter, the .NET runtime automatically calls the `Dispose` method at the end of the `Using` block, so you don't need to explicitly call `Dispose`. The `Using` statement also ensures that `Dispose` is called even if an error occurs within the `Using` block.

In this example, the `Using` statement is as follows:

```
Using prod As Product = New Product
     ' Code to work with the object here
End Using
```

Or, if you are using a `Create` Factory pattern method:

```
Using prod As Product = Product.Create()
     ' Code to work with the object here
End Using
```

NOTE: You can use the `Using` statement only if the class (`Product` in this case) implements the `IDisposable` interface.

If your class does not have or use any unmanaged resources, it can leave it up to the garbage collector to clean things up, and no `Dispose` method is needed. If your class does use unmanaged resources, implement a `Dispose` method using the `IDisposable` interface, as described in this section. Any code that creates an object from your class must then correctly destroy it when it is finished with it by calling the object's `Dispose` method or by using the `Using` statement.

Building Along

For the Purchase Tracker sample application, do not implement the `IDisposable` interface in the `Product` class. The `Product` class does not access any unmanaged resources, so it does not need to have any special dispose processing.

Using Partial Classes

By convention, each business object class resides in a single class file. The `Product` class is in the Product.vb file, the `Customer` class is in the Customer.vb file, and so on. But that is not a requirement. A class can be divided between any number of class files.

You can break a class into two or more files by defining **partial classes**. Partial classes are used primarily in situations where you have a code generator that generates part of the class and custom code for the remainder of the class. By placing the generated code in a file separate from the custom code, you can more easily regenerate the generated code without affecting the custom code. Every time you add a form to a project, Visual Studio creates a partial class and generates the code defining the controls on your user interface in that class, as described in Chapter 4.

To define a partial class for your class:

1. Right-click the project in Solution Explorer and select **Add | New Item** from the context menu, *or* select **Project | Add New Item** from the main menu bar.
   Alternatively, you could select **Add | Class** from the context menu, *or* select **Project | Add Class** from the main menu bar.
2. Select the **Class** template, name the class, and click the **Add** button.
   You cannot have two code files with the same filename, so name the partial class with a unique name.
   Visual Studio creates the class file with a .vb extension, adds it to Solution Explorer, and then displays the class in the Code Editor.
3. In the Code Editor, add the `Partial` keyword to the class definition, and modify the class name to match the original class name.

For example, the class definition of a partial class for the `Product` class is as follows:

```
Partial Public Class Product

End Class
```

When the application is built, the code in the file for the class and the files for any partial classes are combined into one logical class. So the runtime behaves as if there is only one class.

One other benefit of partial classes is the ability to separately define Option Strict. Your primary class can have Option Strict set to On, and your partial class can have Option Strict set to Off. This is useful if you have code that needs to work with objects without concern for conversion of their types, such as when calling components written in VB6 or other Component Object Model (COM)-based technologies.

Don't use partial classes unnecessarily. Dividing a class into multiple class files for no particular purpose makes it more difficult to maintain the class. It is more difficult to find where code resides and see how it interacts with other code in the class.

Use partial classes for the defined purpose—separating generated code from custom code. If you are not writing your own code generators, you may never need to create a partial class. If you use third-party code generators, you may notice the partial classes that they create.

Adding Multiple Classes to a Class File

A single class file can contain any number of classes. Although you normally define a class within its own class file, you can add support classes for that class directly in the same class file.

For example, say you define a `ProductOutOfStockException` class that is used only by the `Product` class. You can define this exception class in the same code file as the `Product` class:

```
End Class ' End of the Product Class

<Serializable()> _
Public Class ProductOutOfStockException
    Inherits ApplicationException

    Public Sub New(ByVal message As String)
        MyBase.New(message)
    End Sub
```

```
    Public Sub New(ByVal message As String, _
       ByVal inner As Exception)
        MyBase.New(message, inner)
    End Sub

    Public Sub New( _
       ByVal info As _
       System.Runtime.Serialization.SerializationInfo, _
       ByVal context As _
       System.Runtime.Serialization.StreamingContext)
        MyBase.New(info, context)
    End Sub
End Class
```

NOTE: This class was created with the Exception snippet. See the next chapter for details on using snippets.

NOTE: The `Serializable` attribute on this class defines that the exception class can be serialized. **Serialization** is the process of converting an object into a sequence of bytes for either storage or transmission to another location. For example, you could serialize an object to a file to save the value of all the object's properties.

This attribute is added to the exception class to support remoting. **Remoting** is the process of passing an object to another computer by serializing the object. If an exception occurs on the remote computer, the exception is serialized and remoted back to the original application.

For more information on remoting, see the Lhotka book in the "Additional Reading" section. See the `System.Runtime.Serialization` .NET Framework class documentation for more information on serialization. See the later section "Obsolescing Methods" for more information on attributes.

This class inherits from `ApplicationException` to ensure that it behaves as an exception. It contains three methods, each of which calls the associated base class method. Using the same named methods with different parameters is described later, in the section "Overloading Methods." You can create your own exceptions any time using this style of exception class.

NOTE: Although the exception snippet inherits from `ApplicationException`, best practices define that you should inherit from `Exception` instead. `ApplicationException` was originally set up for your use, as defined in the preceding example. However, it was misused within the .NET Framework, so the Framework developers recommend that you do not use `ApplicationException` in your code. (See the "Additional Reading" section for the reference to the *Framework Design Guidelines* book containing this recommendation.)

Don't put multiple *business object* classes in a single class file. Reserve this feature for adding support classes or exception classes only. For example, business object-unique exceptions are an excellent type of class to add to a business object class file.

Defining Properties

The **properties** of a class define the data associated with the class. For example, a `Product` class has ProductName, ProductID, and InventoryDate properties. Each object created from the class can have a different set of values for these properties.

This section details the process of creating a property. It then covers some additional techniques for working with properties.

Creating the Property

Create a property in a class for each data attribute identified for the class during the design phase. Following best practices, defining properties requires two steps.

First you create a private variable to retain the property value. This private variable is called a **backing variable** or **backing field** and retains the property's value. You make the variable private so that it cannot be directly accessed by any code outside of the class.

Next you create a `Property` statement. The `Property` statement defines the property and the accessors used to get and set the property. The `Set` accessor, sometimes called the **setter**, sets the property's value, and the `Get` accessor, sometimes called the **getter**, returns the property's value.

