3 OBJECT-ORIENTED PROGRAMMING FUNDAMENTALS

IN BRIEF

With the introduction of a new set of object-oriented features, Visual Basic .NET has become a full-fledged object-oriented language. Visual Basic .NET provides built-in support for all the object-oriented concepts, including abstraction, encapsulation, polymorphism, and inheritance. In this chapter, you will learn the concepts of object-oriented programming and get a complete understanding of how to implement these concepts in Visual Basic .NET.

WHAT YOU NEED

SOFTWARE REQUIREMENTS	Windows 2000, XP, or 2003 .NET Framework 1.1 SDK Visual Studio .NET 2003 with Visual Basic .NET installed
HARDWARE REQUIREMENTS	PC desktop or laptop
SKILL REQUIREMENTS	Basic knowledge of .NET Framework

OBJECT-ORIENTED PROGRAMMING FUNDAMENTALS AT A GLANCE

Object-Oriented Concepts

VB .NET provides a simplified object-oriented syntax and provides the core foundation for implementing object-oriented applications by supporting the four major defining concepts required for a language to be fully object oriented. Those four concepts are listed here:

- **Abstraction**—This is merely the capability of a language to create "black box" code—to create an abstract representation of a concept within a program. A `Customer` object, for instance, is an abstract representation of a real-world customer.
- **Encapsulation**—This is the concept of separation between interface and implementation. The idea is that you can create an interface (`Public` methods in a class) and, as long as that interface remains consistent, the application can interact with your objects. This remains true even if you entirely rewrite the code within a given method—thus, the interface is independent of the implementation. Encapsulation enables you to hide the internal implementation details of a class. For example, the algorithm used to compute pi might be proprietary. You can expose a simple API to the end user, but you hide all of the logic used by the algorithm by encapsulating it within the class.
- **Polymorphism**—This is reflected in the capability to write one routine that can operate on objects from more than one class, treating different objects from different classes in exactly the same way. For instance, if both `Customer` and `Vendor` objects have a `Name` property, and you can write a routine that calls the `Name` property regardless of whether you're using a `Customer` or `Vendor` object, then you have polymorphism.
- **Inheritance**—VB .NET is the first version of VB that supports inheritance. Inheritance is the idea that a class can gain the pre-existing interface and behaviors of an existing class. This is done by inheriting these behaviors from the existing class through a process known as subclassing. With the introduction of full inheritance, VB is now a fully object-orientated language by any reasonable definition.

To start, the next section takes a look at how to create classes and how to create objects by creating instances of the classes.

Creating Classes

Using objects is fairly straightforward and intuitive; even the most novice programmers can pick this up and accept rapidly. Creating classes and objects is a bit more complex and interesting, however; that is covered in the next section.

Creating Basic Classes

As discussed earlier, objects are merely instances of a specific template (or a class). The class contains the code that defines the behavior of its objects and also defines the instance variables that will contain the object's individual data.

Classes are created using the `Class` keyword. A class includes definitions (declaration) and implementations (code) for the variables, methods, properties, and events that make up the class. Each object created based on this class has the same methods, properties, and events, and has its own set of data defined by the variables in the class.

The `Class` Keyword

If you want to create a class that represents a `Customer`, you can use the `Class` keyword:

```
Public Class Customer
  'Implementation code goes here
End Class
```

When you create a new Windows application in Visual Basic using Visual Studio .NET, by default, the project is composed of a set of files with the `.vb` extension. Each file can contain multiple classes. This means that, within a single file, you could have something like this:

```
Public Class Person
  'Implementation code goes here
End Class

Public Class Employee
  'Implementation code goes here
End Class

Public Class Dept
  'Implementation code goes here
End Class
```

Even though you can have multiple classes in a single file, the most common approach is to have a single class per file. This is because the VS .NET Solution Explorer and the code-editing environment are tailored to make it easy to navigate from file to file to find your code. For instance, if you create a single class file with all these classes, Solution Explorer simply shows a single entry, as shown in Figure 3.1.

However, the VS .NET IDE provides the Class View window. If you decide to put multiple classes in each physical `.vb` file, you can use the Class View window to quickly and efficiently navigate through your code, jumping from class to class without having to manually locate those classes in specific code files. To bring up the Class View window, select Class View from the View menu from within Visual Studio .NET. This is shown in Figure 3.2.

