5 Inheritance

MANY PROGRAMMERS HAVE long considered inheritance to be one of the most significant design features of OOP. Inheritance was made popular more than two decades ago by languages such as C++ and Smalltalk. Since then, new languages (e.g., Java) have come along and refined the features and syntax for using inheritance. Now with the emergence of the .NET Framework, Microsoft has designed a platform from the ground up that offers support for what is arguably one of the most elegant forms of inheritance to date.

The more you use the .NET Framework, the more you will realize just how extensively it takes advantage of inheritance. For example, as discussed in Chapter 3, the CTS relies heavily on inheritance. When you use the Windows Forms package, your new forms will inherit from an existing form-based class in the FCL. When you use ASP.NET, your Web pages and Web services will inherit from preexisting classes in the FCL. As a developer building applications or component libraries based on the .NET Framework, you will find that familiarity with inheritance is an absolute necessity.

Inheriting from a Class

Inheritance allows you to derive a new class from an existing class. Suppose that class C inherits from class B. Then class B is typically called the *base*

class, and class C is called the *derived* class. Note that this terminology is not necessarily standard; some refer to B and C as *superclass* and *subclass*, or as *parent* and *child* classes, respectively. This book will stick with the terms "base" and "derived."

A derived class definition automatically inherits the implementation and the programming contract of its base class. The idea is that the derived class starts with a reusable class definition and modifies it by adding more members or by changing the behavior of existing methods and properties.

One of the primary reasons for designing with inheritance is that it provides an effective means for reusing code. For example, you can define a set of fields, methods, and properties in one class, and then use inheritance to reuse this code across several other classes. Inheritance is particularly beneficial in scenarios that require multiple classes with much in common but in which the classes are all specialized in slightly different ways. Examples include the following familiar categories: employees (administrative, technical, sales), database tables (customers, orders, products), and graphical shapes (circle, rectangle, triangle).

LISTING 5.1: A simple base class

```
'*** base class
Public Class Human

  '*** private implementation
  Private m_Name As String

  '*** public members
  Public Property Name() As String
    '*** provide controlled access to m_Name
  End Property

  Public Function Speak() As String
    Return "Hi, I'm a human named " & m_Name
  End Function

End Class
```

Let's start by looking at a simple example to introduce the basic syntax of inheritance. Listing 5.1 defines a class named `Human` that we would like to use as a base class. In Visual Basic .NET, when you want to state explicitly that one class inherits from another, you follow the name of the class

with the `Inherits` keyword and the name of the base class. For example, here are two derived class definitions for `Manager` and `Programmer`, both with `Human` as their base class:

```
'*** first derived class
Public Class Manager
       Inherits Human
  '*** code to extend Human definition
End Class

'*** second derived class
Public Class Programmer : Inherits Human
  '*** code to extend Human definition
End Class
```

For the `Programmer` class, the preceding code fragment uses a colon instead of an actual line break between the class name and the `Inherits` keyword. As you might recall from earlier versions of Visual Basic, the colon acts as the line termination character. This colon-style syntax is often preferred because it improves readability by keeping the name of a derived class on the same line as the name of its base class.

The `Human` class serves as a base class for both the `Manager` class and the `Programmer` class. It's now possible to write implementations for these classes that are quite different as well as to add new members to each class to further increase their specialization. Nevertheless, these two classes will always share a common set of implementation details, and both support a unified programming contract defined by the `Human` class.

By default, every class you create in Visual Basic .NET can be inherited by other classes. If for some reason you don't want other programmers to inherit from your class, you can create a *sealed* class. A sealed class is defined in the Visual Basic .NET language using the keyword `NotInheritable`. The compiler will generate a compile-time error whenever a programmer attempts to define a class that inherits from such a sealed class:

```
Public NotInheritable Class Monkey  '*** sealed class definition
  '*** implementation
End Class

Public Class Programmer : Inherits Monkey  '*** compile-time error!
End Class
```

Figure 5.1 shows a common design view of class definitions known as an *inheritance hierarchy*. The `Human` class is located at the top of the hierarchy. The `Manager` class and the `Programmer` class inherit from the `Human` class and are consequently located directly below it in the inheritance hierarchy. Notice the direction of the arrows when denoting inheritance.

