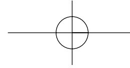
7167_Front.qxd 24/04/2002 11:04 Page 1



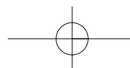
Programmer to Programmer ™

Professional VB.NET 2nd Edition

Written and tested for final release of .NET v1.0

Fred Barwell, Richard Case, Bill Forgey, Billy Hollis, Tim McCarthy, Jonathan Pinnock, Richard Blair, Jonathan Crossland, Whitney Hankison, Rockford Lhotka, Jan Narkiewicz, Rama Ramachandran, Matthew Reynolds, John Roth, Bill Sheldon, Bill Sempf

Wrox technical support at: support@wrox.com
Updates and source code at: www.wrox.com
Peer discussion at: p2p.wrox.com



What you need to use this book

To compile and run the example code in this book you will need:

- Windows 2000 Server, Windows 2000 Professional, or Windows XP with Internet Information Services (IIS) and MSMQ installed
- Visual Studio .NET Professional, or higher
- SQL Server 2000 or Microsoft SQL Server Desktop Engine (MSDE), which ships with the .NET Framework.

In addition, this book assumes the following knowledge:

- A good understanding of VB6. If you are a complete beginner to Visual Basic programming, Beginning Visual .NET (ISBN 1861004966) would be more suitable.

Summary of Contents

Chapter 1:	What is Microsoft .NET?	**9**
Chapter 2:	Introducing VB.NET and VS.NET	**27**
Chapter 3:	The Common Language Runtime	**55**
Chapter 4:	Variables and Types	**75**
Chapter 5:	Object Syntax Introduction	**105**
Chapter 6:	Inheritance and Interfaces	**159**
Chapter 7:	Applying Objects and Components	**219**
Chapter 8:	Namespaces	**253**
Chapter 9:	Error Handling	**267**
Chapter 10:	Using XML in VB.NET	**297**
Chapter 11:	Data Access with ADO.NET	**345**
Chapter 12:	Windows Forms	**391**
Chapter 13:	Creating Windows Controls	**431**
Chapter 14:	Web Forms	**469**
Chapter 15:	Creating Web Controls	**509**
Chapter 16:	Data Binding	**557**
Chapter 17:	Working with Classic COM and Interfaces	**587**
Chapter 18:	Component Services	**611**
Chapter 19:	Threading	**639**
Chapter 20:	Remoting	**687**
Chapter 21:	Windows Services	**717**
Chapter 22:	Web Services	**749**
Chapter 23:	VB.NET and the Internet	**783**
Chapter 24:	Security in the .NET Framework	**817**
Chapter 25:	Assemblies and Deployment	**867**
Appendix A:	Using the Visual Basic Compatibility Library	**927**
Index		**943**

5

Object Syntax Introduction

Visual Basic has had powerful object-oriented capabilities since the introduction of version 4.0. VB.NET carries that tradition forward. VB.NET simplifies some of the syntax and greatly enhances these capabilities, and now supports the four major defining concepts required for a language to be fully object-oriented:

- **Abstraction** – VB has supported abstraction since VB4. Abstraction is merely the ability of a language to create "black box" code – to take a concept and create an abstract representation of that concept within a program. A `Customer` object, for instance, is an abstract representation of a real-world customer. A `Recordset` object is an abstract representation of a set of data.
- **Encapsulation** – This has also been with us since version 4.0. It's the concept of a separation between interface and implementation. The idea is that we can create an interface (`Public` methods in a class) and, as long as that interface remains consistent, the application can interact with our objects. This remains true even if we entirely rewrite the code within a given method – thus the interface is independent of the implementation.

  Encapsulation allows us to hide the internal implementation details of a class. For example, the algorithm we use to compute Pi might be proprietary. We can expose a simple API to the end user, but we hide all of the logic used by our algorithm by encapsulating it within our class.

- **Polymorphism** – Likewise, polymorphism was introduced with VB4. Polymorphism is reflected in the ability 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 we can write a routine that calls the `Name` property regardless of whether we're using a `Customer` or `Vendor` object, then we have polymorphism.

  VB, in fact, supports polymorphism in two ways – through late binding (much like Smalltalk, a classic example of a true object-orientated language) and through the implementation of multiple interfaces. This flexibility is very powerful and is preserved within VB.NET.

- **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.

We'll discuss these concepts in detail in Chapter 7, using this chapter and Chapter 6 to focus on the syntax that enables us to utilize these concepts.

Additionally, because VB.NET is a component-based language, we have some other capabilities that are closely related to traditional concepts of object-orientation:

- **Multiple interfaces** – Each class in VB.NET defines a primary interface (also called the default or native interface) through its Public methods, properties and events. Classes can also implement other, secondary interfaces in addition to this primary interface. An object based on this class then has multiple interfaces, and a client application can choose by which interface it will interact with the object.

- **Assembly (component) level scoping** – Not only can we define our classes and methods to be `Public` (available to anyone), `Protected` (available through inheritance) and `Private` (only available locally), but we can also define them as `Friend` – meaning they are only available within the current assembly or component. This is not a traditional object-oriented concept, but is very powerful when designing component-based applications.

In this chapter we'll explore the creation and use of classes and objects in VB.NET. In Chapter 6, we'll examine inheritance and how it can be used within VB.NET. In Chapter 7, we'll explore object-oriented programming in depth, fully defining the features listed and exploring how we can use these concepts.

Before we get too deep into code, however, it is important that we spend a little time familiarizing ourselves with basic object-oriented terms and concepts.

Object-Oriented Terminology

To start with, let's take a look at the word **object** itself, along with the related **class** and **instance** terms. Then we'll move on to discuss the four terms that define the major functionality in the object-oriented world – encapsulation, abstraction, polymorphism, and inheritance.

Objects, Classes, and Instances

An **object** is a code-based abstraction of a real-world entity or relationship. For instance, we might have a `Customer` object that represents a real-world customer – such as customer number 123 – or we might have a `File` object that represents `C:\config.sys` on our computer's hard drive.

A closely related term is **class**. A class is the code that defines our 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 we create instances of specific objects. For example, in order to have a `Customer` object representing customer number 123, we must first have a `Customer` class that contains all of the code (methods, properties, events, variables, and so on) necessary to create `Customer` objects. Based on that class, we can create any number of objects – each one an **instance** of the class. Each object is identical to the others – except that it may contain different data.

We may create many instances of `Customer` objects based on the same `Customer` class. All of 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.

Composition of an Object

We 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 – but rather those programs must make use of our object's interface.

Let's walk through each of the three elements 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.

> *The word 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, while in .NET an attribute is a coding construct that we can use to control compilation, the IDE, and so on.*

We can also have `Private` methods and properties in our code. While these methods can be called by code within our object, they are not part of the interface and cannot be called by programs written to use our object. Another option is to use the `Friend` keyword, which defines the scope to be our current project, meaning that any code within our project can call the method, but no code outside of our project (that is, from a different .NET assembly) can call the method. To complicate things a bit, we can also declare methods and properties as `Protected`, which are available to classes that inherit from our class. We'll discuss `Protected` in Chapter 6 along with inheritance.

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

```
Public Function CalculateValue() As Integer

End Function
```

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

```
Private Sub DoSomething()

End Sub
```

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

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

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

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

Finally, we can use the `Friend` keyword:

```
Friend Sub DoSomething()

End Sub
```

In this case, the `DoSomething` method can be called by code within our class, or from other classes or modules within our current VB.NET project. Code from outside our 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 of our object itself. However, unlike `Public`, the `Friend` keyword restricts access to code within our current VB.NET project – preventing code in other .NET assemblies from calling the method.

> *This is very unlike the C++ `friend` keyword, which implements a form of tight coupling between objects and which is generally regarded as a bad thing to do. Instead, this is the same `Friend` keyword that VB has had for many years and which was later adopted by Java to provide component-level scoping in that language as well. It is equivalent to the `internal` keyword in C#.*

Implementation or Behavior

The code inside of a method is called the **implementation**. Sometimes it is also called **behavior** since it is this code that actually makes the object do useful work.

For instance, we may have an `Age` property as part of our object's interface. Within that method, we may have some code (perhaps written by an inexperienced developer, since it is just returning a non-calculated value):

```
Private mintAge As Integer

Public ReadOnly Property Age() As Integer
  Get
    Return mintAge
```

```
    End Get
  End Sub
```

In this case, the code is returning a value directly out of a variable, rather than doing something better like calculating the value based on a birth date. However, this kind of code is often written in applications, and it seems to work fine for a while.

The key concept here is to understand that client applications can use our object even if we change the implementation – as long as we don't change the interface. As long as our method name and its parameter list and return data type remain unchanged, we can change the implementation all we want.

The code necessary to call our `Age` property would look something like this:

```
  theAge = MyObject.Age
```

The result of running this code is that we get the `Age` value returned for our use. While our client application will work fine, we'll soon discover that hard coding the age into the application is a problem and so, at some point, we'll want to improve this code. Fortunately, we can change our implementation without changing the client code:

```
  Private mdtBirthDate As Date

  Public ReadOnly Property Age() As Integer
    Get
      Return DateDiff(DateInterval.Year, mdtBirthDate, Now())
    End Get
  End Sub
```

We've changed the implementation behind the interface – effectively changing how it behaves – without changing the interface itself. Now, when our client application is run, we'll find that the `Age` value returned is accurate over time where, with the previous implementation, it was not.

It is important to keep in mind that encapsulation is a syntactic tool – it allows our code to continue to run without change. However, it is not semantic – meaning that, just because our code continues to run, doesn't mean it continues to do what we actually wanted it to do.

In this example, our client code may have been written to overcome the initial limitations of the implementation in some way, and thus might not only rely on being able to retrieve the `Age` value, but the client code might be counting on the result of that call being a fixed value over time.

While our update to the implementation won't stop the client program from running, it may very well prevent the client program from running correctly.

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 absolutely identical in terms of its interface and its implementation – the only thing that can vary at all is the data contained within that particular object.

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

We shouldn't confuse instance variables with *properties*. In VB, a `Property` is a type of method that is geared around retrieving and setting values, while an instance variable is a variable within the class that may *hold* the value exposed by a `Property`.

For instance, we might have a class that has instance variables:

```
Public Class TheClass
  Private mstrName As String
  Private mdtBirthDate As Date
End Class
```

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 only available to code within each specific object.

While member variables *can* be declared as `Public` in scope, this makes them available to any code using our objects in a manner we can't control. Such a choice directly breaks the concept of encapsulation, since code outside our object can directly change data values without following any rules that might otherwise be set in our object's code.

If we want to make the value of an instance variable available to code outside of our object, we should use a **property**:

```
Public Class TheClass
  Private mstrName As String
  Private mdtBirthDate As Date

  Public ReadOnly Property Name() As String
    Get
      Return mstrName
    End Get
  End Property
End Class
```

Since the `Name` property is a method, we are not directly exposing our internal variables to client code, so we preserve encapsulation of our data. At the same time, through this mechanism we are able to safely provide access to our data as needed.

Member variables can also be declared with `Friend` scope – which means they are available to all code in our project. Like declaring them as `Public`, this breaks encapsulation and is strongly discouraged.

Now that we have a grasp on some of the basic object-oriented terminology, we're ready to explore the creation of classes and objects. First, we'll see how VB allows us to interact with objects, and then we'll dive into the actual process of authoring those objects.