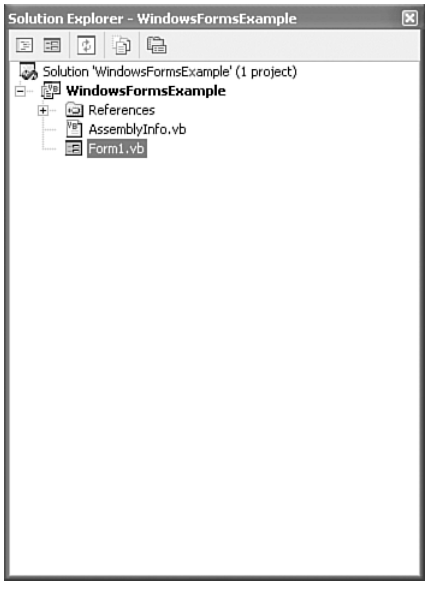

FIGURE 3.1 Classes displayed in Solution Explorer.

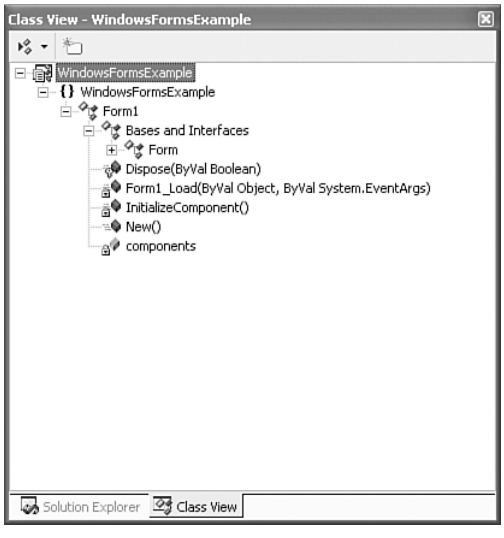

FIGURE 3.2 Showing classes in the Class View window.

The Class View window shown in Figure 3.2 is incredibly useful, even if you keep one class per file, because it provides you with a class-based view of your entire application.

Creating a Class Using Visual Studio .NET

To start, create a new Visual Basic .NET Windows application by selecting File, New Project from the menu. In the New Project dialog box, select Visual Basic Projects as the project type and Windows Application as the template. After entering the name and location in the dialog box, click OK. When the project is created, you can add a new class by selecting `Add Class` from the `Project` menu. You are presented with the standard Add New Item dialog box. Change the name of the class to **Customer.vb**, and click Open. The result is the following code that defines the `Customer` class:


```
Public Class Customer


End Class
```

It is worth noting that all VB .NET source files end with a `.vb` extension, regardless of which type of VB source file you choose (form, class, module, and so on) when you are adding the file to the project. In fact, any forms, classes, components, or controls that you add to your project are actually class modules—they are just specific types of classes that provide the appropriate behaviors. The exception is the Module, which is a special construct that enables you to include code within your application that is not directly contained within any class. As with previous versions of Visual Basic, methods placed in a `Module` can be called directly from any code within a project.

Objects, Classes, and Instances

An object is a code-based abstraction of a real-world entity or relationship. For instance, you might have a `Customer` object that represents a real-world customer, or you might have a `File` object that represents a file on your computer's hard drive.

A closely related term is *class*. A class is the code that defines an object, and all objects are created based on a class. A class is an abstraction of a real-world concept, and it provides the basis from which you create instances of specific objects. For example, to have a `Customer` object, you must first have a `Customer` class that contains all the code (methods, properties, events, variables, and so on) necessary to create `Customer` objects. Based on that class, you can create any number of `Customer` objects. All the `Customer` objects are identical in terms of what they can do and the code they contain, but each one contains its own unique data. This means that each object represents a different physical customer.

Creating Constructors

In VB .NET, classes can implement a special method that is always invoked as an object is created. This method is called the constructor, and it is always named `New`.

The constructor method is an ideal location for such initialization code, since it is always run before any other methods are ever invoked—and it is always run only once for an object. Of course, you can create many objects based on a class, and the constructor method will be run for each object that is created.

The constructor method of a VB .NET class is similar to the `Class_Initialize` event in previous versions of Visual Basic, but it is far more powerful in VB .NET because you can also supply parameter values to the constructor.

You can easily implement a constructor in your class simply by implementing a public method named `New`. For example, add the following lines of initialization code to the constructor of the `Customer` class:

```
Public Class Customer
  Private _country as String
  Public Sub New()
    _country = "US"
  End Sub
  'Implementation code goes here
End Class
```

In this example, you are simply using the constructor method to initialize the country for any new `Customer` object that is created.