As shown in Figure 5.1, an inheritance hierarchy can be designed with multiple levels. Consider the two classes at the bottom of the hierarchy in Figure 5.1, `SeniorProgrammer` and `JuniorProgrammer`. These classes have been defined with the `Programmer` class as their base class. As a consequence, they inherit indirectly from the `Human` class. Thus a class in a multilevel inheritance hierarchy inherits from every class reachable by following the inheritance arrows upward.

An important design rule should always be applied when designing with inheritance. Two classes that will be related through inheritance should be able to pass the *is-a* test. In short, the test goes like this: If it makes sense to say "C is a B," then it makes sense for class C to inherit from class B. If such a sentence doesn't make sense, then you should reconsider the use of inheritance in this situation.

For example, you can correctly say that a programmer "is a" human. You can also correctly say that a senior programmer "is a" programmer. As you can see, the purpose of the is-a test is to ensure that a derived class is designed to model a more specialized version of whatever entity the base class is modeling.

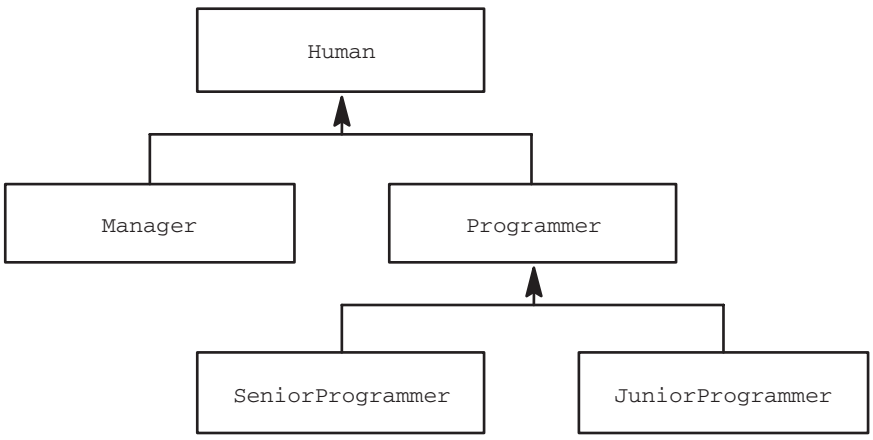

FIGURE 5.1: An inheritance hierarchy

You should try never to establish an inheritance relationship between two classes that cannot pass the is-a test. Imagine you saw a novice programmer trying to make the `Bicycle` class inherit from the `Wheel` class. You should intervene because you cannot correctly say that a bicycle "is a" wheel. You can correctly say that a bicycle "has a" wheel (or two), but that relationship calls for a different design technique that doesn't involve inheritance. When you determine that two entities exhibit the "has a" relationship, the situation most likely calls for a design using *containment*, in which the `Wheel` class is used to define fields within the `Bicycle` class. In other words, the `Bicycle` class *contains* the `Wheel` class.

Base Classes in the .NET Framework

Now that you've seen some of the basic principles and the syntax for using inheritance, it's time to introduce a few important rules that have been imposed by the .NET Common Type System (CTS).

The first rule is that you cannot define a class that inherits directly from more than one base class. In other words, the CTS does not support *multiple inheritance*. It is interesting to note that the lack of support for multiple inheritance in the .NET Framework is consistent with the Java programming language.

The second rule imposed by the CTS is that you cannot define a class that doesn't have a base class. This rule might seem somewhat confusing at first, because you can write a valid class definition in Visual Basic .NET (or in C#) that doesn't explicitly declare a base class. A little more explanation is required to clarify this point.

When you define a class without explicitly specifying a base class, the compiler automatically modifies the class definition to inherit from the `Object` class in the FCL. Once you understand this point, you can see that the following two class definitions have the same base class:

```
'*** implicitly inherit from Object
Public Class Dog
End Class

'*** explicitly inherit from Object
Public Class Cat : Inherits System.Object
End Class
```

These two CTS-imposed rules of inheritance can be summarized as follows: Every class (with the exception of the `Object` class) has exactly one base class. Of course, it's not just classes that have base types. For example, every structure and every enumeration also has a base type. Recall from Chapter 3 that the inheritance hierarchy of the CTS is *singly rooted* because the system-defined `Object` class serves as the ultimate base type (see Figure 3.1). Every other class either inherits directly from the `Object` class or inherits from another class that inherits (either directly or indirectly) from the `Object` class.