This technique encapsulates the property by providing access to it only through the accessors. You can write code in the accessors to validate data, perform formatting, or any other business logic.

To define a property:

1. Open the class in the Code Editor.
2. Declare a private variable for the property.
   For example:

   ```
   Private _ProductName As String
   ```

   By making the variable private, you ensure that code outside of this class can not access the property directly. All code must access the variable value through the `Property` statement.
   Use good naming conventions for your private variable. There are several common conventions, such as prefixing the property name with `m` or `m_` to define the variable as member-level. The convention that is currently gaining popularity is to prefix the property name with an underscore to indicate that the variable should not be used anywhere in the code except in the `Property` statement.
3. Create the `Property` statement for the property.
   For example:

   ```
   Public Property ProductName() As String
   ```

   Use good naming conventions for your property name. The recommended convention is to use the property's human-readable name, concatenating the words and using Pascal case, whereby each word in the name is capitalized.
4. Press the Enter key to automatically generate the remaining structure of the `Property` statement:

   ```
   Public Property ProductName() As String
       Get

       End Get
       Set(ByVal value As String)

       End Set
   End Property
   ```

5. Add code within the `Get` and `Set` blocks.

The minimum code in the getter returns the value of the private variable:

```
Get
     Return _ProductName
End Get
```

NOTE FOR VB6 DEVELOPERS: Use the `Return` statement instead of using the property's name to return a value.

Add any other code to the getter, such as formatting or data conversions. For example, for a product number, the getter could add hyphens or other characters used by the human reader that are not necessarily stored with the actual data.

The minimum code in the setter sets the value of the private variable:

```
Set(ByVal value As String)
     _ProductName = value
End Set
```

Add any other code to the setter, such as validation or data conversion. For example, code could validate that the product name is not empty before it is assigned to its private variable.

NOTE FOR VB6 DEVELOPERS: The `Property` statement is similar to the VB6 property procedures. However, there is no separate `Let` and `Set`. The `Set` statement is no longer needed to assign object variables. Object variables can now be assigned with a simple equals sign, as with any other variable (remember from Chapter 1 that *everything* in .NET is basically an object).

Repeat these steps to define each property of your class. Alternatively, you can use code snippets or the Class Designer, as described in the next chapter, to assist you in defining the properties of your class.

Use properties to define the data managed by your business object. Use `Property` statements to provide access to the properties from other parts of the application.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add the **ProductName** property, as defined in this section.
- Open the **ProductWin** form in the Code Editor.
- In the **Load** event for the **ProductWin** form, add code to set and then display the **ProductName** property:

```
Dim prod as Product
prod = New Product
prod.ProductName = "shoes"
Debug.WriteLine(prod.ProductName) ' Displays shoes
```

Run the application. It displays your splash screen and then shows the MDI parent form. Select **Products** | **Manage Products** to display the ProductWin form. The debug message appears in the Immediate window (**Debug** | **Windows** | **Immediate**) or in the Output window (**Debug** | **Windows** | **Output**), depending on your settings.

Property Statements Versus Public Variables

The example in the preceding section seemed like much more code than simply adding a public variable. Why bother with `Property` statements?

Using a private variable and public `Property` statements has several advantages over just using public variables:

- You can add code that is executed before a property is assigned. This code can perform validation, such as to ensure that no invalid values are assigned to the property.
- You can add code that is executed before a property is retrieved. This code can format or convert the value. For example, it could add dashes to the product number for the human reader even though the dashes are not stored with the data.
- Without a `Property` statement, any code that references the class can manipulate or destroy the property value at will.
- Some of the Visual Studio tools, such as object binding, recognize only properties defined with `Property` statements. (See Chapter 7, "Binding the User Interface to the Business Objects," for more information on object binding.)

For these reasons, always use private variables and public `Property` statements to define the properties for your classes.

Documenting the Property

It is always a good idea to add documentation for a property immediately after defining the property. By adding the documentation right away, you have it in place so that you can use the documentation as you build the remainder of the application.

To document the property:

1. Open the class in the Code Editor.
2. Move the insertion point immediately before the word `Public` in the `Public Property` statement.
3. Type three comment markers, defined in Visual Basic as apostrophes (`'''`), and press the Enter key.
   The XML comments feature automatically creates the structure of your property documentation as follows:

```
''' <summary>
'''
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
```

NOTE: If you type the three comment markers in the empty line above the property definition instead of on the same line as the property definition, you don't need to press the Enter key to generate the documentation structure.

4. Type a summary of the property between the `summary` tags, the value of the property between the `value` tags, and so on.
   Your documentation may be similar to this:

```
''' <summary>
''' Gets or sets the product name
''' </summary>
''' <value>Product Name</value>
''' <returns>Product Name</returns>
''' <remarks></remarks>
```

Use the `summary` tags to describe the purpose of the `Property` statement. By convention, a `Property` statement summary begins with the text "Gets or sets the..." for a read-and-write property, "Gets the..." for read-only properties, and "Sets the..." for write-only properties.

In this example, the `value` and `returns` tags don't provide very useful information, because the product name is self-explanatory. These two tags could be deleted in this case. However, in other cases the property may not be as obvious, so the documentation defined in the XML tags is more useful. For example, a `Status` property is not as obvious, and the XML documentation could provide further information, such as what the status value means and what it actually returns.

When you provide a summary of a property using XML comments, the summary appears in appropriate places within Visual Studio. For example, the summary appears in the Intellisense List Members box when you type the object variable name and a period (`.`).

Using XML comments to document your properties makes it easier for you and other developers to work with your properties.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add documentation for the **ProductName** property using XML comments.

In the Code Editor, view the comment by typing `prod.` in the **Load** event in the **ProductWin** form.

When you type the period (`.`), Visual Studio displays the Intellisense List Members box, showing all the properties and methods for the class. Click the `ProductName` property to see the XML comments. Be sure to remove this code; otherwise, the project will have a syntax error.

Defining Property Accessibility

In most cases, properties are public. The primary purpose of properties is to provide public access to the data relating to a particular object. But in some cases, you may want the property to be read-only and, in rare cases, write-only. You can define a property's accessibility using additional keywords in the `Property` statement.