Parameterized Constructors

You can also use constructors to allow parameters to be passed to your object as it is being created. This is done by simply adding parameters to the `New` method. For example, you can modify the constructor method that was created in the previous section and make it look like as follows:

```
Public Sub New(ByVal countryName As String)
   _country = countryName
End Sub
```

With this change, any time a `Customer` object is created, you are provided with values for the country name parameter. This changes how you can create a new `Customer` object, however. Originally, you used code such as this:

```
Dim custObj As New Customer()
```

Now you have code such as this:

```
Dim custObj As New Customer("US")
```

In fact, because your constructor expects these values, they are mandatory: Any code that wants to create an instance of your `Customer` class must provide these values.

Fortunately, there are alternatives in the form of optional parameters and method overloading (enabling you to create multiple versions of the same method, each accepting a different parameter list).

Constructors with Optional Parameters

In many cases, you want a constructor to accept parameter values for initializing new objects, but you also want to have the capability to create objects without providing those values. This is possible by using optional parameters.

Optional parameters on a constructor method follow the same rules as optional parameters for any other `Sub` routine: They must be the last parameters in the parameter list, and you must provide default values for the optional parameters.

For instance, you can change your `Customer` class as shown:

```
Public Sub New(Optional ByVal country As String = "US")
  _country = country
End Sub
```

Here you changed the country parameter to be optional and provided a default value for the country. With this constructor in place, you have the option of creating a new `Customer` object with or without the parameter values:

```
Dim custObj As New Customer("US")
```

or

```
Dim custObj As New Customer()
```

If you don't provide a value for the country parameter, the default value of `US` will be used and your code will work just fine.

Composition of an Object

You use an interface to get access to an object's data and behavior. The object's data and behaviors are contained within the object, so a client application can treat the object like a black box accessible only through its interface. This is a key object-oriented concept called encapsulation. The idea is that any programs that make use of this object won't have direct access to the behaviors or data; instead, those programs must make use of the object's interface.

This list walks through each of the elements related to object in detail:

- **Interface**—The interface is defined as a set of methods (`Sub` and `Function` routines), properties (`Property` routines), events, and fields (variables or attributes) that are declared `Public` in scope.

- **Attribute**—*Attribute* means one thing in the general object-oriented world and something else in .NET. The OO world often refers to an object's variables as attributes, whereas, in .NET, an attribute is a coding construct that you can use to control compilation, the IDE, and so on.
- **Method**—This is the place where you actually write code that operates on the data. You can also have `Private` methods and properties in your code. Although these methods can be called by code within your object, they are not part of the interface and cannot be called by programs written to use your object. Another option is to use the `Friend` keyword, which defines the scope to be the current project; this means that any code within your project can call the method, but no code outside your project (that is, from a different .NET assembly) can call the method. To complicate things a bit, you can also declare methods and properties as `Protected`, which are available to classes that inherit from your class.

For example, you might have the following code in a class:

```
Public Function ReturnResult() As Integer
End Function
```

Because this method is declared with the `Public` keyword, it is part of your interface and can be called by client applications that are using your object. You might also have a method such as this:

```
Private Sub DoSomething()
End Sub
```

This method is declared as being `Private` and so is not part of your interface. This method can be called only by code within your class, not by any code outside your class, such as the code in a program that is using one of your objects.

On the other hand, you can do something like this:

```
Public Function ReturnResult() As Integer
  DoSomething()
End Function
```

In this case, you are calling the `Private` method from within a `Public` method. Although code using your objects can't directly call a `Private` method, you will frequently use `Private` methods to help structure the code in your class to make it more maintainable and easier to read.

Finally, you can use the `Friend` keyword:

```
Friend Sub DoSomething()
End Sub
```

In this case, the `DoSomething` method can be called by code within your class or from other classes or modules within your current VB .NET project. Code from outside the project will not have access to the method.

The `Friend` scope is very similar to the `Public` scope, in that it makes methods available for use by code outside your object itself. However, unlike `Public`, the `Friend` keyword restricts access to code within your current VB .NET project, preventing code in other .NET assemblies from calling the method.

Implementation or Behavior

The code inside of a method is called the implementation. Sometimes it is also called behavior because this code actually makes the object do useful work. For instance, you might have a `Name` property as part of your object's interface. Within that property, you can write code to combine the first name and last name and return the name:

```
Public Class Customer
  Private _firstName as String
  Private _lastName as String
  Public ReadOnly Property Name() As Integer
    Get
      Return _firstName + "" + _lastName
    End Get
  End Property
End Class
```

In this example, the code returns the full name by combining the first name and the last name. The key concept here is to understand that client applications can use your object even if you change the implementation—as long as you don't change the interface. As long as your method name and its parameter list and return data type remain unchanged, you can change the implementation of the `Name` property all you want.