Working with Objects

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

Object Declaration and Instantiation

Objects are created using the `New` keyword – indicating that we want a new instance of a particular class. There are a number of variations on how or where we can use the `New` keyword in our code. Each one provides different advantages in terms of code readability or flexibility.

> *Unlike previous versions of VB, VB.NET doesn't use the `CreateObject` statement for object creation. `CreateObject` was an outgrowth of VB's relationship with COM and, since VB.NET doesn't use COM, it has no use for `CreateObject`. The `CreateObject` method still exists to support COM interoperability, but is not used to access .NET objects.*

The most obvious way to create an object is to declare an object variable and then create an instance of the object:

```
Dim obj As TheClass
obj = New TheClass()
```

The result of this code is that we have a new instance of `TheClass` ready for our use. To interact with this new object, we will use the `obj` variable that we declared. The `obj` variable contains a reference to the object – a concept we'll explore more later.

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

```
Dim obj As New TheClass()
```

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

This code both declares the variable `obj` as data type `TheClass` and also creates an instance of the class – immediately creating an object that we can use from our code.

Another variation on this theme is:

```
Dim obj As TheClass = New TheClass()
```

Again, this both declares a variable of data type `TheClass` and creates an instance of the class for our use.

This third syntax provides a great deal of flexibility while remaining compact. 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 with multiple interfaces. We might declare the variable to be of one type – say an interface – and instantiate the object based on a class that implements that interface. We'll cover interfaces in detail in Chapter 6 but as an example here, let's create an interface named `ITheInterface`:

```
Public Interface ITheInterface
  Sub DoSomething()
End Interface
```

Our class can then implement that interface, meaning that our class now has its own native interface and also has a secondary interface – `ITheInterface`:

```
Public Class TheClass
  Implements ITheInterface

  Public Sub DoSomething() Implements ITheInterface.DoSomething
    ' implementation goes here
  End Sub
End Class
```

We can now create an instance of `TheClass`, but reference it via the secondary interface by declaring the variable to be of type `ITheInterface`:

```
Dim obj As ITheInterface = New TheClass()
```

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

```
Dim obj As ITheInterface
obj = New TheClass()
```

Either technique works fine and achieves the same result, which is that we have a new object of type `TheClass`, being accessed via its secondary interface. We'll discuss multiple interfaces in more detail in Chapter 6.

So far we've been declaring a variable for our new objects. However, sometimes we may simply need to pass an object as a parameter to a method – in which case we can create an instance of the object right in the call to that method:

```
DoSomething(New TheClass())
```

This calls the `DoSomething` method, passing a new instance of `TheClass` as a parameter.

This can be even more complex. Perhaps, instead of needing an object reference, our method needs an `Integer`. We can provide that `Integer` value from a method on our object:

```
Public Class TheClass
  Public Function GetValue() As Integer
    Return 42
  End Function
End Class
```

We can then instantiate the object and call the method all in one shot, thus passing the value returned from the method as a parameter:

```
DoSomething(New TheClass().GetValue())
```

Obviously, we need to carefully weigh the readability of such code against its compactness – at some point, having more compact code can detract from readability rather than enhancing it.

Notice that nowhere do we use the `Set` statement when working with objects. In VB6, any time we worked with an object reference we had to use the `Set` command – differentiating objects from any other data type in the language.

> **In VB.NET, objects are not treated differently from any other data type, and so we can use direct assignment for objects just like we do with `Integer` or `String` data types. The `Set` command is no longer valid in VB.NET.**

Object References

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

When we create a new object using the `New` keyword, we store a reference to that object in a variable. For instance:

```
Dim obj As New TheClass()
```

This code creates a new instance of `TheClass`. We gain access to this new object via the `obj` variable. This variable holds a reference to the object. We might then do something like this:

```
Dim another As TheClass
another = obj
```

Now we have a second variable, `another`, which also has a reference to that same object. We can use either variable interchangeably, since they both reference the exact same object. The thing we need to remember is that the variable we have is not the object itself but, rather, is just a reference or pointer to the object itself.

Dereferencing Objects

When we are done working with an object, we can indicate that we're through with it by dereferencing the object.

To dereference an object, we need to simply set our object reference to `Nothing`:

```
Dim obj As TheClass

obj = New TheClass()
obj = Nothing
```

This code has no impact on our object itself. In fact, the object may remain blissfully unaware that it has been dereferenced for some time.

Once any and all variables that reference an object are set to `Nothing`, the .NET runtime can tell that we no longer need that object. At some point, the runtime will destroy the object and reclaim the memory and resources consumed by the object.

Between the time that we 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 does destroy the object, the framework will call the `Finalize` method on the object (if it has one). We discussed the `Finalize` method in Chapter 3.

Early versus Late Binding

One of the strengths of Visual Basic has long been that we had access to both early and late binding when interacting with objects.

Early binding means that our code directly interacts with the object, by directly calling its methods. Since the VB compiler knows the object's data type ahead of time, it can directly compile code to invoke the methods on the object. Early binding also allows the IDE to use IntelliSense to aid our development efforts; it allows the compiler to ensure that we are referencing methods that do exist and that we are providing the proper parameter values.

> *In previous versions of VB, early binding was also known as vtable binding. The vtable was an artifact of COM, providing a list of the addresses for all the methods on an object's interface. In .NET, things are simpler and there is no real vtable. Instead, the compiler is able to generate code to directly invoke the methods on an object. From a VB coding perspective this makes no difference, but it is quite a change behind the scenes.*

Late binding means that our code interacts with an object dynamically at run-time. This provides a great deal of flexibility since our code literally doesn't care what type of object it is interacting with as long as the object supports the methods we want to call. Because the type of the object isn't known by the IDE or compiler, neither IntelliSense nor compile-time syntax checking is possible but we get unprecedented flexibility in exchange.

If we enable strict type checking by using `Option Strict On` at the top of our code modules, then the IDE and compiler will enforce early binding behavior. By default, `Option Strict` is turned off and so we have easy access to the use of late binding within our code. We discussed `Option Strict` in Chapter 4.

Implementing Late Binding

Late binding occurs when the compiler can't determine the type of object that we'll be calling. This level of ambiguity is achieved through the use of the `Object` data type. A variable of data type `Object` can hold virtually any value – including a reference to any type of object. Thus, code such as the following could be run against any object that implements a `DoSomething` method that accepts no parameters:

```
Option Strict Off

Module LateBind
  Public Sub DoWork(ByVal obj As Object)
    obj.DoSomething()
  End Sub
End Module
```

If the object passed into this routine does not have a `DoSomething` method that accepts no parameters, then a run-time error will result. Thus, it is recommended that any code that uses late binding always provides error trapping:

```
Option Strict Off

Module LateBind
  Public Sub DoWork(ByVal obj As Object)
    Try
      obj.DoSomething()
    Catch ex As Exception When Err.Number = 438
      ' do something appropriate given failure to call the method
    End Try
  End Sub
End Module
```

Here, we've put the call to the `DoSomething` method in a `Try` block. If it works then the code in the `Catch` block is ignored but, in the case of a failure, the code in the `Catch` block is run. We would need to write code in the `Catch` block to handle the case that the object did not support the `DoSomething` method call. This `Catch` block, in fact, only catches error number 438, which is the error indicating that the method doesn't exist on the object.

While late binding is flexible, it can be error prone and it is slower than early bound code. To make a late bound method call, the .NET runtime must dynamically determine if the target object actually has a method that matches the one we're calling, and then it must invoke that method on our behalf. This takes more time and effort than an early bound call where the compiler knows ahead of time that the method exists and can compile our code to make the call directly. With a late bound call, the compiler has to generate code to make the call dynamically at runtime.

Use of the CType Function

Whether we are using late binding or not, it can be useful to pass object references around using the `Object` data type – converting them to an appropriate type when we need to interact with them. This is particularly useful when working with objects that use inheritance or implement multiple interfaces – concepts that we'll discuss in Chapter 6.

If `Option Strict` is turned off, which is the default, we can write code that allows us to use a variable of type `Object` to make an early bound method call:

```
Module LateBind
  Public Sub DoWork(obj As Object)
    Dim local As TheClass
```

```
      local = obj
      local.DoSomething()
    End Sub
  End Module
```

We are using a strongly typed variable, `local`, to reference what was a generic object value. Behind the scenes, VB.NET converts the generic type to a specific type so it can be assigned to the strongly typed variable. If the conversion can't be done we'll get a trappable runtime error.

The same thing can be done using the `CType` function. If `Option Strict` is enabled, then the previous approach will not compile and the `CType` function must be used. Here is the same code making use of `CType`:

```
Module LateBind
  Public Sub DoWork(obj As Object)
    Dim local As TheClass

    local = CType(obj, TheClass)
    local.DoSomething()
  End Sub
End Module
```

Here, we've declared a variable of type `TheClass`, which is an early bound data type that we want to use. The parameter we're accepting, though, is of the generic `Object` data type, and so we use the `CType()` method to gain an early bound reference to the object. If the object isn't of type `TheClass`, the call to `CType()` will fail with a trappable error.

Once we have a reference to the object, we can call methods by using the early bound variable, `local`.

> *Since all the method calls with `CType()` are early bound, this code will work even if we override the default and set `Option Strict On`.*

This code can be shortened to avoid the use of the intermediate variable. Instead, we can simply call methods directly from the data type:

```
Module LateBind
  Public Sub DoWork(obj As Object)
    CType(obj, TheClass).DoSomething()
  End Sub
End Module
```

Even though the variable we're working with is of type `Object` and, thus, any calls to it will be late bound, we are using the `CType` method to temporarily convert the variable into a +specific type – in this case, the type `TheClass`.

> **If the object passed as a parameter is not of type `TheClass`, we will get a trappable error, so it is always wise to wrap this code in a `Try...Catch` block.**

The CType function can be very useful when working with objects that implement multiple interfaces, since we can reference a single object variable through the appropriate type as needed. For instance, as we discussed earlier, if we have an object of type TheClass that also implements ITheInterface, we can use that interface with the following code:

```
Dim obj As TheClass

obj = New TheClass
CType(obj, ITheInterface).DoSomething()
```

In this way, we can make early bound calls to other interfaces on an object without needing to declare a new variable of the interface type. We'll discuss multiple interfaces in detail in Chapter 6.

Creating Classes

Using objects is fairly straightforward and intuitive. It is the kind of thing that even the most novice programmers pick up and accept rapidly. Creating classes and objects is a bit more complex and interesting, however, and that is what we'll cover through the rest of the chapter.