Some fields should be changed only by code in the class, not by any code outside the class. For example, a `ProductID` should not be changed by code outside the class, because the ID is the key property used to identify the product. It should be set only when the product is created and then never changed. (See Chapter 8 for more information on primary key fields.)

You can define a property to be read-only using the `ReadOnly` keyword. If you need to define a property as write-only, you can use the `WriteOnly` keyword.

```
Public ReadOnly Property ProductID() As Integer
    Get

    End Get
End Property
```

When you make the property read-only or write-only using the keyword, the code in the class cannot access the property either. If the property is read-only, the code in the class must access the private backing variable to update the value. It would be better to define the accessibility on the accessors so that the getter could be public but the setter could be private. This would allow the code in the class to set the property but make it appear read-only outside this class.

To define separate accessibility on the accessors, add an accessibility keyword to either the getter or setter:

```
Public Property ProductID() As Integer
    Get

    End Get
    Private Set(ByVal value As Integer)

    End Set
End Property
```

Notice the `Private` keyword on the setter. This allows the getter to be public but restricts the setter to be private. The code within the class can then get or set the property, and code outside the class can only get the property.

Some restrictions and rules apply when you use accessibility on the accessors:

- The accessibility on the `Property` statement must be less restrictive than the accessibility on the accessor.
  For example, you cannot define the `Property` statement to be private and then make the getter public.
- You can add accessibility to the getter or setter, but not both.
  If the getter needs to be friend and the setter needs to be private, for example, make the `Property` statement friend (the least restrictive), and make the setter private.
- If you use the `ReadOnly` or `WriteOnly` keywords, you cannot add accessibility on the accessor.

Define accessibility appropriately to ensure that your properties are accessed only as they should be. Most properties are public, but for some properties, such as IDs, define private setters to allow reading but not setting of the property.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add a **ProductID** property with a private setter, as defined in this section.
- Add code to declare a backing variable, and get and set its value in the property.
- Add documentation for the **ProductID** property using XML comments.

This new property is used later in this chapter.

Handling Nulls

A data type is said to be **nullable** if it can be assigned a value *or* a null reference. **Reference types**, such as strings and class types, are nullable; they can be set to a null reference, and the result is a null value. **Value types**, such as integers, Booleans, and dates, are *not* nullable. If you set a value type to a null reference, the result is a default value, such as 0 or

false. A value type can express only the values appropriate to its type; there is no easy way for a value type to understand that it is null.

The .NET Framework 2.0 introduces a `Nullable` class and an associated `Nullable` structure. The `Nullable` structure includes the value type itself and a field identifying whether the value is null. A variable of a `Nullable` type can represent all the values of the underlying type, plus an additional null value. The `Nullable` structure supports only value types because reference types are nullable by design.

For example, say you have a `Product` class with a `ProductID` property defined as an integer, a `ProductName` property defined as a string, and an `InventoryDate` property defined as a date. The following code sets each property to `Nothing` to assign a null reference:

```
Dim prod as Product
prod = New Product
prod.ProductID = Nothing
prod.ProductName = Nothing
prod.InventoryDate = Nothing

Debug.WriteLine(prod.ProductID)    ' Displays 0
Debug.WriteLine(prod.ProductName) ' Displays (Nothing)
Debug.WriteLine(prod.InventoryDate)
                                   ' Displays 1/1/0001 12:00:00 AM
```

If you view these values, they are `0`, `Nothing`, and `1/1/0001 12:00:00 AM`, respectively. The `ProductID` and `InventoryDate` properties are value types and therefore cannot store a null. Instead, they store a default value when they are assigned a null reference.

There may be cases, however, when you need your code to really handle a null as a null and not as a default value. It would be odd, for example, to handle a null date by hard-coding a check for the 1/1/0001 date.

To make a value type property nullable, you need to declare it using the `Nullable` structure. However, you still want your property to be strongly typed as an integer, date, Boolean, or the appropriate underlying type. The ability to use a class or structure for only a specific type of data is the purpose of generics.

Generics allow you to tailor a class, structure, method, or interface to a specific data type. So you can create a class, structure, method, or interface with generalized code. When you use it, you define that it can work only on a particular data type. This gives you greater code reusability and type safety.

NOTE: A number of generic collection classes are also provided in the .NET Framework. These are great for creating collections of objects in which only a particular type of object can be in the collection. (See the next chapter for more information on generic collections.) You can use the generic types defined in the .NET Framework or create your own.

The .NET Framework built-in `Nullable` structure is generic. When you use the structure, you define the particular data type to use.

As a specific example, an `InventoryDate` property that allows the date to be a date or a null value uses the generic `Nullable` structure as follows:

```
Private _InventoryDate As Nullable(Of Date)
Public Property InventoryDate() As Nullable(Of Date)
    Get
        Return _InventoryDate
    End Get
    Set(ByVal value As Nullable(Of Date))
        _InventoryDate = value
    End Set
End Property
```

Notice the syntax of the `Nullable` structure. Since it supports generics, it has the standard `(Of T)` syntax, where `T` is the specific data type you want it to accept. In this case, the `Nullable` structure supports dates, so the `(Of Date)` syntax is used. This ensures that the `Nullable` structure contains only a date or a null value.

You can then use this property in your application as needed. For example:

```
Dim prod as Product
prod = New Product
If prod.InventoryDate.HasValue Then
    If prod.InventoryDate.Value < Now.Date.AddDays(-10) Then
        MessageBox.Show("Need to do an inventory")
    End If
Else
    MessageBox.Show("Need to do an inventory - never been done")
End If
```

The `HasValue` property of the `Nullable` class defines whether the value type has a value—in other words, whether it is null. If it does have a value, you can retrieve the value using the `Value` property of the `Nullable` class.

NOTE: The `Nullable` type does not support the compare (=) operator. So you cannot use code such as:

```
If prod.InventoryDate = Nothing Then
```

You must instead use the `HasValue` and `Value` properties, as shown in the preceding code example.

The `Nullable` structure is exceptionally useful when you're working with databases, because empty fields in a database are often null. Assuming that you have an `InventoryDate` field in a table, you could write code as follows:

```
If dt.Rows(0).Item("InventoryDate") Is DBNull.Value Then
    prod.InventoryDate = Nothing
Else
    prod.InventoryDate = _
          CType(dt.Rows(0).Item("InventoryDate"), Date)
End If
```

The `If` statement is required here because you cannot convert a `DBNull` to a date using `CType`. So you first need to ensure that it is not a null.