The code necessary to call the `Name` property looks something like this:

```
Dim name as String = cust.Name
```

The result of running this code is the `Name` value returned for your use. Although your client application will work fine, you might suddenly discover that you want to return the name in a different format, meaning that you need to improve upon the code. Fortunately, you can change your implementation without changing the client code:

```
Public Class Customer
  Private _firstName as String
  Private _lastName as String
  Public ReadOnly Property Name() As Integer
```

```
    Get
      Return _lastName + ", " + _firstName
    End Get
  End Property
End Class
```

In this code, the implementation of the `Name` property is changed behind the interface without changing the interface itself. Now if run your client application, you will find that `Name` property returns the new format without breaking the client.

NOTE

It is important to keep in mind that encapsulation is a syntactic tool: It allows your code to continue to run without change. However, it is not semantic. Just because your code continues to run doesn't mean that it continues to do what you actually wanted it to do.

Member or Instance Variables

The third key part of an object is its data, or state. In fact, it might be argued that the only important part of an object is its data. After all, every instance of a class is identical in terms of its interface and its implementation—the only thing that can vary is the data contained within that particular object.

Member variables are declared so that they are available to all code within your class. Typically, member variables are `Private` in scope, meaning that they are available only to the code in your class itself. They are also sometimes referred to as instance variables or attributes. The .NET Framework refers to them as fields.

TIP

Don't confuse instance variables with properties. In VB, a property is a type of method that is geared toward retrieving and setting values, whereas an instance variable is a variable within the class that can hold the value exposed by a property.

For instance, you might have a class that has instance variables such as first name and last name, as shown in the following code:

```
Public Class Customer
  Private _firstName As String
  Private _lastName As String
End Class
```

Objects, Classes, and Instances

Each instance of the class (each object) will have its own set of these variables in which to store data. Because these variables are declared with the `Private` keyword, they are available only to code within each specific object. Although member variables can be declared as `Public` in scope, this makes them available to any code using your objects in a manner that you can't control. Such a choice directly breaks the concept of encapsulation because code outside your object can directly change data values without following any rules that might otherwise be set in your object's code.

If you want to make the value of an instance variable available to code outside your object, you should use a property and let the client application invoke the property to set or get the value of the instance variable. Listing 3.1 shows the implementation of the `Customer` class and demonstrates how properties can be used to access the private variables.

LISTING 3.1

`Customer` Class with Properties

```
Public Class Customer
  Private _city As String
  Private _zip As Date
1 Public Property City() As String
    Get
      Return _city
    End Get
    Set
      'Add code to validate the value of the city
      _city = value
    End Set
  End Property

2 Public Property Zip() As String
    Get
      Return _zip
    End Get
    Set
      'Add code to validate the zip code
      _zip = value
    End Set
  End Property
End Class
```

Because the private City 1 and ZIP 2 attributes are accessed through the properties, you are not directly exposing your internal variables to the client application, thereby preserving encapsulation of data. At the same time, through this mechanism, you can safely provide access to your data as needed.

TIP

Member variables can also be declared with `Friend` scope, meaning that they are available to all code in your project. As with declaring them as `Public`, this breaks encapsulation and is strongly discouraged.

Now that you have a grasp of some of the basic object-oriented terminology, you are ready to explore the creation of classes and objects. First, you will see how VB enables you to interact with objects. Then you will move on to the actual process of authoring those objects.

Working with Objects

In the .NET environment—and within VB, in particular—you use objects all the time without even thinking about it. Every control on a form—and, in fact, every form—is an object. When you open a file or interact with a database, you are using objects to do that work.

Object Declaration and Instantiation

Objects are created using the `New` keyword, indicating that you want a new instance of a particular class. A number of variations exist for how or where you can use the `New` keyword in your code. Each one provides different advantages in terms of code readability or flexibility. The most obvious way to create an object is to declare an object variable and then create an instance of the object:

```
Dim cust As Customer
cust = New Customer()
```

The result of this code is a new instance of the `Customer` class ready for your use. To interact with this new object, you use the `cust` variable that you declared. The `cust` variable contains a reference to the object.

You can shorten this by combining the declaration of the variable with the creation of the instance:

```
Dim cust As New Customer()
```

In previous versions of VB, this was a very poor thing to do because it had both negative performance and maintainability effects. However, in VB .NET, there is no difference between the first example and this one, other than that this line of code is shorter.