Creating Basic Classes

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

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

The Class Keyword

If we wanted to create a class that represents a person – a Person class – we could use the Class keyword like so:

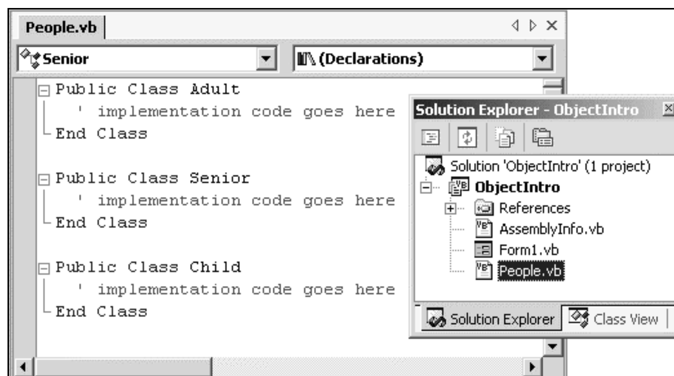
```
Public Class Person
  ' implementation code goes here
End Class
```

As we know, VB.NET projects are composed of a set of files with the .vb extension. Each file can contain multiple classes. This means that, within a single file, we could have something like this:

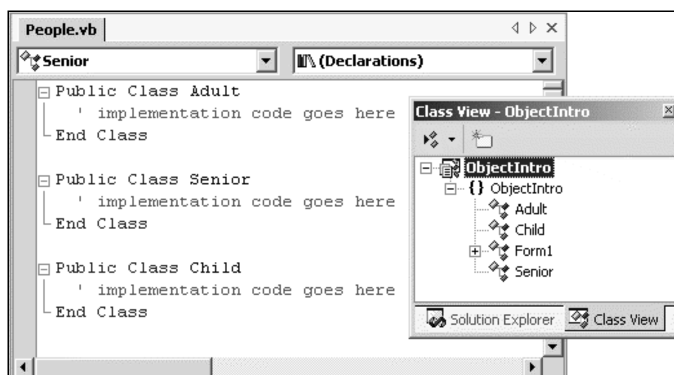
```
Public Class Adult
  ' implementation code goes here
End Class

Public Class Senior
  ' implementation code goes here
```

```
End Class

Public Class Child
  ' implementation code goes here
End Class
```

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 our code. For instance, if we create a single class file with all these classes, the Solution Explorer simply shows a single entry:



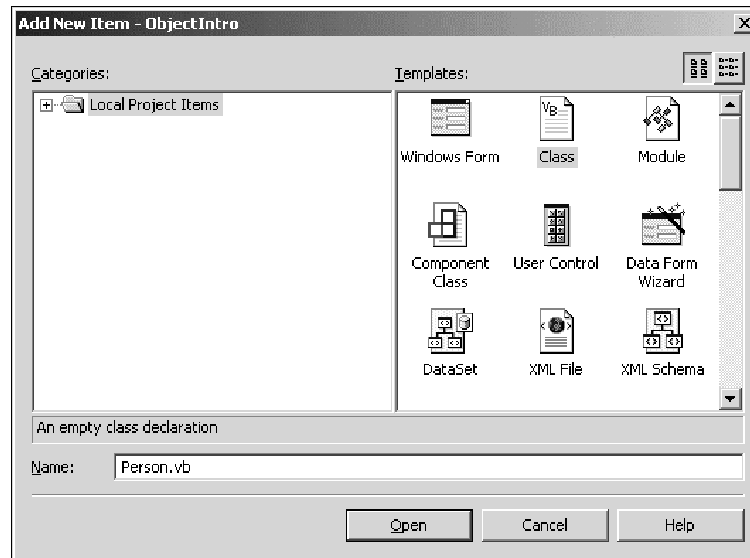
However, the VS.NET IDE does provide the Class View window. If we do decide to put multiple classes in each physical `.vb` file, we can make use of the Class View window to quickly and efficiently navigate through our code – jumping from class to class without having to manually locate those classes in specific code files:



> **The Class View window is incredibly useful even if we keep to one class per file, since it still provides us with a class-based view of our entire application.**

In this chapter, we'll stick with one class per file, as it is the most common approach. Open the VS.NET IDE and create a new Windows Application project. Name it `ObjectIntro`.

Choose the Project | Add Class menu option to add a new class module to the project. We'll be presented with the standard Add New Item dialog.



Change the name to `Person.vb` and click Open. The result will be the following code that defines our `Person` class:

```
Public Class Person

End Class
```

It is worth noting that all VB.NET source files end in a `.vb` extension, regardless of which type of VB source file we choose (form, class, module, etc.) when we are adding the file to our project. In fact, any forms, classes, components, or controls that we add to our project are actually class modules – they are just specific types of classes that provide the appropriate behaviors. Typically, these behaviors come from another class via inheritance, which we'll discuss in Chapter 6.

> *The exception is the `Module`, which is a special construct that allows us to include code within our 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 our project.*

With our `Person` class created, we're ready to start adding code to declare our interface, implement our behaviors, and to declare our instance variables.

Member Variables

Member or instance variables are variables declared in our class that will be available to each individual object when our application is run. Each object gets its own set of data – basically each object gets its own copy of the variables.

At the beginning of the chapter, we discussed how a class is simply a template from which we create specific objects. Variables that we define within our class are also simply templates – and each object gets its own copy of those variables in which to store its data.

Declaring member variables is as easy as declaring variables within the `Class` block structure. Add the following code to our `Person` class:

```
Public Class Person
  Private mstrName As String
  Private mdtBirthDate As Date

End Class
```

We can control the scope of our variables by using the following keywords:

- `Private` – available only to code within our class
- `Friend` – available only to code within our project/component
- `Protected` – available only to classes that inherit from our class – discussed in detail in Chapter 6
- `Protected Friend` – available to code within our project/component and classes that inherit from our class whether in our project or not – discussed in detail in Chapter 6
- `Public` – available to code outside our class

Typically, member variables are declared using the `Private` keyword – making them available only to code within each instance of our class. Choosing any other option should be done with great care, as all the other options allow code *outside* our class to directly interact with the variable – meaning that the value could be changed and our code would never know that a change took place.

One common exception to making variables `Private` is the use of the `Protected` keyword, as we'll discuss in Chapter 6.

Methods

Objects typically need to provide services (or functions) that we can call when working with the object. Using their own data, or data passed as parameters to the method, they manipulate information to yield a result or to perform a service.

Methods declared as `Public`, `Friend`, or `Protected` in scope define the interface of our class. Methods that are `Private` in scope are only available to the code within the class itself, and can be used to provide structure and organization to our code. As we discussed earlier, the actual code within each method is called *implementation*, while the declaration of the method itself is what defines our interface.

Methods are simply routines that we code within the class to implement the services that we want to provide to the users of our object. Some methods return values or provide information back to the calling code. These are called **interrogative methods**. Others, called **imperative methods**, just perform a service and return nothing to the calling code.

In VB.NET, methods are implemented using `Sub` (for imperative methods) or `Function` (for interrogative methods) routines within the class module that defines our object. `Sub` routines may accept parameters, but they don't return any result value when they are complete. `Function` routines can also accept parameters, and they always generate a result value that can be used by the calling code.

A method declared with the `Sub` keyword is merely one that returns no value. Add the following code to our `Person` class:

```
Public Sub Walk()
  ' implementation code goes here
End Sub
```

The `Walk` method would presumably contain some code that performed some useful work when called, but has no result value to return when it is complete.

To use this method, we might write code such as:

```
Dim myPerson As New Person()
myPerson.Walk()
```

Once we've created an instance of the `Person` class, we can simply invoke the `Walk` method.

Methods that Return Values

If we have a method that does generate some value that should be returned, we need to use the `Function` keyword:

```
  Public Function Age() As Integer
    Return DateDiff(DateInterval.Year, mdtBirthDate, Now())
  End Function
```

Notice that we need to indicate the data type of the return value when we declare a `Function`. In this example, we are returning the calculated age as a result of the method. We can return any value of the appropriate data type by using the `Return` keyword.

We can also return the value without using the `Return` keyword, by setting the value of the function name itself:

```
Public Function Age() As Integer
    Age = DateDiff(DateInterval.Year, mdtBirthDate, Now())
End Function
```

This is functionally equivalent to the previous code. Either way, we can use this method with code similar to the following:

```
Dim myPerson As New Person()
Dim intAge As Integer

intAge = myPerson.Age()
```

The `Age` method returns an `Integer` data value that we can use in our program as required – in this case we're just storing it into a variable.

Indicating Method Scope

Adding the appropriate keyword in front of the method declaration indicates the scope:

```
Public Sub Walk()
```

This indicates that `Walk` is a `Public` method and is thus available to code outside our class and even outside our current project. Any application that references our assembly can make use of this method. By being `Public`, this method becomes part of our object's interface.

On the other hand, we might choose to restrict the method somewhat:

```
Friend Sub Walk()
```

By declaring the method with the `Friend` keyword, we are indicating that it should be part of our object's interface only for code inside our project – any other applications or projects that make use of our assembly will not be able to call the `Walk` method.

```
Private Function Age() As Integer
```

The `Private` keyword indicates that a method is only available to the code within our particular class. `Private` methods are very useful to help us organize complex code within each class. Sometimes our methods will contain very lengthy and complex code. In order to make this code more understandable, we may choose to break it up into several smaller routines, having our main method call these routines in the proper order. Additionally, we may use these routines from several places within our class and so, by making them separate methods, we enable reuse of the code. These sub-routines should never be called by code outside our object – and so we make them `Private`.

Method Parameters

We will often want to pass information into a method as we call it. This information is provided via parameters to the method. For instance, in our `Person` class, perhaps we want our `Walk` method to track the distance the person walks over time. In such a case, the `Walk` method would need to know how far the person is to walk each time the method is called. Add the following code to our Person class:

```
Public Class Person
  Private mstrName As String
  Private mdtBirthDate As Date
  Private mintTotalDistance As Integer

  Public Sub Walk(ByVal Distance As Integer)
    mintTotalDistance += Distance
  End Sub

  Public Function Age() As Integer
    Return DateDiff(DateInterval.Year, mdtBirthDate, Now())
  End Function
End Class
```

With this implementation, a `Person` object will sum up all of the distances that are walked over time. Each time the `Walk` method is called, the calling code must pass an `Integer` value indicating the distance to be walked. Our code to call this method would be similar to the following:

```
Dim myPerson As New Person()
myPerson.Walk(12)
```

The parameter is accepted using the `ByVal` keyword. This indicates that the parameter value is a *copy* of the original value. This is the default way VB.NET accepts all parameters. Typically, this is desirable because it means that we can work with the parameter inside our code – including changing its value – with no risk of accidentally changing the original value back in the calling code.

If we do want to be able to change the value in the calling code, we can change the declaration to pass the parameter by reference by using the `ByRef` qualifier:

```
Public Sub Walk(ByRef Distance As Integer)
```

In this case, we'll get a reference (or pointer) back to the original value rather than receiving a copy. This means that any change we make to the `Distance` parameter will be reflected back in the calling code – very similar to the way object references work, as we discussed earlier in Chapter 4.