Use the `Nullable` structure any time you need to support nulls in a value type, such as an integer, Boolean, or date.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add the **InventoryDate** property, as defined in this section.
- Add code to declare a backing variable, and get and set its value in the property, as defined in this section.
- Add documentation for the **InventoryDate** property using XML comments.
- Open the **ProductWin** form in the Code Editor.

- In the **Load** event for the **ProductWin** form, add code to set and then display the **InventoryDate** property:

```
Dim prod as Product
prod = New Product
prod.InventoryDate = Now()
Debug.WriteLine(prod.InventoryDate)
                        ' Displays 9/25/2006 1:45:53 PM
```

Run the application. It displays your splash screen and then shows the MDI parent form. Select **Products** | **Manage Products** to display the ProductWin form. The debug message appears in the Immediate window (**Debug** | **Windows** | **Immediate**) or in the Output window (**Debug** | **Windows** | **Output**), depending on your settings.

By adding properties to your classes, you provide your application with easy access to object data. This allows the user interface, for example, to display and update the data.

Stateful Versus Stateless Classes

For some developers, myself included, it may seem unnatural at first to have properties defined for your business objects. Until recently, the best practice for Web applications and large-scale systems was to keep your business objects stateless so that they did not retain any property values between calls. The business objects consisted of only methods. Any data needed by those methods was passed in as parameters.

Stateful classes were deemed inappropriate for Web applications because there was no efficient way to maintain the values of the properties between calls to a page.

Stateful classes were deemed inefficient for large-scale systems with application servers because each time the user interface requested a property from the business object, a network hit was required to retrieve the data from the application server and pass it down. These were called "chatty" calls.

With the simplicity of deployment, all application components are now often deployed to the user's system, reducing the need for application servers. And many features have been added to simplify Web state management.

With many of the new features of .NET and today's architectural practices, it now makes sense to build stateful business objects. This opens the door for building objects that can easily support object binding.

Defining Methods

The **methods** of a class define the behavior and functionality associated with the class. Methods are implemented as subroutines and functions. For example, a `Product` class has `Create` and `Save` methods.

This section details the process of creating a method. It then covers some additional techniques for defining methods.

Creating a Method

Methods define the logic in your application. Create a method in a class for each set of business logic identified for the class during the design phase.

Implement a method using a subroutine when the method does not need to return a value, or a function if the method *does* need to return a value.

To define a method:

1. Open the class in the Code Editor.
2. Create the subroutine or function for the method.
   For example:

   ```
   Public Function Create
   ```

   Use good naming conventions for your method name. The recommended convention is to use the method's human-readable name, concatenating the words and using Pascal case, whereby each word in the name is capitalized.
   The purpose of this particular method is to create an instance of the business object class using the Factory pattern, so it was named `Create`. Some developers don't like to use that name because it could imply that a new item, such as a new product, is being created when instead an instance is created for an existing item. Alternatively, you could name this method `CreateInstance` or `GetProduct` or simply `Retrieve`.
3. Add the parameters appropriate for the method.
   For example:

   ```
   Public Function Create(ByVal prodID As Integer)
   ```

   Parameters define the data that is passed into or out of the function or subroutine. The number, name, and type of the parameters depend on the data that needs to be passed. In this example, the

product ID is passed in to create an object populated with data for the defined ID.

Use good naming conventions for your parameter names. The recommended standard is to use a logical parameter name, concatenating the words and using camel case, whereby the first letter is lowercase and the beginning of every other word is capitalized. Be sure that the parameter names do not conflict with any of your property names.

4. If you are defining a function, define the method's return type. For example:

```
Public Function Create(ByVal prodID As Integer) _
                                          As Product
```

The return type depends on the data that needs to be passed back from the function. In this example, the return type is an instance of the `Product` class.

5. Press the Enter key to automatically generate the method's remaining structure:

```
Public Function Create(ByVal prodID As Integer) _
                                          As Product


End Function
```

6. Add code within the method to perform the desired operation.
7. If you're implementing a `Function`, use the `Return` statement to return the value.

NOTE FOR VB6 DEVELOPERS: Use the `Return` statement instead of using the method's name to return a value.

The purpose of this particular `Create` method is to create an instance of the class. As discussed earlier in this chapter, objects are often created from a class using the Factory pattern. The `Create` method is then used, instead of the constructor, to create instances of the class.

A `Create` method used to create an instance of a class would look similar to this:

```
Public Function Create(ByVal prodID As Integer) As Product
    Dim prod As Product
```

```
    ' Create a new instance
    prod = New Product()

    ' Populate the object
    If prodID = 1 Then
        prod.ProductID = 1
        prod.ProductName = "Mithril Coat"
        prod.InventoryDate = #4/1/2006#
    End If

    Return prod
End Function
```

The first line of this function declares an object variable. The `New` keyword is then used to create a new instance of the `Product` class. The object properties are then populated. Notice that these are hard-coded in this case. The property values will be assigned from data in a database in Chapter 8. For now, the values are hard-coded so that the `Create` method works at this point without needing the data access layer in place just yet. The last line of the function returns the instantiated and populated `Product` object.

Although these steps demonstrate a `Create` method, you can create any type of method using these steps. Alternatively, you can use the Class Designer, as described in the next chapter, to assist you in defining the methods of your class.

Use methods to perform all of the processing required by your application. To create good methods, ensure that each method has a single purpose and that the method is no longer than about one page. If a method is long, break it into multiple methods. This makes each method much easier to build and maintain.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add the **Create** method, as defined in this section.
- Open the **ProductWin** form in the Code Editor.
- In the **Load** event for the **ProductWin** form, modify the code to call the **Create** method and display the resulting values:

```
Dim prod as Product
prod = New Product
prod = prod.Create(1)
Debug.WriteLine(prod.ProductID)    ' Displays 1
Debug.WriteLine(prod.ProductName)
                        ' Displays Mithril Coat
Debug.WriteLine(prod.InventoryDate)
                        ' Displays 4/1/2006 12:00:00 AM
```

Run the application. It displays your splash screen and then shows the MDI parent form. Select **Products** | **Manage Products** to display the ProductWin form. The debug messages appear in the Immediate window (**Debug** | **Windows** | **Immediate**) or in the Output window (**Debug** | **Windows** | **Output**), depending on your settings.