The following code both declares the variable `cust` as the data type `Customer` and immediately creates an instance of the class that you can use from your code:

```
Dim cust As Customer = New Customer()
```

The previous syntax provides a great deal of flexibility while remaining compact. Even though it is a single line of code, it separates the declaration of the variable's data type from the creation of the object. Such flexibility is very useful when working with inheritance or multiple interfaces. You might declare the variable to be of one type—say, an interface—and instantiate the object based on a class that implements that interface. To demonstrate this, create an interface as shown here:

```
Public Interface ICustomer
  Sub GetCustomer()
End Interface
```


After you have created the Customer interface, you can implement that interface, meaning that your class now has its own native interface and also a secondary interface called `ICustomer`. This is how the declaration of the class looks:

```
Public Class Customer
  Implements ICustomer
  Public Sub GetCustomer() Implements ICustomer.GetCustomer
    'Implementation goes here
  End Sub
End Class
```

You can now create an instance of the `Customer` class but reference it via the secondary interface by declaring the variable to be of type `ICustomer`.

```
Dim cust As ICustomer = New Customer()
```

You can also do this using two separate lines of code:

```
Dim cust As ICustomer

cust = New Customer()
```

Either technique works fine and achieves the same result: You have a new object of type `Customer` that is accessed via its secondary interface.

Object References

Typically, when you work with an object, you are using a reference to that object. On the other hand, when you are working with simple data types such as `Integer`, you are working with the actual value rather than a reference. In this section, you explore these concepts and see how they work and interact.

When you create a new object using the `New` keyword, you store a reference to that object in a variable. For instance, this line of code creates a new instance of the `Customer` class:

```
Dim cust As New Customer()
```

To gain access to this new object, use the `cust` variable. This variable holds a reference to the object. When you have the reference, you can also do something like this:

```
Dim custObjRef As Customer
custObjRef = cust
```

Now you have a second variable, `custObjRef`, which also has a reference to that same object. You can use any one of these variables interchangeably because they both reference the exact same object. Remember, however, that the variable you have is not the object itself, but just a reference or pointer to the object itself.

Dereferencing Objects

When you are done working with an object, you can indicate that by dereferencing the object. To dereference an object, simply set your object reference to `Nothing`:

```
Dim cust As Customer
Cust = New Customer()
cust = Nothing
```

This code has no impact on your object itself. In fact, the object can remain blissfully unaware that it has been dereferenced for some time. After all the variables that reference an object are set to `Nothing`, the .NET runtime can tell that you no longer need that object. At some point, the runtime destroys the object and reclaims the memory and resources that the object consumes.

Between the time that you dereference the object and the time that .NET gets around to actually destroying it, the object simply sits in memory, unaware that it has been dereferenced. Right before .NET destroys the object, the framework calls the `Finalize` method on the object (if it has one).

Handling Events

Using methods and properties, you can write code that interacts with your objects by invoking specific functionality as needed. It is often useful for objects to provide notification as certain activities occur during processing. You can see examples of this with controls, with a button indicating that it was clicked via a `Click` event, or a text box indicating that its contents have changed via the `TextChanged` event.

Handling Events

Objects can raise events of their own, thereby providing a powerful and easily implemented mechanism by which objects can notify the client code of important activities or events.

TIP

Delegates are used to provide a mapping between the event and the event handlers that will respond to the event. They are the .NET equivalent of function pointers in C++.

Handling Single Events

Before you learn how to handle events, you first will add a button control to the form that you created in the previous sections. Double-click the button to bring up the following code in the code editor:

```
Private Sub button1_Click (ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click
End Sub
```

Typically, you just write your code in this routine without paying a lot of attention to the code that the VS .NET IDE creates. However, take a second look at that code here, to take note of a couple important things.

First, notice the use of the `Handles` keyword. This keyword specifically indicates that this method will be handling the `Click` event from the `button1` control. Of course, a control is just an object, so you are indicating here that this method will be handling the `Click` event from the `button1` object. Also notice that the method accepts two parameters. The `Button` control class defines these parameters. It turns out that any method that accepts two parameters with these data types can be used to handle the `Click` event. For instance, you could create a new method to handle the event:

```
Private Sub ButtonClickEvent(ByVal s As System.Object, _
  ByVal args As System.EventArgs) Handles button1.Click
End Sub
```

Even though you have changed the method name and the names of the parameters, you are still accepting parameters of the same data types, and you still have the `Handles` clause to indicate that this method will handle the event.