Inheriting Base Class Members

Although a derived class inherits the members of its base class, the manner in which certain kinds of members are inherited isn't completely intuitive. While the way things work with fields, methods, and properties is fairly straightforward, the manner in which constructors are inherited brings up issues that are a bit more complex.

Inheritance and Fields, Methods, and Properties

Every field, method, and property that is part of the base class definition is inherited by the derived class definition. As a result, each object created from a derived type carries with it all the states and behaviors that are defined by its base class. However, whether code in a derived class has access to the members inherited from its base class is a different matter altogether.

As mentioned in Chapter 4, each member of a class is defined with a level of accessibility that determines whether other code may access it. There are five levels of accessibility:

- *Private.* A base class member defined with the `Private` access modifier is not accessible to either code inside the derived class or client-side code using either class.
- *Protected.* A base class member defined with the `Protected` access modifier is accessible to code inside the derived class but is not accessible to client-side code. Private and protected members of a

base class are similar in that they are not accessible to client-side code written against either the base class or the derived class.

- *Public.* A base class member defined with the `Public` access modifier is accessible to code inside the derived class as well as all client-side code. Public members are unlike private and protected members in that they add functionality to the programming contract that a derived class exposes to its clients.
- *Friend.* A member that is defined with the `Friend` access modifier is accessible to all code inside the containing assembly but inaccessible to code in other assemblies. The friend level of accessibility is not affected by whether the accessing code is part of a derived class.
- *Protected Friend.* The protected friend level of accessibility is achieved by combining the `Protected` access modifier with the `Friend` access modifier. A protected friend is accessible to all code inside the containing assembly *and* to code within derived classes (whether the derived class is part of the same assembly or not).

Listing 5.2 presents an example that summarizes the discussion of what is legal and what is not legal with respect to accessing fields of varying levels of accessibility.

LISTING 5.2: The meaning of access modifiers in the presence of inheritance

```
'*** base class
Public Class BettysBaseClass
  Private Field1 As Integer
  Protected Field2 As Integer
  Public Field3 As Integer
End Class

'*** derived class
Public Class DannysDerivedClass : Inherits BettysBaseClass
  Public Sub Method1()
    Me.Field1 = 10  '*** illegal (compile-time error)
    Me.Field2 = 20  '*** legal
    Me.Field3 = 30  '*** legal
  End Sub
End Class

'*** client-side code
Module BobsApp
```

continues

```
  Public Sub Main()
    Dim obj As New DannysDerivedClass()
    obj.Field1 = 10  '*** illegal (compile-time error)
    obj.Field2 = 20  '*** illegal (compile-time error)
    obj.Field3 = 30  '*** legal
  End Sub
End Module
```

Now that we have outlined the rules of member accessibility, it's time to discuss how to properly encapsulate base class members when designing for inheritance. *Encapsulation* is the practice of hiding the implementation details of classes and assemblies from other code. For example, a protected member is encapsulated from client-side code. A private member is encapsulated from client-side code and derived classes. A friend member is encapsulated from code in other assemblies.

Imagine you are designing a component library that you plan to sell to other companies. You will update this component library from time to time and send the newest version to your customers. If your design involves distributing base classes that other programmers will likely extend through the use of inheritance, you need to think very carefully through the issues of defining various base class members as private versus protected.

Any member defined as private is fully encapsulated and can be modified or removed without violating the original contract between a base class and any of its derived classes. In contrast, members defined as protected are a significant part of the contract between a base class and its derived classes. Modifying or removing protected members can introduce breaking changes to your customer's code.

To keep your customers happy, you must devise a way to maintain and evolve the base classes in your component library without introducing breaking changes. A decade's worth of experience with inheritance has told the software industry that this challenge can be very hard to meet.

When authoring base classes, it's critical to start thinking about *versioning* in the initial design phase. You must determine how easy (or how difficult) it will be to modify derived classes if modifications to base classes cause breaking changes. It helps to design with the knowledge that it's a common mistake to underestimate the importance of encapsulating base class members from derived classes.

Another important consideration is whether it makes sense to use inheritance across assembly boundaries. While the compilers and the plumbing of the CLR are more than capable of fusing a base class implementation from one assembly together with the derived class implementation in a second assembly, you should recognize that versioning management grows ever more difficult as the scope of the inheritance increases.