Passing Parameters

Parameters to methods are passed either `ByVal` or `ByRef`. `ByVal` is short for "by value" and means that the parameter value is evaluated and then its value is passed to the method. `ByRef` is short for "by reference" and means that a reference to the parameter is passed to the method.

If you don't specify the passing mechanism, the default is `ByVal`. In most cases, you want to pass your parameters by value.

NOTE FOR VB6 DEVELOPERS: The default in the classic versions of Visual Basic was `ByRef`, so watch for this when converting from VB6 to Visual Basic 2005.

The only time you need to use `ByRef` is when you want to modify the parameter within the method and allow the calling code to receive the modified value upon return from the method call. `ByRef` can also be used to return parameters from the method if you need more than one return value.

For example, a `ProcessRequest` method needs to return the number of items processed and a response string to the calling code. The method signature uses the `ByRef` keyword as follows:

```
Private Function ProcessRequest(ByVal requestType As String, _
    ByRef requestResponse As String) As Integer
     Dim itemsProcessed As Integer = 0

     ' Code that performs the request

     requestResponse = "Test Reply"
     Return itemsProcessed
End Function
```

This function is called as follows:

```
Dim requestCount As Integer
Dim response As String
requestCount = ProcessRequest("Test", response)
Debug.WriteLine(requestCount.toString & " " & response)
                                              ' Displays 0 Test Reply
```

If the `requestResponse` parameter was declared using the `ByVal` keyword, the `response` variable would always return `Nothing`, because it would not pass back the changed value from the function. By using the `ByRef` keyword, the `response` variable is set to the changed value.

Documenting the Method

It is always a good idea to add documentation for a method immediately after creating the method. You may even want to add the documentation just after defining the method signature and before you write the code within the method. By adding the documentation right away, you focus on the method's purpose, which helps you keep the method encapsulated. It is also much easier to document each method as you go along instead of facing the large task of going back later and documenting all the methods.

To document the method:

1. Open the class in the Code Editor.
2. Move the insertion point immediately before the word `Public` in the `Public Function` or `Public Sub` statement.
3. Type three comment markers, defined in Visual Basic as apostrophes (`'''`), and press the Enter key.

The XML comments feature automatically creates the structure of your method documentation as follows:

```
''' <summary>
'''
''' </summary>
''' <param name="prodID"></param>
''' <returns></returns>
''' <remarks></remarks>
```

Notice how this automatically generates a `param` tag with the name of each method parameter.

NOTE: If you type the three comment markers in the empty line above the method definition instead of on the same line as the method definition, you don't need to press the Enter key to generate the documentation structure.

---

4. Type a summary of the method between the `summary` tags, the parameter descriptions between the `param` tags, and so on. Your documentation may be similar to this:

```
''' <summary>
''' Creates a populated instance of this class
''' </summary>
''' <param name="prodID">ID of the product to
''' create</param>
''' <returns>Instance of the Product class</returns>
''' <remarks></remarks>
```

Use the `summary` tags to describe the method's purpose and the `param` tags to define each parameter. The `summary` and `param` are the most important tags because they are used by Visual Studio.

When you provide method documentation using XML comments, your method displays documentation about itself in appropriate places within Visual Studio. For example, the documentation appears in the Intellisense List Members box when you type the object variable name and a period (`.`).

Using XML comments to document your methods makes it easier for you and other developers to work with your methods.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add documentation for the **Create** method using XML comments.

In the Code Editor, view the comment by typing `prod.` in the **Load** event in the **ProductWin** form. When you type the period (`.`), Visual Studio displays the Intellisense List Members box, showing all the properties and methods for the class. Click the `Create` method to see the XML comments. Be sure to remove this code; otherwise, the project will have a syntax error.

Overloading Methods

There may be times when you want to have different sets of parameters for a method. For example, you may want your `Create` method to accept an ID or string name. Or you may want a `Retrieve` method to work with no parameters to retrieve all the data or an ID to retrieve data for a particular ID.

You could define different method names to support different signatures, but a better way is to use overloaded methods. An **overloaded method** is a method that has the same name as another method but different parameters.

For example, the `Create` method defined in this section has a parameter for a product ID. If you want to allow creating objects by name as well, you could define an overloaded `Create` method as follows:

```
Public Function Create(ByVal prodName As String) As Product
    Dim prod As Product

    ' Create a new instance
    prod = New Product()

    ' Populate the object
    '...

    Return prod
End Function
```

A method can have any number of overloads, each with different sets of parameters. The parameters are evaluated based on the number and

type of parameters, not the parameter names. So if you defined a `Create` method with a product name string parameter, you could not add a `Create` method with a description string parameter. This is because when you call the function and pass a string, the .NET runtime would not be able to tell which `Create` method you want to execute. Each overload must have a unique set of parameters.

Overloading is also great for enhancing your methods. For example, suppose you originally created a `Create` method with one parameter. You then need to add a `withComments` parameter to define whether to populate comment information. If code in your application calls the original `Create` method you don't want to break that code by adding a new parameter. Instead, you can create an overload for the `Create` method with the new parameter without needing to modify any existing code that calls the original method:

```
Public Function Create(ByVal prodName As String, _
   ByVal withComments As Boolean) As Product
    Dim prod As Product

    ' Create a new instance
    prod = New Product()

    ' Populate the object
    '...

    If withComments Then
        ' also populate the comments
    End If

    Return prod
End Function
```

In many cases, the code you need to execute in each of the overloaded methods is similar. So a common technique is to have one overload call the other. So the `Create` method from the prior code example could be changed to the following:

```
Public Function Create(ByVal prodName As String) As Product
    Return Create(prodName, False)
End Function
```

The overload with one parameter simply calls the other overload, passing a default value for the `withComments` flag. In most cases, the majority of the code is in the overload with the most parameters.

Each overload appears in the Intellisense Parameter Info, as shown in Figure 5.2.

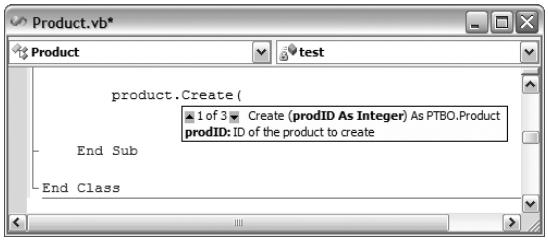

Figure 5.2 The 1 of 3 advises you that this method has three overloads. Use the up and down arrows to show the Parameter Info for each overloaded method.