Handling Multiple Events

The `Handles` keyword offers even more flexibility. Not only can the method name be anything you choose, but a single method can handle multiple events, if you desire.

Again, the only requirement is that the method and all the events being raised must have the same parameter list. This explains why all the standard events that the .NET system class library raises have exactly two parameters: the sender and an `EventArgs` object. By being so generic, it is possible to write very generic and powerful event handlers that can accept virtually any event that the class library raises.

One common scenario in which this is useful is having multiple instances of an object that raises events, such as two buttons on a form:

```
Private Sub ButtonClickEvent(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles button1.Click, button2.Click
End Sub
```


Notice that you have modified the `Handles` clause to have a comma-separated list of events to handle. Either event causes the method to run, giving you a central location to handle these events.

The `WithEvents` Keyword

The `WithEvents` keyword tells VB that you want to handle any events raised by the object within your code:

```
Private WithEvents button1 As System.Windows.Forms.Button
```

The `WithEvents` keyword makes any events from an object available for your use, and the `Handles` keyword is used to link specific events to your methods so you can receive and handle them. This is true not only for controls on forms, but also for any objects that you create.

The `WithEvents` keyword cannot be used to declare a variable of a type that doesn't raise events. In other words, if the `Button` class didn't contain code to raise events, you would get a syntax error when you attempted to declare the variable using the `WithEvents` keyword. The compiler can tell which classes will and won't raise events by examining their interface. Any class that will be raising an event will have that event declared as part of its interface. In VB .NET, this means that you will have used the `Event` keyword to declare at least one event as part of the interface for your class.

Raising Events

Objects can raise events just like a control, and the client code that uses your object can receive these events by using the `WithEvents` and `Handles` keywords. Before you can raise an event from your object, however, you need to declare the event within your class by using the `Event` keyword.

Handling Events

In your `Customer` class, for instance, you might want to raise an event any time the `LoadCustomer` method is called. For instance, if you call this method `Loaded`, you can add the following declaration to your `Customer` class:

```
Public Class Customer
  Private _firstName As String
  Private _secondName As String

  Public Event Loaded()

  Public Sub LoadCustomer()
    'Implementation goes here
  End Sub
End Class
```


Events can also have parameters so that you can supply values to the code receiving the event. A typical button's `Click` event receives two parameters, for instance. In the `Loaded` method, perhaps you want to also indicate the number of customers that were loaded. You can do this by changing the event declaration:

```
Public Event Loaded(ByVal count As Integer)
```

Now that the event is declared, you can raise that event within your code, where appropriate. In this case, you raise that within the `LoadCustomer` method so that any time a `Customer` object is used to load customers, it fires an event indicating the number of customers loaded. Modify the `LoadCustomer` method to look like the following:

```
Public Sub LoadCustomer()
  'Add code to load the customer objects into a result set
  Dim count as Integer = 10   'Assign the number of loaded  customers to the count
  ➥variable
  RaiseEvent Loaded(count)
End Sub
```

The `RaiseEvent` keyword is used to raise the actual event. Because the event requires a parameter, that value is passed within parentheses and is delivered to any recipient that handles the event. In fact, the `RaiseEvent` statement causes the event to be delivered to all code that has the `Customer` object declared using the `WithEvents` keyword with a `Handles` clause for this event, or any code that has used the `AddHandler` method.

If more than one method will be receiving the event, the event will be delivered to each recipient one at a time. The order of delivery is not defined, so you can't predict the order in which the recipients will receive the event, but the event will be delivered to all handlers. Note that this is a serial, synchronous process.

The event is delivered to one handler at a time, and it is not delivered to the next handler until the current handler is complete. After you call the `RaiseEvent` method, the event is delivered to all listeners, one after another, until it is complete; there is no way for you to intervene and stop the process in the middle.

Receiving Events

So far, you have seen how to raise events using the `RaiseEvent` keyword. After you raise the event, the next step is to handle the raised event and do some processing based on that event. You can handle raised events in two ways, as discussed in the next two sections.

Receiving Events with `WithEvents`

Now that you have implemented an event within your `Customer` class, you can write client code to declare an object using the `WithEvents` keyword. For instance, in the project's Form1 code module, you can write the following:

```
Public Class Form1
  Inherits System.Windows.Forms.Form
  Private WithEvents cust As Customer
```

By declaring the variable `WithEvents`, you are indicating that you want to receive any events that this object raises. You can also choose to declare the variable without the `WithEvents` keyword, although, in that case, you would not receive events from the object as described here. Instead, you would use the `AddHandler` method, discussed shortly.