That doesn't mean that you should never use cross-assembly inheritance. Many experienced designers have employed this strategy very effectively. For example, when you leverage one of the popular .NET frameworks such as Windows Forms or ASP.NET, you're required to create a class in a user-defined assembly that inherits from a class in a system-defined assembly. But understand one thing: The designers at Microsoft who created these frameworks thought long and hard about how to maintain and evolve their base classes without introducing breaking changes to your code.

If you plan to create base classes for use by programmers in other development teams, you must be prepared to think through these same issues. It is naive to ponder encapsulation only in terms of stand-alone classes, and only in terms of a single version. Inheritance and component-based development make these issues much more complex. They also make mistakes far more costly. In general, you shouldn't expose base class members to derived classes and/or other assemblies if these members might ever change in name, type, or signature. Following this simple rule will help you maintain backward compatibility with existing derived classes while evolving the implementation of your base classes.

Inheritance and Constructors

The way in which constructors are inherited isn't as obvious as for other kinds of base class members. From the perspective of a client attempting to create an object from the derived class, the derived class definition does not inherit any of the constructors defined in the base class. Instead, the derived class must contain one or more of its own constructors to support object instantiation. Furthermore, each constructor defined in a derived class must call one of the constructors in its base class before performing any of its own initialization work.

Recall from Chapter 4 that the Visual Basic .NET compiler will automatically create a public default constructor for any class definition that does not define a constructor of its own. This default constructor also contains code to call the default constructor of the base class. As an example, consider the following class definition:

```
'*** a class you write
Public Class Dog
End Class
```

Once compiled, the definition of this class really looks like this:

```
'*** code generated by compiler
Public Class Dog : Inherits System.Object

  '*** default constructor generated by compiler
  Public Sub New()
    MyBase.New()  '*** call to default constructor in System.Object
  End Sub

End Class
```

As this example reveals, the compiler will generate the required constructor automatically along with a call to the base class's default constructor. But what about the situation in which the base class does not have an accessible default constructor? In such a case, the derived class definition will not compile because the automatically generated default constructor is invalid. As a result, the author of the derived class must provide a constructor of his or her own, with an explicit call to a constructor in the base class.

The only time a derived class author can get away with not explicitly defining a constructor is when the base class provides an accessible default constructor. As it turns out, the `Object` class contains a public default constructor, which explains why you don't have to explicitly add a constructor to a class that inherits from the `Object` class. Likewise, you don't have to explicitly add a constructor to a class that inherits from another class with an accessible default constructor.

Sometimes, however, you must inherit from a class that doesn't contain a default constructor. Consider the following code, which includes a revised definition of the `Human` class discussed earlier. In particular, note that the `Human` class now contains a parameterized constructor:

```
Public Class Human
  Protected m_Name As String

  Public Sub New(ByVal Name As String)
    m_Name = Name
  End Sub

  Public Function Speak() As String
    Return "Hi, I'm a human named " & m_Name
  End Function
End Class

Public Class Programmer : Inherits Human
  '*** this class definition will not compile
End Class
```

Because this definition contains a single parameterized constructor, the compiler doesn't automatically add a default constructor to class `Human`. As a result, when you try to define the `Programmer` class without an explicit constructor, your code will not compile because Visual Basic .NET cannot generate a valid default constructor. To make the `Programmer` class compile, you must add a constructor that explicitly calls an accessible constructor defined in the `Human` class:

```
Public Class Programmer : Inherits Human

  Public Sub New(Name As String)
    MyBase.New(Name) '*** call to base class constructor
    '*** programmer-specific initialization goes here
  End Sub

End Class
```

As shown in the preceding code, you make an explicit call to a base class constructor by using `MyBase.New` and passing the appropriate list of parameters. Note that when you explicitly call a base class constructor from a derived class constructor, you can do it only once and the call must be the first statement in the constructor's implementation.

Of course, you never *have* to rely on compiler-generated calls to the default constructor. In some cases you might prefer to call a different constructor. An explicit call to `MyBase.New` can always be used at the top of a derived class constructor to call the exact base class constructor you want. Some programmers even add explicit calls to the default constructor by

using `MyBase.New`. Even though such calls can be automatically generated by the compiler, making these calls explicit can serve to make your code self-documenting and easier to understand.