> **Using this technique can be dangerous, since it is not explicitly clear to the caller of our method that the value will change. Such unintended side effects can be hard to debug and should be avoided.**

Properties

The .NET environment provides for a specialized type of method called a **property**. A property is a method specifically designed for setting and retrieving data values. For instance, we declared a variable in our `Person` class to contain a name, so our `Person` class may include code to allow that name to be set and retrieved. This could be done using regular methods:

```
Public Sub SetName(ByVal Name As String)
  mstrName = Name
End Sub

Public Function GetName() As String
  Return mstrName
End Function
```

Using methods like these, we would write code to interact with our object such as:

```
Dim myPerson As New Person()

myPerson.SetName("Jones")
MsgBox(myPerson.GetName())
```

While this is perfectly acceptable, it is not as nice as it could be through the use of a property. A `Property` style method consolidates the setting and retrieving of a value into a single structure, and also makes the code within our class smoother overall. We can rewrite these two methods into a single property. Add the following code to the `Person` class:

```
    Public Property Name() As String
      Get
        Return mstrName
      End Get
      Set(ByVal Value As String)
        mstrName = Value
      End Set
    End Property
```

By using a property method instead, we can make our client code much more readable:

```
Dim myPerson As New Person()

myPerson.Name = "Jones"
MsgBox(myPerson.Name)
```

The `Property` method is declared with both a scope and a data type:

```
    Public Property Name() As String
```

In this example, we've declared the property as `Public` in scope, but it can be declared using the same scope options as any other method – `Public`, `Friend`, `Private`, or `Protected`.

As with other methods, a `Public` property is accessible to any code outside our class, while `Friend` is available outside our class, but only to code within our VB project. `Protected` properties are available through inheritance, as we'll discuss in Chapter 6, and `Private` properties are only available to code within our class.

The return data type of this property is `String`. A property can return virtually any data type as appropriate for the nature of the value. In this regard, a property is very similar to a method declared using the `Function` keyword.

Though a `Property` method is a single structure, it is divided into two parts: a getter and a setter. The getter is contained within a `Get...End Get` block and is responsible for returning the value of the property on demand:

```
      Get
        Return mstrName
      End Get
```

Though the code in this example is very simple, it could be more complex – perhaps calculating the value to be returned or applying other business logic to change the value as it is returned.

Likewise, the code to change the value is contained within a `Set`...`End Set` block:

```
      Set(ByVal Value As String)
        mstrName = Value
      End Set
```

The `Set` statement accepts a single parameter value that stores the new value. Our code in the block can then use this value to set the property's value as appropriate. The data type of this parameter must match the data type of the property itself. By having the parameter declared in this manner, we can change the variable name used for the parameter value if needed.

By default, the parameter is named `Value`. However, if we dislike the name `Value`, we can change the parameter name to something else, for example:

```
      Set(ByVal NewName As String)
        mstrName = NewName
      End Set
```

In many cases, we may apply business rules or other logic within this routine to ensure that the new value is appropriate before we actually update the data within our object.

Parameterized Properties

The `Name` property we created is an example of a single-value property. We can also create property arrays or parameterized properties. These properties reflect a range, or array, of values. As an example, a person will often have several phone numbers. We might implement a `PhoneNumber` property as a parameterized property – storing not only phone numbers, but also a description of each number. To retrieve a specific phone number we'd write code such as:

```
  Dim myPerson As New Person()
  Dim strHomePhone As String

  strHomePhone = myPerson.Phone("home")
```

Or, to add or change a specific phone number, we'd write:

```
  myPerson.Phone("work") = "555-9876"
```

Not only are we retrieving and updating a phone number property, but also we're updating some specific phone number. This implies a couple of things. First off, we're no longer able to use a simple variable to hold the phone number, since we are now storing a list of numbers and their associated names. Secondly, we've effectively added a parameter to our property – we're actually passing the name of the phone number as a parameter on each property call.

To store the list of phone numbers we can use the `Hashtable` class. The `Hashtable` is very similar to the standard VB `Collection` object, but it is more powerful – allowing us to test for the existence of an existing element. Add the following declaration to the `Person` class:

```
  Public Class Person
    Private mstrName As String
    Private mdtBirthDate As Date
    Private mintTotalDistance As Integer
    Private colPhones As New Hashtable()
```

We can implement the `Phone` property by adding the following code to our `Person` class:

```
Public Property Phone(ByVal Location As String) As String
  Get
    Return CStr(colPhones.Item(Location))
  End Get
  Set(ByVal Value As String)
    If colPhones.ContainsKey(Location) Then
      colPhones.Item(Location) = Value
    Else
      colPhones.Add(Location, Value)
    End If
  End Set
End Property
```

The declaration of the `Property` method itself is a bit different from what we've seen:

```
Public Property Phone(ByVal Location As String) As String
```

In particular, we've added a parameter, `Location`, to the property itself. This parameter will act as the index into our list of phone numbers and must be provided both when setting or retrieving phone number values.

Since the `Location` parameter is declared at the `Property` level, it is available to all code within the property – including both the `Get` and `Set` blocks.

Within our `Get` block, we use the `Location` parameter to select the appropriate phone number to return from the `Hashtable`:

```
  Get
    Return colPhones.Item(Location)
  End Get
```

With this code, if there is no value stored matching the `Location`, we'll get a trappable runtime error.

Similarly, in the `Set` block, we use the `Location` to update or add the appropriate element in the `Hashtable`. In this case, we're using the `ContainsKey` method of `Hashtable` to determine whether the phone number already exists in the list. If it does, we'll simply update the value in the list – otherwise, we'll add a new element to the list for the value:

```
  Set(ByVal Value As String)
    If colPhones.ContainsKey(Location) Then
      colPhones.Item(Location) = Value
    Else
      colPhones.Add(Location, Value)
    End If
  End Set
```

In this way, we're able to add or update a specific phone number entry based on the parameter passed by the calling code.

Read-Only Properties

There are times when we may want a property to be read-only – so that it can't be changed. In our Person class, for instance, we may have a read-write property for BirthDate, but just a read-only property for Age. In such a case, the BirthDate property is a normal property, as follows:

```
Public Property BirthDate() As Date
  Get
    Return mdtBirthDate
  End Get
  Set(ByVal Value As Date)
    mdtBirthDate = Value
  End Set
End Property
```

The Age value, on the other hand, is a derived value based on BirthDate. This is not a value that should ever be directly altered and, thus, is a perfect candidate for read-only status.

We already have an Age method – implemented as a Function. Remove that code from the Person class, as we'll be replacing it with a Property routine instead.

The difference between a Function routine and a ReadOnly Property is quite subtle. Both return a value to the calling code and, either way, our object is running a subroutine defined by our class module to return the value.

The difference is less a programmatic one than a design choice. We could create all our objects without any Property routines at all, just using methods for all interactions with the object. However, Property routines are obviously attributes of the object, while a Function might be an attribute or a method. By carefully implementing all attributes as ReadOnly Property routines, and any interrogative methods as Function routines, we will create more readable and understandable code.

To make a property read-only, we use the ReadOnly keyword and only implement the Get block:

```
Public ReadOnly Property Age() As Integer
  Get
    Return CInt(DateDiff(DateInterval.Year, mdtBirthDate, Now()))
  End Get
End Property
```

Since the property is read-only, we'll get a syntax error if we attempt to implement a Set block.

Write-Only Properties

As with read-only properties, there are times when a property should be write-only – where the value can be changed, but not retrieved.

Many people have allergies, so perhaps our Person object should have some understanding of the ambient allergens in the area. This is not a property that should be read from the Person object since allergens come from the environment rather than from the person, but it is data that the Person object needs in order to function properly. Add the following variable declaration to our class:

```
Public Class Person
  Private mstrName As String
  Private mdtBirthDate As Date
  Private mintTotalDistance As Integer
  Private colPhones As New Hashtable()
  Private mintAllergens As Integer
```

We can implement an `AmbientAllergens` property as follows:

```
Public WriteOnly Property AmbientAllergens() As Integer
  Set(ByVal Value As Integer)
    mintAllergens = Value
  End Set
End Property
```

To create a write-only property, we use the `WriteOnly` keyword and only implement a `Set` block in our code. Since the property is write-only, we'll get a syntax error if we attempt to implement a `Get` block.

The Default Property

Objects can implement a default property if desired. A default property can be used to simplify the use of our object at times, by making it appear as if our object has a native value. A good example of this behavior is the `Collection` object, which has a default property called `Item` that returns the value of a specific item, allowing us to write code similar to:

```
Dim colData As New Collection()

Return colData(Index)
```

Default properties *must be* parameterized properties. A property without a parameter cannot be marked as the default. This is a change from previous versions of VB, where any property could be marked as the default.

Our `Person` class has a parameterized property – the `Phone` property we built earlier. We can make this the default property by using the `Default` keyword:

```
Default Public Property Phone(ByVal Location As String) As String
  Get
    Return colPhones.Item(Location)
  End Get
  Set(ByVal Value As String)
    If colPhones.ContainsKey(Location) Then
      colPhones.Item(Location) = Value
    Else
      colPhones.Add(Location, Value)
    End If
  End Set
End Property
```

Prior to this change, we would need code such as the following to use the `Phone` property:

```
Dim myPerson As New Person()

MyPerson.Phone("home") = "555-1234"
```

But now, with the property marked as `Default`, we can simplify our code:

```
myPerson("home") = "555-1234"
```

By picking appropriate default properties, we can potentially make the use of our objects more intuitive.

Events

Both methods and properties allow us to write code that interacts with our objects by invoking specific functionality as needed. It is often useful for our objects to provide notification as certain activities occur during processing. We see examples of this all the time with controls, where a button indicates it was clicked via a `Click` event, or a textbox indicates its contents have changed via the `TextChanged` event.

Our objects can raise events of their own – providing a powerful and easily implemented mechanism by which objects can notify our client code of important activities or events. In VB.NET, events are provided using the standard .NET mechanism of **delegates**. We'll discuss delegates after we explore how to work with events in VB.

Handling Events

We are all used to seeing code in a form to handle the `Click` event of a button – code such as:

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

End Sub
```

Typically we just write our code in this routine without paying a lot of attention to the code created by the VS.NET IDE. However, let's take a second look at that code, since there are a couple of important things to note here.

First off, 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 what we're indicating here is 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, we could create a new method to handle the event:

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

End Sub
```

Even though we've changed the method name, and the names of the parameters, we are still accepting parameters of the same data types and we 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 we choose, but a single method can handle multiple events if we 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 raised by the .NET system class library 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 than can accept virtually any event raised by the class library.*

One common scenario where this is useful is where we have multiple instances of an object that raises events, such as two buttons on a form:

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

End Sub
```

Notice that we've modified the `Handles` clause to have a comma-separated list of events to handle. Either event will cause our method to run, giving us a central location to handle these events.

The WithEvents Keyword

The `WithEvents` keyword tells VB that we want to handle any events raised by the object within our code. For example:

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

The `WithEvents` keyword makes any events from an object available for our use, while the `Handles` keyword is used to link specific events to our methods so we can receive and handle them. This is true not only for controls on forms, but also for any objects that we 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, we'd get a syntax error when we 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 we will have used the `Event` keyword to declare at least one event as part of the interface for our class.

Raising Events

Our objects can raise events just like a control, and the code using our object can receive these events by using the `WithEvents` and `Handles` keywords. Before we can raise an event from our object, however, we need to declare the event within our class by using the `Event` keyword.

In our `Person` class, for instance, we may want to raise an event any time the `Walk` method is called. If we call this event `Walked`, we can add the following declaration to our `Person` class:

```
Public Class Person
  Private mstrName As String
  Private mdtBirthDate As Date
```

```
    Private mintTotalDistance As Integer
    Private colPhones As New Hashtable()
    Private mintAllergens As Integer

    Public Event Walked()
```

Our events can also have parameters – values that are provided to the code receiving the event. A typical button's `Click` event receives two parameters, for instance. In our `Walked` method, perhaps we want to also indicate the distance that was walked. We can do this by changing the event declaration:

```
    Public Event Walked(ByVal Distance As Integer)
```

Now that our event is declared, we can raise that event within our code where appropriate. In this case, we'll raise it within the `Walk` method – so any time that a `Person` object is instructed to walk, it will fire an event indicating the distance walked. Make the following change to the `Walk` method:

```
    Public Sub Walk(ByVal Distance As Integer)
      mintTotalDistance += Distance
      RaiseEvent Walked(Distance)
    End Sub
```

The `RaiseEvent` keyword is used to raise the actual event. Since our event requires a parameter, that value is passed within parentheses and will be delivered to any recipient that handles the event.

In fact, the `RaiseEvent` statement will cause the event to be delivered to all code that has our 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 – meaning that we 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. Once we call the `RaiseEvent` method, the event will be delivered to all listeners one after another until it is complete – there is no way for us to intervene and stop the process in the middle.

Receiving Events with WithEvents

Now that we've implemented an event within our `Person` class, we can write client code to declare an object using the `WithEvents` keyword. For instance, in our project's Form1 code module, we can write the following:

```
  Public Class Form1
    Inherits System.Windows.Forms.Form

    Private WithEvents mobjPerson As Person
```

By declaring the variable `WithEvents`, we are indicating that we want to receive any events raised by this object.

We can also choose to declare the variable without the WithEvents keyword, though, in that case, we would not receive events from the object as described here. Instead we would use the AddHandler method, which we'll discuss after we cover the use of WithEvents.

We 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

  mobjPerson = New Person()

End Sub
```

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

```
Private Sub OnWalk(ByVal Distance As Integer) Handles mobjPerson.Walked
  MsgBox("Person walked " & Distance)
End Sub
```

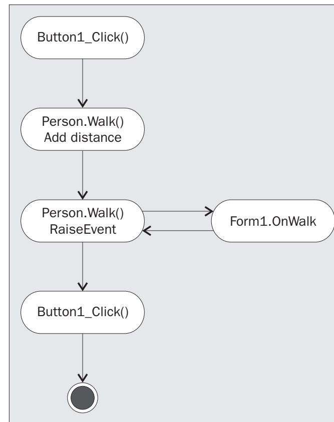
We're using the Handles keyword to indicate which event should be handled by this method. We're also receiving an Integer parameter. If the parameter list of our method doesn't match the list for the event, we'll get a compiler error indicating the mismatch.

Finally, we need to call the Walk method on our Person 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

  mobjPerson.Walk(42)

End Sub
```

When the button is clicked, we'll simply call the Walk method, passing an Integer value. This will cause the code in our class to be run – including the RaiseEvent statement. The result will be an event firing back into our form, since we declared the mobjPerson variable using the WithEvents keyword. Our OnWalk method will be run to handle the event, since it has the Handles clause linking it to the event.

The following diagram illustrates the flow of control:



The diagram illustrates how the code in the button's click event calls the `Walk` method, causing it to add to the total distance walked and then to raise its event. The `RaiseEvent` causes the `OnWalk` method in the form to be invoked and, once it is done, control returns to the `Walk` method in the object. Since we have no code in the `Walk` method after we call `RaiseEvent`, the control returns to the `Click` event back in the form, and then we're all done.

> *Many people have the misconception that events use multiple threads to do their work. This is not the case. Only one thread is involved in this process. Raising an event is much like making a method call, in that our existing thread is used to run the code in the event handler. This means our application's processing is suspended until the event processing is complete.*

Receiving Events with AddHandler

Now that we've seen how to receive and handle events using the `WithEvents` and `Handles` keywords, let's take a look at an alternative approach. We can use the `AddHandler` method to dynamically add event handlers through our code.

`WithEvents` and the `Handles` clause require that we declare both the object variable and event handler as we build our code, effectively creating a linkage that is compiled right into our code. `AddHandler`, on the other hand, creates this linkage at runtime, which can provide us with more flexibility. Before we get too deep into that however, let's see how `AddHandler` works.

In Form1, we can change the way our code interacts with the `Person` object – first eliminating the `WithEvents` keyword:

```
Private mobjPerson As Person
```

and then also eliminating the `Handles` clause:

```
Private Sub OnWalk(ByVal Distance As Integer)
  MsgBox("Person walked " & Distance)
End Sub
```

With these changes, we've eliminated all event handling for our object and so our form will no longer receive the event, even though the Person object raises it.

Now we 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 we've created our object, we 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
  mobjPerson = New Person()
  AddHandler mobjPerson.Walked, AddressOf OnWalk
End Sub
```

This single line of code does the same thing as our earlier use of WithEvents and the Handles clause – causing the OnWalk method to be invoked when the Walked event is raised from our Person object.

However, this linkage is done at runtime, and so we have more control over the process than we have otherwise. For instance, we could have extra code to decide *which* event handler to link up. Suppose we have another possible method to handle the event in the case that a message box is not desirable. Add this code to Form1:

```
Private Sub LogOnWalk(ByVal Distance As Integer)
  System.Diagnostics.Debug.WriteLine("Person walked " & Distance)
End Sub
```

Rather than popping up a message box, this version of the handler logs the event to the Output window in the IDE.

Now we can enhance our AddHandler code to decide which handler should be used – dynamically at runtime:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
  mobjPerson = New Person()
  If Microsoft.VisualBasic.Command = "nodisplay" Then
    AddHandler mobjPerson.Walked, AddressOf LogOnWalk
  Else
    AddHandler mobjPerson.Walked, AddressOf OnWalk
  End If
End Sub
```

If the word nodisplay is on the command line when our application is run, the new version of the event handler will be used – otherwise we'll continue to use the message box handler.

Constructor Methods

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. We've seen this used before – most notably in a regular Windows form, where the New method is used to hold any initialization code for the form.

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 only ever run once for an object. Of course, we 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 is far more powerful in VB.NET since we can accept parameter values as input to the method.*

We can implement a constructor in our classes as well – using it to initialize our objects as needed. This is as easy as implementing a `Public` method named `New`. Add the following to our `Person` class:

```
Public Sub New()
  Phone("home") = "555-1234"
  Phone("work") = "555-5678"
End Sub
```

In this example, we're simply using the constructor method to initialize the home and work phone numbers for any new `Person` object that is created.

Parameterized Constructors

We can also use constructors to allow parameters to be passed to our object as it is being created. This is done by simply adding parameters to the `New` method. For example, we can change the `Person` class as follows:

```
Public Sub New(ByVal Name As String, ByVal BirthDate As Date)
  mstrName = Name
  mdtBirthDate = BirthDate

  Phone("home") = "555-1234"
  Phone("work") = "555-5678"
End Sub
```

With this change, any time a `Person` object is created, we'll be provided with values for both the name and birth date. This changes how we can create a new `Person` object, however. Where we used to have code such as:

```
Dim myPerson As New Person()
```

Now we will have code such as:

```
Dim myPerson As New Person("Peter", "1/1/1960")
```

In fact, since our constructor expects these values, they are mandatory – any code wishing to create an instance of our `Person` class *must* provide these values. Fortunately, there are alternatives in the form of optional parameters and method overloading (which allows us to create multiple versions of the same method – each accepting a different parameter list – something we'll discuss later in the chapter).

Constructors with Optional Parameters

In many cases, we may want our constructor to accept parameter values for initializing new objects – but we also want to have the ability to create objects without providing those values. This is possible through method overloading, which we'll discuss later, or through the use of 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 we must provide default values for the optional parameters.

For instance, we can change our Person class as shown:

```
    Public Sub New(Optional ByVal Name As String = "", _
        Optional ByVal BirthDate As Date = #1/1/1900#)
      mstrName = Name
      mdtBirthDate = BirthDate

      Phone("home") = "555-1234"
      Phone("work") = "555-5678"
    End Sub
```

Here we've changed both the Name and BirthDate parameters to be optional, and we are providing default values for both of them. Now we have the option of creating a new Person object with or without the parameter values:

```
Dim myPerson As New Person("Peter", "1/1/1960")
```

or

```
Dim myPerson As New Person()
```

If we don't provide the parameter values then the default values of an empty String and 1/1/1900 will be used and our code will work just fine.

Termination and Cleanup

In the .NET environment, an object is destroyed and the memory and resources it consumes are reclaimed when there are no references remaining for the object.

As we discussed earlier in the chapter, when we are using objects, our variables actually hold a reference or pointer to the object itself. If we have code such as:

```
Dim myPerson As New Person()
```

we know that the myPerson variable is just a reference to the Person object we created. If we also have code like this:

```
Dim anotherPerson As Person
anotherPerson = myPerson
```

we know that the anotherPerson variable is also a reference to *the same object*. This means that this specific Person object is being referenced by two variables.

When there are *no* variables left referencing an object, it can be terminated by the .NET runtime environment. In particular, it is terminated and reclaimed by a mechanism called garbage collection, which we'll discuss shortly.

> **Unlike COM (and thus VB6), the .NET runtime does not use reference counting to determine when an object should be terminated. Instead, it uses a scheme known as garbage collection to terminate objects. This means that, in VB.NET, we do not have deterministic finalization, so it is not possible to predict exactly when an object will be destroyed.**

Before we get to garbage collection, however, let's review how we can eliminate references to an object.

We can explicitly remove a reference by setting our variable equal to `Nothing`, with code such as:

```
myPerson = Nothing
```

There are two schools of thought as to whether we should still explicitly set variables to `Nothing` *even when they fall out of scope. On one hand, we can save writing extra lines of code by allowing the variable to automatically be destroyed but, on the other hand, we can explicitly show our intent to destroy the object by setting it to* `Nothing` *manually.*

Perhaps most important is the fact that the garbage collection mechanism will sometimes reclaim our objects in the middle of our processing. This can only happen if our code doesn't use the object later in the method. Setting the variable to Nothing at the end of the method will prevent the garbage collection mechanism from proactively reclaiming our objects.

We can also remove a reference to an object by changing the variable to reference a different object. Since a variable can only point to one object at a time, it follows naturally that changing a variable to point at another object must cause it to no longer point to the first one. This means we can have code such as:

```
myPerson = New Person()
```

which causes the variable to point to a brand new object – thus releasing this reference to the prior object.

These are examples of *explicit* dereferencing. VB.NET also provides facilities for *implicit* dereferencing of objects when a variable goes out of scope. For instance, if we have a variable declared within a method, when that method completes the variable will be automatically destroyed – thus dereferencing any object to which it may have pointed. In fact, any time a variable referencing an object goes out of scope, the reference to that object is automatically eliminated.

This is illustrated by the following code:

```
Private Sub DoSomething()
  Dim myPerson As Person

  myPerson = New Person()
End Sub
```

Even though we didn't explicitly set the value of `myPerson` to `Nothing`, we know that the `myPerson` variable will be destroyed when the method is complete since it will fall out of scope. This process implicitly removes the reference to the `Person` object created within the routine.

Of course, another scenario where objects become dereferenced is when the application itself completes and is terminated. At that point, all variables are destroyed and so, by definition, all object references go away as well.

We discussed garbage collection and the `Finalize` method in Chapter 3. When we discussed these concepts, we mentioned that there was no automatic way to perform the cleanup when the final reference to an object is released, although implementing the `IDisposable` interface provides one solution. We'll investigate that solution now.

The IDisposable Interface

In some cases the `Finalize` behavior is not acceptable. If we have an object that is using some expensive or limited resource – such as a database connection, a file handle, or a system lock – we might need to ensure that the resource is freed as soon as the object is no longer in use.

To accomplish this, we can implement a method to be called by the client code to force our object to clean up and release its resources. This is not a perfect solution, but it is workable. The thing to remember is that this method is not called automatically by the .NET runtime environment, but instead must be called directly by the code using the object.

The .NET framework provides the `IDisposable` interface that formalizes the declaration of this cleanup method. We'll discuss creating and working with multiple interfaces in detail later so, for now, we'll just focus on the implementation of the `Dispose` method from a cleanup perspective.

Any class that derives from `System.ComponentModel.Component` automatically gains the `IDisposable` interface. This includes all of the forms and controls that are used in a Windows Forms UI, as well as various other classes within the .NET framework. For most of our custom classes, however, we'll need to implement the interface ourselves.

We can implement it in our `Person` class by adding the following code to the top of the class:

```
Public Class Person
  Implements IDisposable
```

This interface defines a single method – `Dispose` – that we need to implement in our class. It is implemented by adding the following code to the class:

```
  Private Sub Dispose() Implements IDisposable.Dispose
    colPhones = Nothing
  End Sub
```

In this case, we're using this method to release our reference to the `HashTable` object that the `colPhones` variable points to. While not strictly necessary, this illustrates how our code can release other objects when the `Dispose` method is called.

It is up to our client code to call this method at the appropriate time to ensure that cleanup occurs. Typically, we'll want to call the method as soon as we're done using the object.

This is not always as easy as it might sound. In particular, an object may be referenced by more than one variable and just because we're dereferencing the object from *one* variable doesn't mean it has been dereferenced by *all* the other variables. If we call the `Dispose` method while other references remain – our object may become unusable and may cause errors when invoked via those other references. There is no easy solution to this problem – so careful design is required in the case that we choose to use the `IDispose` interface.

In our application's Form1 code, we use the `OnLoad` method of the form to create an instance of the `Person` object. In the form's `OnClosed` method, we may want to make sure to clean up by disposing of the `Person` object. To do this, add the following code to the form:

```
Private Sub Form1_Closed(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Closed

  CType(mobjPerson, IDisposable).Dispose()

End Sub
```

The `OnClosed` method runs as the form is being closed, and so it is an appropriate place to do cleanup work.

Before we can dereference the `Person` object, however, we can now call its `Dispose` method. Since this method is part of a secondary interface (something we'll discuss more later), we need to use the `CType()` method to access that specific interface in order to call the method:

```
CType(mobjPerson(), IDisposable).Dispose()
```

`CType()` allows us to indicate the specific interface by which we want to access the object – in this case the `IDisposable` interface. Once we're using that interface, we can call the `Dispose` method to cause the object to do any cleanup before we release our reference:

```
mobjPerson() = Nothing
```

Once we've released the reference, we know that the garbage collection mechanism will eventually find and terminate the object – thus running its `Finalize` method. In the meantime, however, we've forced the object to do any cleanup immediately, so its resources are not consumed during the time between our release of the reference and the garbage collection terminating the object.

Advanced Concepts

So far we've seen how to work with objects, how to create classes with methods, properties, and events, and how to use constructors. We've also discussed how objects are destroyed within the .NET environment and how we can hook into that process to do any cleanup required by our objects.

Now let's move on to discuss some more complex topics and variations on what we've discussed so far. First, we'll cover some advanced variations in terms of the methods we can implement in our classes, including an exploration of the underlying technology behind events.

From there we'll move on to delegates, the difference between components and classes, and .NET attributes as they pertain to classes and methods.

Advanced Methods

So far, the methods we've worked with have been quite straightforward. They've either been `Sub` or `Function` routines. We've also discussed `Property` routines, which are a specialized type of method.

Now let's take a look at some advanced concepts that provide us with a great deal more power and capability as we work with methods.

Overloading Methods

Methods often accept parameter values. Our `Person` object's `Walk` method, for instance, accepts an `Integer` parameter:

```
Public Sub Walk(ByVal Distance As Integer)
  mintTotalDistance += Distance
  RaiseEvent Walked(Distance)
End Sub
```

Sometimes we may not want to require the parameter. To solve this issue we can use the `Optional` keyword to make the parameter optional:

```
Public Sub Walk(Optional ByVal Distance As Integer = 0)
  mintTotalDistance += Distance
  RaiseEvent Walked(Distance)
End Sub
```

This doesn't provide us with a lot of flexibility, however, since the optional parameter or parameters must always be the last ones in the list. Additionally, all this allows us to do is choose to pass or not to pass the parameter – suppose we want to do something fancier such as allow different data types, or even entirely different lists of parameters?

Method **overloading** provides exactly those capabilities. By overloading methods, we can create several methods of the *same name*, with each one accepting a different set of parameters or parameters of different data types.

As a simple example, instead of using the `Optional` keyword in our `Walk` method, we could use overloading. We'll keep our original `Walk` method, but we'll also add *another* `Walk` method that accepts a different parameter list. Change the code in our `Person` class back to:

```
Public Sub Walk(ByVal Distance As Integer)
  mintTotalDistance += Distance
  RaiseEvent Walked(Distance)
End Sub
```

Then we can create another method – a method with the same name, but with a different parameter list (in this case no parameters). Add this code to the class, without removing or changing the existing `Walk` method:

```
Public Sub Walk()
  RaiseEvent Walked(0)
End Sub
```

At this point we have *two* Walk methods. The only way to tell them apart is by the list of parameters each accepts – the first requiring a single Integer parameter, the second having no parameter.

> *There is an* Overloads *keyword as well. This keyword is not needed for simple overloading of methods as described here, but is required when combining overloading and inheritance. We'll discuss this in Chapter 6.*

Now we have the option of calling our Walk method in a couple different ways. We can call it with a parameter:

```
objPerson.Walk(42)
```

or without a parameter:

```
objPerson.Walk()
```

We can have any number of Walk methods in our class – as long as each individual Walk method has a different **method signature**.

Method Signatures

All methods have a signature, which is defined by the method name and the data types of its parameters.

```
Public Function CalculateValue() As Integer

End Sub
```

In this example, the signature is *f*().

The letter *f* is often used to indicate a method or function. It is appropriate here, because we don't care about the *name* of our function, only its parameter list is important.

If we add a parameter to the method, the signature will change. For instance, we could change the method to accept a Double:

```
Public Function CalculateValue(ByVal Value As Double) As Integer
```

Then the signature of the method is *f*(Double).

Notice that, in VB.NET, the return value is not part of the signature. We can't overload a Function routine by just having its return value's data type vary. It is the data types in the parameter list that must vary to utilize overloading.

Also make note that the *name* of the parameter is totally immaterial – only the data type is important. This means that the following methods have identical signatures:

```
Public Sub DoWork(ByVal X As Integer, ByVal Y As Integer)

Public Sub DoWork(ByVal Value1 As Integer, ByVal Value2 As Integer)
```

In both cases the signature is *f*(Integer, Integer).

Not only do the data types of the parameters define the method signature, but whether the parameters are passed ByVal or ByRef is also important. Changing a parameter from ByVal to ByRef will change the method signature.

Combining Overloading and Optional Parameters

Overloading is more flexible than using optional parameters, but optional parameters have the advantage that they can be used to provide default values as well as making a parameter optional.

We can combine the two concepts – overloading a method and also having one or more of those methods utilize optional parameters. Obviously, this sort of thing could get very confusing if overused, since we're employing two types of method "overloading" at the same time.

The Optional keyword causes a single method to effectively have two signatures. This means that a method declared as:

```
Public Sub DoWork(ByVal X As Integer, Optional ByVal Y As Integer = 0)
```

has two signatures at once: *f*(Integer, Integer) and *f*(Integer).

Because of this, when we use overloading along with optional parameters, our other overloaded methods cannot match *either* of these two signatures. However, as long as our other methods don't match either signature, we can use overloading as we discussed earlier. For instance, we could implement methods with the following different signatures:

```
Public Sub DoWork(ByVal X As Integer, _
      Optional ByVal Y As Integer = 0)
```

and

```
Public Sub DoWork(ByVal Data As String)
```

since there are no conflicting method signatures. In fact, with these two methods, we've really created three signatures:

- *f*(Integer, Integer)
- *f*(Integer)
- *f*(String)

The IntelliSense built into the VS.NET IDE will show that we have two overloaded methods – one of which has an optional parameter. This is different from if we'd created three different overloaded methods to match these three signatures – in which case the IntelliSense would list three variations on the method from which we can choose.

Overloading the Constructor Method

We can combine the concept of a constructor method with method overloading to allow for different ways of creating instances of our class. This can be a very powerful combination, as it allows a great deal of flexibility in object creation.

We've already explored how to use optional parameters in the constructor. Now let's change our implementation in the `Person` class to make use of overloading instead. Change the existing `New` method as follows:

```
Public Sub New(ByVal Name As String, ByVal BirthDate As Date)
  mstrName = Name
  mdtBirthDate = BirthDate

  Phone("home") = "555-1234"
  Phone("work") = "555-5678"
End Sub
```

With this change, we've returned to requiring the two parameter values be supplied.

Now add that second implementation as shown:

```
Public Sub New()
  Phone("home") = "555-1234"
  Phone("work") = "555-5678"
End Sub
```

This second implementation accepts no parameters – meaning that we can now create `Person` objects in two different ways – either with no parameters or by passing the name and birth date:

```
Dim myPerson As New Person()
```

or:

```
Dim myPerson As New Person("Fred", "1/11/60")
```

This type of capability is very powerful, as it allows us to define the various ways in which applications can create our objects. In fact, the VS.NET IDE takes this into account so, when we are typing the code to create an object, the IntelliSense tool tip will display the overloaded variations on the method – providing a level of automatic documentation for our class.

Shared Methods, Variables, and Events

So far, all of the methods we've built or used have been **instance methods** – methods that require us to have an actual instance of the class before they can be called. These methods have used instance variables or member variables to do their work – meaning that they have been working with a set of data that is unique to each individual object.

VB.NET allows us to create variables and methods that belong to the *class* rather than to any specific *object*. Another way to say this is that these variables and methods belong to *all* objects of a given class and are shared across all the instances of the class.

We can use the Shared keyword to indicate which variables and methods belong to the class rather than to specific objects. For instance, we may be interested in knowing the total number of Person objects created as our application is running – kind of a statistical counter.

Shared Variables

Since regular variables are unique to each individual Person object, they don't allow us to easily track the total number of Person objects ever created. However, if we had a variable that had a common value *across* all instances of the Person class, we could use that as a counter. Add the following variable declaration to our Person class:

```
Public Class Person
  Implements IDisposable

  Private Shared sintCounter As Integer
```

By using the Shared keyword, we are indicating that this variable's value should be shared across all Person objects within our application. This means that if one Person object makes the value be 42, all other Person objects will see the value as 42 – it is a shared piece of data.

> *We are using the letter "s" as a prefix to this variable rather than "m". The letter "m" is commonly used for member variables (or module variables), but this variable is not a member variable – it is a shared variable. Using a different prefix can help distinguish between member and shared variables within our code.*

We can now use this variable within our code. For instance, we can add code to the constructor method, New, to increment the variable so it acts as a counter – adding 1 each time a new Person object is created. Change the New methods as shown:

```
  Public Sub New()
    Phone("home") = "555-1234"
    Phone("work") = "555-5678"
    sintCounter += 1
  End Sub

  Public Sub New(ByVal Name As String, ByVal BirthDate As Date)
    mstrName = Name
    mdtBirthDate = BirthDate

    Phone("home") = "555-1234"
    Phone("work") = "555-5678"
    sintCounter += 1
  End Sub
```

The sintCounter variable will now maintain a value indicating the total number of Person objects created during the life of our application. We may want to add a property routine to allow access to this value by writing the following code:

```
  Public ReadOnly Property PersonCount() As Integer
    Get
      Return sintCounter
```

```
      End Get
    End Property
```

Notice that we're creating a regular property that returns the value of a shared variable. This is perfectly acceptable. As we'll see shortly, we could also choose to create a shared property to return the value.

Now we could write code to use our class as follows:

```
Dim myPerson As Person

myPerson = New Person()
myPerson = New Person()
myPerson = New Person()

MsgBox(myPerson.PersonCount)
```

The resulting display would show 3 – since we've created three instances of the `Person` class.

Shared Methods

We cannot only share variables across all instances of our class, but we can also share methods. Where a regular method or property belongs to each specific object, a shared method or property is common across all instances of the class.

There are a couple of ramifications to this approach.

First off, since shared methods don't belong to any specific object, they can't access any instance variables from any objects. The only variables available for use within a shared method are shared variables, parameters passed into the method, or variables declared locally within the method itself. If we attempt to access an instance variable within a shared method, we'll get a compiler error.

Also, since shared methods are actually part of the *class* rather than any *object*, we can write code to call them directly from the class – without having to create an instance of the class first.

For instance, a regular instance method is invoked from an object:

```
Dim myPerson As New Person()

myPerson.Walk(42)
```

but a shared method can be invoked directly from the class itself:

```
Person.SharedMethod()
```

This saves the effort of creating an object just to invoke a method, and can be very appropriate for methods that act on shared variables, or methods that act only on values passed in via parameters. We can also invoke a shared method from an object just like a regular method. Shared methods are flexible in that they can be called with or without creating an instance of the class first.

To create a shared method we again use the Shared keyword. For instance, the PersonCount property we created earlier could easily be changed to be a shared method instead:

```
Public Shared ReadOnly Property PersonCount() As Integer
  Get
    Return sintCounter
  End Get
End Property
```

Since this property returns the value of a shared variable, it is perfectly acceptable for it to be implemented as a shared method. With this change, we can now find out how many Person objects have ever been created without having to actually create a Person object first:

```
MsgBox(Person.PersonCount)
```

As another example, in our Person class we could create a method that compares the ages of two people. Add a shared method with the following code:

```
Public Shared Function CompareAge(ByVal Person1 As Person, _
    ByVal Person2 As Person) As Boolean

  Return Person1.Age > Person2.Age

End Function
```

This method simply accepts two parameters – each a Person – and returns True if the first is older than the second. The use of the Shared keyword indicates that this method doesn't require a specific instance of the Person class for us to use it.

Within this code, we are invoking the Age property on two separate objects – the objects passed as parameters to the method. It is important to recognize that we're not *directly* using any instance variables within the method, but rather are accepting two objects as parameters and are invoking methods on those objects.

To use this method, we can call it directly from the class:

```
If Person.CompareAge(myPerson1, myPerson2) Then
```

Alternately, we can also invoke it from any Person object:

```
Dim myPerson As New Person()

If myPerson.CompareAge(myPerson, myPerson2) Then
```

Either way, we're invoking the same shared method and we'll get the same behavior whether we call it from the class or a specific instance of the class.

Shared Properties

As with other types of methods, we can also have shared property methods. Properties follow the same rules as regular methods – they can interact with shared variables, but not member variables, and they can invoke other shared methods or properties, but can't invoke instance methods without first creating an instance of the class.

We can add a shared property to our `Person` class with the following code:

```
Public Shared ReadOnly Property RetirementAge() As Integer
  Get
    Return 62
  End Get
End Property
```

This simply adds a property to our class that indicates the global retirement age for all people. To use this value, we can simply access it directly from the class:

```
MsgBox(Person.RetirementAge)
```

Alternately, we can also access it from any `Person` object:

```
Dim myPerson As New Person()

MsgBox(myPerson.RetirementAge)
```

Either way, we're invoking the same shared property.

Shared Events

As with other interface elements, events can also be marked as `Shared`. For instance, we could declare a shared event in the `Person` class such as:

```
Public Shared Event NewPerson()
```

Shared events can be raised from both instance methods and shared methods. Regular events can not be raised by shared methods. Since shared events can be raised by regular methods, we can raise this one from the constructors in the Person class:

```
Public Sub New()
  Phone("home") = "555-1234"
  Phone("work") = "555-5678"
  sintCounter += 1
  RaiseEvent NewPerson()
End Sub

Public Sub New(ByVal Name As String, ByVal BirthDate As Date)
  mstrName = Name
  mdtBirthDate = BirthDate

  Phone("home") = "555-1234"
```

```
    Phone("work") = "555-5678"
    sintCounter += 1
    RaiseEvent NewPerson()
  End Sub
```

The interesting thing about receiving shared events is that we can get them from either an object, like a normal event, or from the *class* itself. For instance, we can use the `AddHandler` method in our form's code to catch this event directly from the Person class.

First let's add a method to the form to handle the event:

```
  Private Sub OnNewPerson()
    MsgBox("new person " & Person.PersonCount)
  End Sub
```

Then, in the form's `Load` event, add a statement to link the event to this method:

```
  Private Sub Form1_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load

    AddHandler Person.NewPerson, AddressOf OnNewPerson

    mobjPerson = New Person()
    If Microsoft.VisualBasic.Command = "nodisplay" Then
      AddHandler mobjPerson.Walked, AddressOf LogOnWalk
    Else
      AddHandler mobjPerson.Walked, AddressOf OnWalk
    End If
  End Sub
```

Notice that we are using the *class* rather than any specific object in the `AddHandler` statement. We could use an object as well – treating this like a normal event, but this illustrates how a class itself can raise an event.

When we run the application now, any time a `Person` object is created we'll see this event raised.

Delegates

There are times when it would be nice to be able to pass a procedure as a parameter to a method. The classic case is when building a generic sort routine, where we not only need to provide the data to be sorted, but we need to provide a comparison routine appropriate for the specific data.

It is easy enough to write a sort routine that sorts `Person` objects by name, or to write a sort routine that sorts `SalesOrder` objects by sales date. However, if we want to write a sort routine that can sort any type of object based on arbitrary sort criteria, that gets pretty difficult. At the same time, it would be nice to do, since some sort routines can get very complex and it would be nice to reuse that code without having to copy-and-paste it for each different sort scenario.

By using delegates, we can create such a generic routine for sorting – and in so doing we can see how delegates work and can be used to create many other types of generic routines.

The concept of a **delegate** formalizes the process of declaring a routine to be called and calling that routine.

> *The underlying mechanism used by the .NET environment for callback methods is the delegate. VB.NET uses delegates behind the scenes as it implements the `Event`, `RaiseEvent`, `WithEvents`, and `Handles` keywords.*

Declaring a Delegate

In our code, we can declare what a delegate procedure must look like from an interface standpoint. This is done using the `Delegate` keyword. To see how this can work, let's create a routine to sort any kind of data.

To do this, we'll declare a delegate that defines a method signature for a method that compares the value of two objects and returns a Boolean indicating whether the first object has a larger value that the second object. We'll then create a sort algorithm that uses this generic comparison method to sort data. Finally, we'll create an actual method that *implements* the comparison and we'll pass the address of that method to the sort routine.

Add a new module to our project by choosing the Project | Add Module menu option. Name the module `Sort.vb` and then add the following code:

```
Module Sort

  Public Delegate Function Compare(ByVal v1 As Object, ByVal v2 As Object) _
    As Boolean

End Module
```

This line of code does something interesting. It actually defines a method signature as a *data type*. This new data type is named `Compare` and it can be used within our code to declare variables or parameters that will be accepted by our methods. A variable or parameter declared using this data type can actually hold the address of a method that matches the defined method signature – and we can then invoke that method by using the variable.

Any method with the signature:

f`(Object, Object)`

Can be viewed as being of type `Compare`.

Using the Delegate Data Type

We can write a routine that accepts this data type as a parameter – meaning that anyone calling our routine must pass us the address of a method that conforms to this interface. Add the following sort routine to the code module:

```
  Public Sub DoSort(ByVal theData() As Object, ByVal GreaterThan As Compare)
    Dim outer As Integer
    Dim inner As Integer
    Dim temp As Object

    For outer = 0 To UBound(theData)
      For inner = outer + 1 To UBound(theData)
```

```
        If GreaterThan.Invoke(theData(outer), theData(inner)) Then
          temp = theData(outer)
          theData(outer) = theData(inner)
          theData(inner) = temp
        End If
      Next
    Next
  End Sub
```

The GreaterThan parameter is a variable that holds the address of a method matching the method signature defined by our Compare delegate. The address of any method with a matching signature can be passed as a parameter to our Sort routine.

Note the use of the Invoke method, which is the way a delegate is called from our code. Also note that the routine deals entirely with the generic System.Object data type rather than with any specific type of data. The specific comparison of one object to another is left to the delegate routine that is passed in as a parameter.

Implementing a Delegate Method

All that remains is to actually create the implementation of the delegate routine and call our sort method. On a very basic level, all we need to do is create a method that has a matching method signature. For instance, we could create a method such as:

```
Public Function PersonCompare(ByVal Person1 As Object, _
    ByVal Person2 As Object) As Boolean

End Function
```

The method signature of this method exactly matches that which we defined by our delegate earlier:

```
Compare(Object, Object)
```

In both cases, we're defining two parameters of type Object.

Of course, there's more to it than simply creating the stub of a method. We know that the method needs to return a value of True if its first parameter is greater than the second parameter, but otherwise should be written to deal with some specific type of data.

The Delegate statement defines a data type based on a specific method interface. To call a routine that expects a parameter of this new data type, it must pass us the address of a method that conforms to the defined interface.

To conform to the interface, a method must have the same number of parameters with the same data types as we've defined in our Delegate statement. Additionally, the method must provide the same return type as defined. The actual name of the method doesn't matter – it is the number, order, and data type of the parameters and return value that count.

To find the address of a specific method, we can use the AddressOf operator. This operator returns the address of any procedure or method, allowing us to pass that value as a parameter to any routine that expects a delegate as a parameter.

Our Person class already has a shared method named CompareAge that generally does what we want. Unfortunately, it accepts parameters of type Person rather than of type Object as required by the Compare delegate. We can use method overloading to solve this problem.

Create a second implementation of CompareAge that accepts parameters of type Object as required by the delegate, rather than of type Person as we have in the existing implementation:

```
Public Shared Function CompareAge(ByVal Person1 As Object, _
    ByVal Person2 As Object) As Boolean