You can then create an instance of the object as the form is created by adding the following code:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) _
  Handles MyBase.Load
  cust = New Customer()
End Sub
```

At this point, you have declared the object variable using `WithEvents` and have created an instance of the `Customer` class so that you actually have an object with which to work. You can now proceed to write a method to handle the `Walked` event from the object by adding the following code to the form. You can name this method anything you like. The `Handles` clause is important because it links the event from the object directly to this method, so it is invoked when the event is raised:

```
Private Sub OnCustomerLoad(ByVal count as Integer) Handles cust.Loaded
  MessageBox.Show("No of Customers Loaded" & count)
End Sub
```

In his code, you are using the `Handles` keyword to indicate which event this method should handle. You are also receiving an `Integer` parameter. If the parameter list of the method doesn't match the list for the event, you will get a compiler error indicating the mismatch.

Finally, you need to call the `LoadCustomer` method on your `Customer` object. Add a button to the form and write the following code for its `Click` event:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
➥System.EventArgs) _
  Handles button1.Click
  cust.LoadCustomer()
End Sub
```


When the button is clicked, you simply call the `LoadCustomer` method. This causes the code in your class—including the `RaiseEvent` statement—to be run. The result is an event firing back into your form because you declared the `cust` variable using the `WithEvents` keyword. The `OnCustomerLoad` method is run to handle the event because it has the `Handles` clause linking it to the event. The `RaiseEvent` causes the `OnCustomerLoad` method in the form to be invoked; control returns to the `Walk` method in the object afterward. Because you have no code in the `Walk` method after you call `RaiseEvent`, the control returns to the `Click` event back in the form, and then you are finished.

Receiving Events with `AddHandler`

Now that you understand how to receive and handle events using the `WithEvents` and `Handles` keywords, this section takes a look at an alternative approach. You can use the `AddHandler` method to dynamically add event handlers through code.

`WithEvents` and the `Handles` clause require you to declare both the object variable and event handler as you build your code, effectively creating a linkage that is compiled right into your code. `AddHandler`, on the other hand, creates this linkage at runtime, which can provide you with more flexibility. Before going deep into that, you should see how `AddHandler` works.

In Form1, you can change the way your code interacts with the `Customer` object, first eliminating the `WithEvents` keyword:

```
Private Customer As Person
```

Then you eliminate the `Handles` clause:

```
Private Sub OnCustomerLoad(ByVal count As Integer)
  MessageBox.Show("No of Customers Loaded" & count)
End Sub
```

With these changes, you have eliminated all event handling for your object, so your form no longer receives the event, even though the `Customer` object raises it. Now you can change the code to dynamically add an event handler at runtime by using the `AddHandler` method. This method simply links an object's event to a method that should be called to handle that event. Any time after you've created your object, you can call `AddHandler` to set up the linkage:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
➥System.EventArgs)  _
  Handles MyBase.Load
  cust = New Customer()
  AddHandler cust.Loaded, AddressOf OnCustomerLoad
End Sub
```

This single line of code does the same thing as the earlier use of `WithEvents` and the `Handles` clause, causing the `OnCustomerLoad` method to be invoked when the `Loaded` event is raised from the `Customer` object.

TIP

With this approach of using `AddHandler` to receive events, it is important to note that it does the linkage at runtime. More control over the process also is provided than you have with the `WithEvents`-based approach.

Inheritance in Visual Basic .NET

Inheritance is often considered one of the most exciting features of Visual Basic .NET and is considered fundamental to the creation of object-based systems. Objects are a way to represent concepts or entities in code. Each of the object features of Visual Basic is designed to help you make the most useful representation possible. In many situations, one entity or concept is really what you would call a subobject of a more basic entity or concept. Consider the class of objects designed to represent animals, such as a cat, a tiger, or a lion. The class would have various properties, such as `Color`, `Weight`, and `Height`.

In this example, the general `Animal` class really contains several subclasses of objects, such as `Cats` and `Tigers`. Those classes, of course, would have all the properties of their parent class, `Animal`, such as `Color`, `Weight`, and `Height`, but they could also have unique properties of their own. For example, a `Tiger` class could have properties that describe the size and behavior of a tiger. This relationship between `Animal` and its subclasses of `Tiger` and `Cat` would be considered a parent-child relationship; the method for representing this relationship in your systems is called inheritance. The class `Tiger` is said to inherit from its base class, `Animal`. This relationship means that, in addition to any properties and methods created in the child class, the child also possesses all the properties and methods of the parent.

Inheritance in Visual Basic .NET

This section demonstrates the concepts of inheritance by looking at an example. To start, you could have a base class of `Animal`, which could represent any type of animal (tiger, cat, lion) and has the properties of `Color`, `Weight`, and `Height`. You could easily represent this class in Visual Basic code, as shown in Listing 3.2.

LISTING 3.2
`Animal` Class