Let's take a moment and consider the sequence in which the constructors are executed during object instantiation by examining the scenario where a client creates an object from the `Programmer` class using the `New` operator. When the client calls `New`, a constructor in the `Programmer` class begins to execute. Before this constructor can do anything interesting, however, it must call a constructor in the `Human` class. The constructor in the `Human` class faces the same constraints. Before it can do anything interesting, it must call the default constructor of the `Object` class.

The important observation is that constructors execute in a chain starting with the least-derived class (i.e., `Object`) and ending with the most-derived class. The implementation for the constructor of the `Object` class always runs to completion first. In the case of creating a new `Programmer` object, when the constructor for the `Object` class completes, it returns and the constructor for the `Human` class next runs to completion. Once the constructor for the `Human` class returns, the constructor for the `Programmer` class runs to completion. After the entire chain of constructors finishes executing, control finally returns to the client that started the sequence by calling the `New` operator on the `Programmer` class.

Limiting Inheritance to the Containing Assembly

Now that you understand the basics of how constructors must be coordinated between a base class and its derived classes, let's explore a useful design technique that prevents other programmers in other assemblies from inheriting from your base classes. The benefit of using this technique is that you can take advantage of inheritance within your assembly, but prevent cross-assembly inheritance. This approach eliminates the need to worry about how changes to the protected members in the base class might introduce breaking changes to code in other assemblies. Why would you want to do this? An example will describe a situation in which you might find this technique useful.

Suppose you're designing a component library in which you plan to use inheritance. You've already created an assembly with a base class named `Human` and a few other classes that derive from `Human`, such as `Manager` and

`Programmer`. In this scenario, you will benefit from the features of inheritance inside the scope of your assembly. In an effort to minimize your versioning concerns, however, you want to limit the use of inheritance to your assembly alone. To be concise, you'd like to prevent classes in other assemblies from inheriting from your classes.

Of course, it's a simple matter to prevent other programmers from inheriting from the derived classes such as `Programmer` and `Manager`: You simply declare them as sealed classes using the `NotInheritable` keyword. Such a declaration makes it impossible for another programmer to inherit from these classes. However, you cannot define the `Human` class using the `NotInheritable` keyword because your design relies on it serving as a base class. Also, you cannot define a base class such as `Human` as a friend class when public classes such as `Programmer` and `Manager` must inherit from it. The CTS doesn't allow a public class to inherit from a friend class, because in general the accessibility of an entity must be equal to or greater than the accessibility of the new entity being defined (a new entity can restrict access, but cannot expand access).

To summarize the problem, you want the `Human` class to be inheritable from within the assembly and, at the same time, to be non-inheritable to classes outside the assembly. There is a popular design technique that experienced software designers often use to solve this problem—they create a public base class with constructors that are only accessible from within the current assembly. You accomplish this by declaring the base class constructors with the `Friend` keyword:

```
'*** only inheritable from within the current assembly!
Public Class Human

  Friend Sub New()
    '*** implementation
  End Sub

End Class
```

Now the definition of the `Human` class is inheritable from within its assembly but not inheritable to classes in other assemblies. While this technique is not overly intuitive at first, it can prove very valuable. It allows the compiler to enforce your design decision so as to prohibit cross-assembly inheritance. Again, the reason to use this technique is to help simplify a design that

involves inheritance. Remember that cross-assembly inheritance brings up many issues that are often best avoided when they're not a requirement.

Polymorphism and Type Substitution

In the first part of this chapter, inheritance was presented as an effective way to reuse the implementation details of a base class across many derived classes. While reuse of implementations is valuable, another aspect of inheritance is equally important—its support for polymorphic programming.

Polymorphism

Every derived class inherits the programming contract that is defined by the public members of its base class. As a result, you can program against any object created from a derived class by using the same programming contract that's defined by the base class. In other words, inheritance provides the ability to program against different types of objects using a single programming contract. This *polymorphic* programming style is a powerful programming technique because it allows you to write generic, client-side code. That is, the client-side code, which is written against the base class, is also compatible with any of the derived classes, because they all share a common, inherited design. The derived classes are thus plug-compatible from the perspective of the client, making the client's code more applicable to a wider range of situations.

Poly-whatism?