Use overloading any time you want to define methods with the same name but different method signatures. Be sure that the signatures differ in the number or type of parameters.

Building Along

You can skip this "Building Along" without impacting any "Building Along" activities in later chapters. However, these overloads do appear in future screen shots to demonstrate how overloaded functions appear.

If you want to try out overloading, for the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add the two **Create** method overloaded functions as described in this section.

In the Code Editor, view the overloads in Intellisense as shown in Figure 5.2.

Defining Shared Methods

Shared methods, sometimes called **static methods**, are methods that are shared between all the instances of a class. They do not require that you create an object before calling the method.

For example, if you want to display something to the debug window, you don't first create an instance of the `Debug` class and then use an object variable to call the `WriteLine` method. Instead, you call the `WriteLine` method directly for the class itself. The `WriteLine` method is a shared method.

You define shared methods in your classes using the `Shared` keyword on the method signature. For example:

```
Public Shared Function Create(ByVal prodID As Integer) As Product
    ...
End Function
```

When a method is shared, you no longer need to create an object from the class before using the method. So instead of using code that looks like this:

```
Dim prod as Product
prod = New Product
prod = prod.Create(1)
```

the code instead looks like this:

```
Dim prod as Product
prod = Product.Create(1)
```

Notice that the code does not create an instance and uses the class name (`Product` in this example) instead of the object variable name (`prod`) to call the method. This is because a shared method cannot be accessed using an instance.

NOTE: If you do try to call a shared method using an object variable instead of the class name, you get a warning stating that you cannot access a shared member through an instance.

The most common use of shared methods is for Factory pattern methods, as shown in the `Create` method example, and for function libraries. The .NET Framework makes extensive use of shared methods in its function libraries, such as the `WriteLine` method in the `Debug` class.

You can also use the `Shared` keyword on properties to share a property across all instances. This is useful for properties such as a count that needs to be aware of all instances.

When defining a shared property or method, keep the following in mind:

- A shared property or method cannot reference nonshared properties.
  In the `Create` method example, the shared method created an instance of the class and used that instance to reference the properties. It cannot access the properties without an instance, because those properties are not shared. The properties are unique for each instance.
- A shared property or method cannot reference a nonshared method of the class.
  If you need to call a nonshared property or method of the class within the shared property or method, you can create an instance of the class and use that instance to call the nonshared property or method.
- `Me` is not valid within a shared property or method.
  `Me` references the current running instance, and a shared property or method does not have an instance.

Use the `Shared` keyword any time you want to define a property or method that is shared across all instances of your class. Access shared properties and methods using the class name instead of an instance variable.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Add the `Shared` keyword to the **Create** method, as defined in this section.
- Uncomment the private constructor.
  The Factory pattern method is now in place, so your code should no longer create an instance of the class using the `New` keyword.
- Open the **ProductWin** form in the Code Editor.

- In the **Load** event for the **ProductWin** form, modify the code to call the **Create** method and display the resulting values:

```
Dim prod as Product
prod = Product.Create(1)
Debug.WriteLine(prod.ProductID)    ' Displays 1
Debug.WriteLine(prod.ProductName)
                        ' Displays Mithril Coat
Debug.WriteLine(prod.InventoryDate)
                        ' Displays 4/1/2006 12:00:00 AM
```

Run the application. It displays your splash screen and then shows the MDI parent form. Select **Products** | **Manage Products** to display the ProductWin form. The debug messages appear in the Immediate window (**Debug** | **Windows** | **Immediate**) or in the Output window (**Debug** | **Windows** | **Output**), depending on your settings.

Obsolescing Methods

Code changes over time. Methods that you created today may no longer be needed tomorrow. But if you delete them, every piece of code that calls the method needs to be changed. Depending on the features you are implementing, this may be necessary. But in many cases, you can define a smoother obsolescence plan.

Obsolescence is the concept in which methods become obsolete over time. Instead of changing a method for a new feature, you add a method overload to support the new feature, essentially making the original method obsolete. That way, you need to change only the code required for the new feature, not every piece of code that calls the original method. You can then obsolete the original method so that you don't forget to remove it at a later point in time.

In looking at the `Create` method example from earlier in this chapter, you defined the method with a product name parameter. Suppose you later find that you need to sometimes manage comment information. So you add an overloaded method with a flag defining whether to handle comment information. You want every call to the method to ultimately call the new method signature. But in the interim, by having the original method remain in place, any unchanged code still works.

To identify a method as obsolete, use the `Obsolete` attribute. An **attribute** is metadata that you can associate with programming elements such as classes, properties, and methods. Attributes are defined in Visual Basic using less-than (<) and greater-than (>) signs. Attributes must be defined on the same line as the declaration of the class, property, or method to which the attribute is assigned. Use the line-continuation character (_) to separate the attribute from the declaration so that they are easier to read.

For example, to define one overload of the `Create` method as obsolete, add the `Obsolete` attribute to the method:

```
<ObsoleteAttribute( _
  "Use the Create(prodName, withComments) instead", False)> _
Public Shared Function Create(ByVal prodName) As Product
    Return Create(prodID, False)
End Function
```

The attribute's name can be defined with or without the "Attribute" suffix; either `Obsolete` or `ObsoleteAttribute` can be used. Your coding standards may define that the suffix is included for clarity or not included for brevity. Either way, be consistent with all attributes.

Some attributes, such as the `Obsolete` attribute, have parameters. The first parameter in this case is a message to any developer using your obsolete method, and the second parameter is an error flag. This message appears in the Error List window (see Figure 5.3) as a warning if the second parameter is `False`, or as an error if the second parameter is `True`. This gives the developer using the method a warning or error, depending on your standard method obsolescence path.

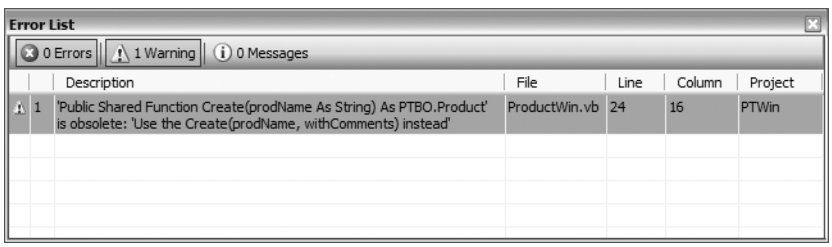

Figure 5.3 When you define a property or method as obsolete, any developer using the method knows that the property or method is on the obsolescence path.