  Return CType(Person1, Person).Age > CType(Person2, Person).Age

End Function
```

This method simply returns True if the first Person object's age is greater than the second. The routine accepts two Object parameters rather than specific Person type parameters, so we have to use the CType() method to access those objects as type Person. We accept the parameters as type Object because that is what is defined by the Delegate statement. We are matching its method signature:

f(Object, Object)

Since this method's parameter data types and return value match the delegate, we can use it when calling the sort routine. Place a button on the form and write the following code behind that button:

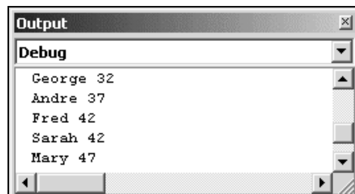
```
Private Sub Button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles button2.Click

  Dim myPeople(4) As Person

  myPeople(0) = New Person("Fred", #7/9/1960#)
  myPeople(1) = New Person("Mary", #1/21/1955#)
  myPeople(2) = New Person("Sarah", #2/1/1960#)
  myPeople(3) = New Person("George", #5/13/1970#)
  myPeople(4) = New Person("Andre", #10/1/1965#)

  DoSort(myPeople, AddressOf Person.CompareAge)
End Sub
```

This code creates an array of Person objects and populates them. It then calls the DoSort routine from our module, passing the array as the first parameter and the address of our shared CompareAge method as the second. To display the contents of the sorted array in the IDE's output window, we can add the following code:

```
Private Sub button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles button2.Click

  Dim myPeople(4) As Person

  myPeople(0) = New Person("Fred", #7/9/1960#)
  myPeople(1) = New Person("Mary", #1/21/1955#)
  myPeople(2) = New Person("Sarah", #2/1/1960#)
```

```
    myPeople(3) = New Person("George", #5/13/1970#)
    myPeople(4) = New Person("Andre", #10/1/1965#)

    DoSort(myPeople, AddressOf Person.CompareAge)

    Dim myPerson As Person

    For Each myPerson In myPeople
      System.Diagnostics.Debug.WriteLine(myPerson.Name & " " & myPerson.Age)
    Next
  End Sub
```

When we run the application and click the button, the output window will display a list of the people, sorted by age:



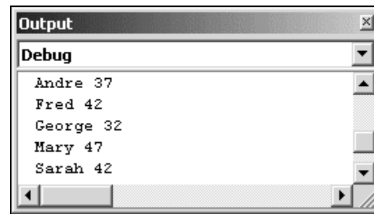
What makes this whole thing very powerful is that we can change the comparison routine without changing the sort mechanism. Simply add another comparison routine to the `Person` class:

```
  Public Shared Function CompareName(ByVal Person1 As Object, _
      ByVal Person2 As Object) As Boolean

    Return CType(Person1, Person).Name > CType(Person2, Person).Name

  End Function
```

and then change the code behind the button on the form to use that alternate comparison routine:

```
  Private Sub button2_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles button2.Click

    Dim myPeople(4) As Person

    myPeople(0) = New Person("Fred", #7/9/1960#)
    myPeople(1) = New Person("Mary", #1/21/1955#)
    myPeople(2) = New Person("Sarah", #2/1/1960#)
    myPeople(3) = New Person("George", #5/13/1970#)
    myPeople(4) = New Person("Andre", #10/1/1965#)

    DoSort(myPeople, AddressOf Person.CompareName)

    Dim myPerson As Person

    For Each myPerson In myPeople
      System.Diagnostics.Debug.WriteLine(myPerson.Name & " " & myPerson.Age)
    Next
  End Sub
```

When we run this updated code, we'll find that our array contains a set of data sorted by name rather than by age:



By simply creating a new compare routine and passing it as a parameter, we can entirely change the way that the data is sorted. Better still, this sort routine can operate on any type of object, as long as we provide an appropriate delegate method that knows how to compare that type of object.

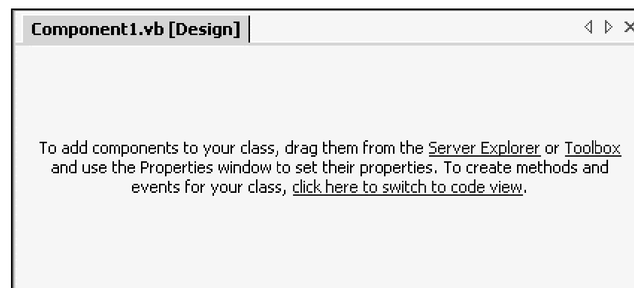
Classes vs. Components

VB.NET has another concept that is very similar to a class – the component. In fact, we can pretty much use a component and a class interchangeably, though there are some differences that we'll discuss.

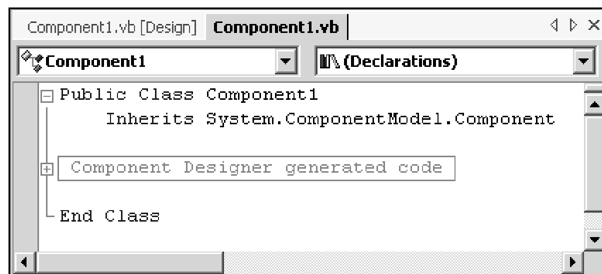
A component is really little more than a regular class, but it is one that supports a graphical designer within the VB.NET IDE. This means we can use drag-and-drop to provide the code in our component with access to items from the Server Explorer or from the Toolbox.

To add a component to a project, select the Project | Add Component menu option, give the component a name, and click Open in the Add New Item dialog.

When we add a class to our project we are presented with the code window. When we add a *component* on the other hand, we are presented with a graphical designer surface, much like what we'd see when adding a Web Form to the project:



If we switch to the code view (by right-clicking in the designer and choosing View Code), we will see the code that is created for us automatically:



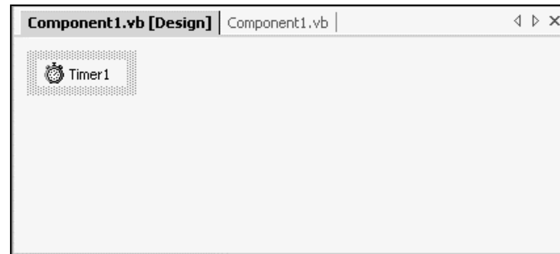
This isn't a lot more code than we'd see with a regular class, though there certainly are differences. First off, we see that this class inherits from `System.ComponentModel.Component`. While we'll discuss the concepts of inheritance in Chapters 6 and 7, it is important to note here that this `Inherits` line is what brings in all the support for the graphical designer we just saw.

There's also a collapsed region of code in a component. This region contains code generated by the graphical designer. Here's a quick look at what is included by default:

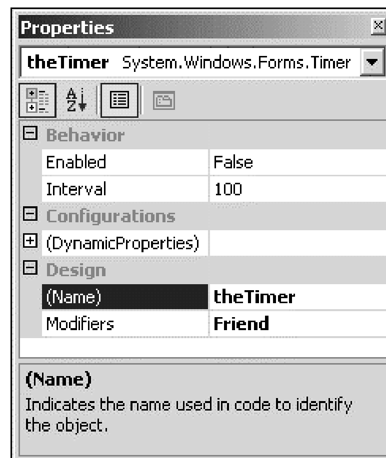
```
#Region " Component Designer generated code "

    Public Sub New(Container As System.ComponentModel.IContainer)
        MyClass.New()

        'Required for Windows.Forms Class Composition Designer support
        Container.Add(me)
    End Sub

    Public Sub New()
        MyBase.New()

        'This call is required by the Component Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Component overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Component Designer
    Private components As System.ComponentModel.IContainer

    'NOTE: The following procedure is required by the Component Designer
    'It can be modified using the Component Designer.
```

```
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        components = New System.ComponentModel.Container()
    End Sub

#End Region
```

As it stands, this code does very little beyond creating a single `Container` class object. However, if we switch the view back to the designer, we can drag-and-drop items onto our component. For instance, in the Toolbox there is a Components tab, which has entries for a variety of useful items such as a MessageQueue, a DirectoryEntry, and so forth. If we drag-and-drop a Timer (from the Components tab of the Toolbox) onto our component, it will be displayed in the designer:



From here, we can set its properties using the standard Properties window in the IDE, just like we would for a control on a form. For instance, we can set its `Name` property to `theTimer`:



If we now return to the code window and look at the automatically generated code, we'll see that the region now includes code to declare, create, and initialize the `Timer` object:

```
#Region " Component Designer generated code "

    Public Sub New(Container As System.ComponentModel.IContainer)
        MyClass.New()
```

```
        'Required for Windows.Forms Class Composition Designer support
        Container.Add(me)
    End Sub

    Public Sub New()
        MyBase.New()

        'This call is required by the Component Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Component overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Component Designer
    Private components As System.ComponentModel.IContainer

    'NOTE: The following procedure is required by the Component Designer
    'It can be modified using the Component Designer.
    'Do not modify it using the code editor.
  Friend WithEvents theTimer As System.Windows.Forms.Timer
  <System.Diagnostics.DebuggerStepThrough()> _
  Private Sub InitializeComponent()
    Me.components = New System.ComponentModel.Container()
    Me.theTimer = New System.Windows.Forms.Timer(Me.components)

  End Sub

#End Region
```

Normally, we don't really care about the fact that this code was generated. Rather, what is important is that we now automatically, simply by dragging and dropping and setting some properties, have access to a `Timer` object named `theTimer`.

This means that we can write code within our component, just like we might in a class, to use this object:

```
  Public Sub Start()
    theTimer.Enabled = True
  End Sub

  Public Sub [Stop]()
    theTimer.Enabled = False
```

```
    End Sub

    Private Sub theTimer_Elapsed(ByVal sender As System.Object, _
      ByVal e As System.Timers.ElapsedEventArgs) Handles theTimer.Elapsed

      ' do work

    End Sub
```

Here we can see that, with a simple drag-and-drop operation, we've gained access to a variable called `theTimer` referencing a `Timer` object, and we are able to create methods that interact with and use that object much like we would with a control dropped onto a form.

For the most part, we can use a component interchangeably with a basic class, but the use of a component incurs some extra overhead that a basic class does not, since it inherits all the functionality of `System.ComponentModel.Component`.

Summary

VB.NET offers us a fully object-oriented language with all the capabilities we would expect. In this chapter, we've explored the basic concepts around classes and objects, as well as the separation of interface from implementation and data.

We've seen how to use the `Class` keyword to create classes, and how those classes can be instantiated into specific objects – each one an instance of the class. These objects have methods and properties that can be invoked by client code, and can act on data within the object stored in member or instance variables.

We also explored some more advanced concepts, including method overloading, shared or static variables and methods, and the use of delegates. Finally, we wrapped up with a brief discussion of attributes and how they can be used to affect the interaction of our class or our methods with the .NET environment.

In Chapter 6, we'll continue our discussion of object syntax as we explore the concept of inheritance and all the syntax that enables inheritance within VB.NET. We will also walk through the creation, implementation, and use of multiple interfaces – a powerful concept that allows our objects to be used in different ways depending on the interface chosen by the client application.

Then, in Chapter 7, we'll wrap up our discussion of objects and object-oriented programming by applying all of this syntax. We'll discuss the key object-oriented concepts of abstraction, encapsulation, polymorphism, and inheritance and see how they all tie together to provide a powerful way of designing and implementing applications.