```
Public Class Animal
  Public Property Color() As String
    Get
    End Get
    Set
    End Set
  End Property
  Public Property Weight() As Long
    Get
    End Get
    Set
    End Set
  End Property
  Public Function Description() As String
  End Function
End Class
```


The code that goes within the various procedures in the `Animal` class is not really relevant to this example, so it is left blank for now. If you were to jump to some code to try using your object (for instance, in the `Sub Main()` section), you would see that you can create objects of type `Animal` and work with their properties:

```
Shared Sub Main()
  Dim animalObj As Animal
  animalObj = New Animal
  animalObj.Color = "Red"
  animalObj.Weight = 100
End Sub
```

Now add another class, named `Tiger`, to your project by selecting Add Class from the Project menu. This class will inherit from `Animal`, just as the real class of object `Tiger` is a subclass or child of the `Animal` class of objects. Because you are creating a class designed to deal with just tigers, you can add properties (such as `Speed`) specific to this subclass of `Tiger`:

```
Public Class Tiger
  Inherits Animal
  Public Property Speed() As Long
    Get
    End Get
    Set
    End Set
  End Property
End Class
```


The key to this code is the line `Inherits Animal`, which tells Visual Basic that this class is a child of the `Animal` class and, therefore, should inherit all that class's properties and methods. Again, no code is actually placed into any of these property definitions because they are not really relevant at this time. When that code is in place, without having to do anything else, you can see the effect of the `Inherits` statement.

Returning to your `Main()` procedure, add code to create an object of type `Tiger`, and you will quickly see that it exposes both its own properties and those of the parent class.

Overriding Methods

When an inherited class adds new methods or properties, it is said to be extending the base class. In addition to extending, it is possible for a child class to override some or all of the functionality of the base class. This is done when the child implements a method or property that is also defined in the parent or base class. In such a case, the code in the child is executed instead of the code in the parent, enabling you to create specialized versions of the base method or property.

NOTE

For a child class to override some part of the base class, that portion must be marked `Overridable` in the base class definition. For instance, in the version of `Animal` listed earlier, none of its properties had the keyword `Overridable`, so child classes would be incapable of providing their own implementations.

As an example of how overriding is set up in the base and child classes, the code in Listing 3.3 marks the `Description()` function as being overridable and then overrides it in the `Tiger` child class. Note that nonrelevant portions of the two classes have been removed for clarity.

LISTING 3.3
Using the `Overridable` and `Overrides` Keywords

```
Public Class Animal
1   Overridable Public Function Description() As String
      Return "This is the description for a generic animal"
    End Function
End Class

Public Class Tiger
  Inherits Animal
  Overrides Public Function Description() As String
    Return "This is the description for tiger"
  End Function
End Class
```


When overriding a method or property, as you did in Listing 3.3, you can refer to the original member of the base class by using the built-in object `MyBase`. For example, to refer to the existing `Description()` method 1 of the `Animal` class, you could call `MyBase.Description()` from within the `Description` method of `Tiger`. This functionality enables you to provide additional functionality through overriding, without having to redo all the original code as well.

TIP

In addition to marking code as `Overridable`, it is possible to mark a method or property as `MustOverride` and a class as `MustInherit`. The `MustOverride` keyword indicates that any child of this class must provide its own version of this property or method. The `MustInherit` keyword means that this class cannot be used on its own (you must base other classes off it). It is important to note that if a class contains a method marked `MustOverride`, the class itself must be marked as `MustInherit`.

Summary

VB .NET offers you a fully object-oriented language with all the capabilities you would expect. This chapter covered the basic concepts surrounding classes and objects, as well as the separation of interface from implementation and data. It also demonstrated how to use the `Class` keyword to create classes and how those classes can be instantiated into specific objects.

These objects have methods and properties that client code can invoke, and they can act on data that is stored in member or instance variables of the object. The chapter also explored advanced concepts, such as shared properties, shared methods, events, and the use of delegates.

Further Reading

MSDN documentation on Visual Basic .NET.

Visual Basic Developer Center: http://msdn.microsoft.com/vbasic/.

Mackenzie, Duncan, and Kent Sharkey. *Sam's Teach Yourself Visual Basic .NET in 21 Days*. Pearson Education, 2001.