Define a standard obsolescence plan for your application. This plan defines when properties and methods are made obsolete, how long they should be obsolete in a warning mode, and at what point they should be marked with a compile-time error. This provides a phased approach to modifying your application.

Building a Base Business Object Class

Business objects have standard housekeeping tasks that they must perform. For example, they must keep track of their state (unchanged, added, modified, deleted). The purpose of a base business object class is to define a standard set of operations that are applicable to all business objects. This keeps the housekeeping code out of the business objects themselves.

Creating base classes was covered in detail in Chapter 4, which demonstrated how to build a base form class. This section provides information on building a base business object class. For all the definitions, benefits, and techniques of building a base class, see Chapter 4.

Creating the Base Business Object Class

The primary code in a business object base class is housekeeping code—code that manages the object state, whether it is "dirty" (meaning changed), and whether it is valid. The base business object class performs any task that is common for all the business objects.

To create a base business object class:

1. Add a project item to your business object Class Library project using the **Class** template.
   If you created your own class template, you can use it here.
   Use a clear name for the base business object class. This helps you (and your project team) keep track of the base class.
2. Add code as desired.
   Add any code that is common to the business objects to the base business object class.

As an example, the base business object class can keep track of the object state. The code required for this has three parts. First, the set of valid business object states must be defined. Then one or more properties must be created to expose the state. Finally, a method is needed to manage the state.

The set of valid business object states can be implemented using an enumeration, defined with the `Enum` keyword. An **enumeration** defines a set of named constants whose underlying type is an integer. You can define the integer assigned to each constant; otherwise, the enumeration sets each constant to a sequential integer value starting with 0.

For example, the business object state values are defined in an enumerated type as follows:

```
Public Enum EntityStateEnum
    Unchanged
    Added
    Deleted
    Modified
End Enum
```

In this example, the value of `Unchanged` is 0, `Added` is 1, and so on. Any variable declared to be of this enumeration type can be assigned to one of these defined constants.

A business object's state is exposed by defining a property that gets and sets the object's state. For example, an `EntityState` property could be defined as follows:

```
Private _EntityState As EntityStateEnum
''' <summary>
''' Gets the business object state
''' </summary>
''' <value>Unchanged, Added, Deleted, or Modified</value>
''' <returns>Value identifying the entity's state</returns>
''' <remarks></remarks>
Protected Property EntityState() As EntityStateEnum
    Get
        Return _EntityState
    End Get
    Private Set(ByVal value As EntityStateEnum)
        _EntityState = value
    End Set
End Property
```

Notice that the property uses the `Protected` keyword to ensure that it can be accessed only by classes that inherit from this base class. The setter uses the `Private` keyword to ensure that code outside of the class cannot modify the entity's state.

You may want to define other properties that expose the object state in different ways. For example, it is common for a business object to have a Boolean `IsDirty` property that identifies whether an entity has been changed. Although the `EntityStateEnum` could be used to determine this, adding an `IsDirty` property provides a shortcut:

```
''' <summary>
''' Gets whether the business object has changes
''' </summary>
''' <value>True or False</value>
''' <returns>True if there are unsaved changes;
''' False if not</returns>
''' <remarks></remarks>
Protected ReadOnly Property IsDirty() As Boolean
    Get
        Return Me.EntityState <> EntityStateEnum.Unchanged
    End Get
End Property
```

This property does not have its own private backing variable. Instead, it uses the value of the `EntityState` property.

You also need code that manages the state. This is normally implemented as a method:

```
''' <summary>
''' Changes the state of the entity
''' </summary>
''' <param name="dataState">New entity state</param>
''' <remarks></remarks>
Protected Sub DataStateChanged(ByVal dataState As EntityStateEnum)
    ' If the state is deleted, mark it as deleted
    If dataState = EntityStateEnum.Deleted Then
        Me.EntityState = dataState
    End If

    ' Only set data states if the existing state is unchanged
    If Me.EntityState = EntityStateEnum.Unchanged _
       OrElse dataState = EntityStateEnum.Unchanged Then
        Me.EntityState = dataState
    End If
End Sub
```

This code sets the state appropriately. This is not as simple as just assigning the state to the value passed in to the method, because some states cannot be changed. For example, if the state is already defined to be `Added`, further changes to the object leave the state as `Added`. And if the state is `Deleted`, it does not matter which other state it was; it needs to be deleted.

In your code, call `DataStateChanged` with a state of `Added` when the user creates a new item. Call `DataStateChanged` with a state of `Deleted` when the user deletes an item. Call `DataStateChanged` with a state of `Modified` whenever the user changes any of the data associated with an object. Because you defined all your object data with properties, you can add the call to `DataStateChanged` to the setter for each property, as described in the next section.

Building a base business object class keeps the majority of the housekeeping code out of the business object class itself and lets you focus on the unique business rules and business processing code required for the specific business object.

Building Along

For the Purchase Tracker sample application:

- Add a class project item to the business object Class Library project (**PTBO**) using the **Class** template.
  If you created your own class template using steps from Chapter 3, you can use your class template here.
  Name the class **PTBOBase**.
  If not already added by the selected template, add the standard set of regions to the class as described earlier in this chapter.
- Add documentation to the class using XML comments.
- Add the code defined in this section.

You now have an operational base business object class that ensures the business objects from all derived classes consistently handle their state. But at this point, the base class does not actually do anything. To make use of the base class, you need to inherit from it, as described in the next section.

Inheriting from the Base Business Object Class

After you create a base business object class, you use it by inheriting from it. Each business object class that needs to manage its state can inherit from the base business object class. The business object then has access to the properties and methods from the base business object class.

The `Inherits` keyword specifies that a class inherits from another class. Add the `Inherits` keyword to any business object class as follows:

```
Public Class Product
    Inherits PTBOBase
```

The class, in this case `Product`, then has all the properties and methods from the base business object class. You can easily see this by typing `Me.` somewhere within a property or method of the `Product` class. The Intellisense List Members box displays properties and methods of both the base class (`PTBOBase`) and the derived class (`Product` in this case).

To take advantage of the code in the base business object class, the derived classes can use the properties and methods of the base class. For example, when a property in the business object is changed, the code calls the `DataStateChanged` method in the base business object class to correctly set the business object state.

The code in the `ProductName` property provides an example:

```
Public Property ProductName() As String
    Get
        Return _ProductName
    End Get
    Set(ByVal value As String)
        If _ProductName <> value Then
            Dim propertyName As String = "ProductName"
            Me.DataStateChanged(EntityStateEnum.Modified)
            _ProductName = value
        End If
    End Set
End Property
```

NOTE: This code does not currently use the `propertyName` variable. It is used later when validation code is added in Chapter 7.

The `Dim` statement declaring the `propertyName` variable could be a `Const` statement instead since the property name does not change within the property.

The setter code first determines whether the value is the same as it was. If so, it does not reset it. If the value is indeed changed, the setter sets a variable for the property's name. The `DataStateChanged` method in the base business object class is then called and passed a state of `Modified`. Finally, the property value is changed to the passed-in value.

In every derived class, modify each updatable property to include similar code. When any property value changes, the object is marked as modified. This ensures that each object is aware of its state so that it can react accordingly.

Building Along

For the Purchase Tracker sample application:

- Open the **Product** class in the Code Editor.
- Modify the **Product** class to inherit from the base business object class (**PTBOBase**), as demonstrated in this section.
- Modify the setter for each *updatable* property defined in the **Product** class, to include a call to `DataStateChanged` as shown in this section.

NOTE: `ProductID` is not updatable because its setter is private. So it should not call the `DataStateChanged` method.

NOTE: Recall that the `InventoryDate` is a `Nullable` type. `Nullable` types do not support the not-equal (<>) operator. So to check the value of the `InventoryDate` property against the value passed in, you need to use some additional code:

```
If (_InventoryDate.HasValue<>value.HasValue) OrElse _
      (_InventoryDate.HasValue AndAlso _
      value.HasValue AndAlso _
      InventoryDate.Value <> value.Value) Then
    Dim propertyName As String = "InventoryDate"
```

```
    Me.DataStateChanged(EntityStateEnum.Modified)
    _InventoryDate = value
End If
```

This code first determines if the property has a value. The code changes the value only if the HasValue changes or if it has a current value and that value has changed.

- Modify the **Create** method to reset the object's entity state to `Unchanged` after it sets the property values. This ensures that the entity state tracks the user's changes, not changes made to the properties when they are first populated.
  Add the following code as the last line of the `Create` method:
  `prod.DataStateChanged(EntityStateEnum.Unchanged)`

Overriding Base Class Members

Sometimes the derived class needs to modify the functionality of one of the base class members. When this is required, you can **override** the base class member by implementing the property or method in the derived class. When the property or method is called, the implementation in the derived class overrides the implementation from the base class.

For example, say that one business object requires some additional processing in the base class `DataStateChanged` method. To override this method, implement the method in the business object using the exact same method signature:

```
''' <summary>
''' Changes the state of the entity
''' </summary>
''' <param name="dataState">New entity state</param>
''' <remarks></remarks>
Protected Sub DataStateChanged(ByVal dataState As EntityStateEnum)
    MyBase.DataStateChange(dataState) ' Performs base processing

    ' Do unique code
    ...
End Sub
```

Notice that this code calls the base business object class to perform its processing and then performs its unique processing. It could instead perform its processing first and then call the base business object class. Or it can do all of its own processing.

Use overriding whenever the derived class needs its own implementation of a property or method in the base class.

Conclusion

What Did This Chapter Cover?

This chapter described how to build code in the business logic layer for your application. It provided information on defining properties, coding methods, using generics, and building a base business object class.

This chapter covered several real productivity enhancers:

- Writing class documentation using the XML documentation feature makes it easy to create the class's documentation. That documentation is then available in many places within Visual Studio, making it easier for you or your team to work with the class's properties and methods.
- Defining regions helps you focus on the code you are working on.
- Using partial classes for generated code makes it easier to regenerate the generated code without impacting the custom code.
- Handling nulls using the generic `Nullable` structure simplifies working with value types and null values.
- Defining a Factory pattern method or class library method with the `Shared` keyword makes it easy to call the method, because you don't need to create an instance of the class.
- Building a base business object class provides standardized processing for all your application's business objects and significantly reduces the amount of housekeeping code required in each business object class.

The next chapter provides additional tools and techniques for working with classes.

Building Along

If you are "building along" with the Purchase Tracker sample application, this chapter added the basic code you need for your business object component.

Since the user interface from the preceding chapter does not yet reference any information in the business object component, running the application provides the same results as at the end of the preceding chapter.

The next chapter adds functionality and unit testing to the business object component of the Purchase Tracker sample application using some of the new Visual Studio 2005 tools and techniques.

Additional Reading

Cwalina, Krzysztof, and Brad Abrams. *Framework Design Guidelines*. Upper Saddle River, NJ: Addison Wesley, 2006.

This is an *excellent* book for any .NET developer. It provides general guidelines and many specific recommendations for handling everything from naming conventions to base classes to exceptions.

Lhotka, Rockford. *Expert VB 2005 Business Objects*, Second Edition. APress, 2006.

This book demonstrates how to build a framework for business objects that handles all the complex issues of .NET. It then shows you how to build Windows Forms, Web Forms, and Web Services interfaces on top of the objects, using all the data binding and other productivity features built into .NET 2.0 and Visual Studio 2005.

Richter, Jeffrey. "Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework." *MSDN* magazine, November 2000.

This article provides details on the .NET garbage collector.

Try It!

Here are a few suggestions for trying some of the techniques presented in this chapter:

1. Add a `SalesRep` business object class to the business object Class Library project.
   Ensure that the class inherits from the base business object class. Add properties for name, employee number, and so on. Add a `Create` method using the same Factory pattern defined in this chapter. Add other methods as appropriate.
2. Add an exception handler class, such as `SalesRepNotFoundException`, to the `SalesRep` class code file using an exception class.
   Throw the exception in the `Create` method if the ID passed into the method is not the ID you hard-coded data for.
3. Add several overloads for the `Create` method in the `SalesRep` class.
4. Make one of the overloads obsolete using the techniques presented in this chapter.