Polymorphism is one of the authors' favorite computer science terms. While precise definitions vary, polymorphism can be envisioned as the notion that an operation (method) supports one or more types (classes). For example, you can write "X + Y", and this operation will work in .NET whether X and Y are both integers, reals, or strings. Hence the operator "+" is polymorphic, and the code "X + Y" represents polymorphic programming. Expressing this in OOP terms, it might be easier to think of this code as "X.plus(Y)". At the end of the day, "X.plus(Y)" is more generic because it works in multiple situations.

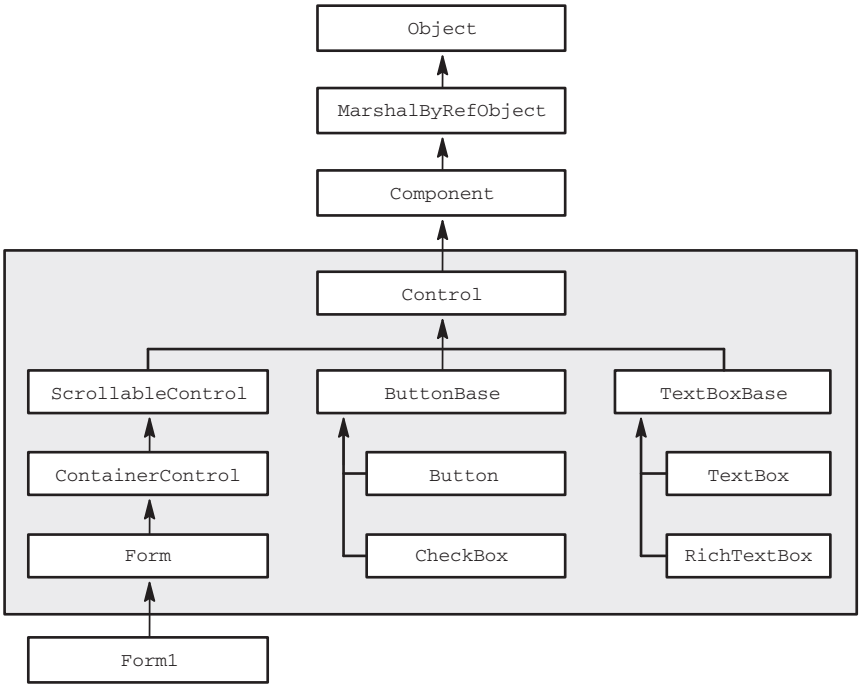

FIGURE 5.2: The inheritance hierarchy of the Windows Forms framework

Let's look at an example of writing generic client-side code based on the principle of polymorphism. This example will use classes within an inheritance hierarchy that has been designed by the Windows Forms team at Microsoft. Figure 5.2 shows some of the more commonly used classes in the Windows Forms framework that provide the implementations for forms and various controls.

The hierarchy depicted in Figure 5.2 contains a class named `Control` that serves as a base class for several other classes. Let's write a method definition called `FormatControl` against this `Control` class:

```
Imports System.Windows.Forms
Imports System.Drawing

Public Class ControlManager

  Public Sub FormatControl(ByVal ctrl As Control)
    '*** generic code using Control class
    ctrl.BackColor = Color.Blue
    ctrl.ForeColor = Color.Red
  End Sub

End Class
```

This implementation of `FormatControl` is truly generic code. You can call this method and pass a reference to many different types of objects, including `Button`, `CheckBox`, `TextBox`, and `Form`. When a method such as this one defines a parameter based on the `Control` class, you can pass an object of any type that inherits either directly or indirectly from the `Control` class in the inheritance hierarchy.

Suppose you are writing an implementation for an event handler in a form that contains a command button named `cmdExecuteTask`, a check box named `chkDisplayMessage`, and a text box named `txtMessage`. You can achieve polymorphic behavior by writing the following code:

```
Dim mgr As New ControlManager()
mgr.FormatControl(cmdExecuteTask)
mgr.FormatControl(chkDisplayMessage)
mgr.FormatControl(txtMessage)
```

Each of these calls is dispatched to the `FormatControl` method. Even though each call passes a distinct type of object, the `FormatControl` method responds appropriately because each object "is a" control. The important lesson you should draw from this example is that objects created from a derived class can always be substituted when an instance of the base class is expected.

Method overloading adds yet another dimension to polymorphic programming. For example, you could overload the `FormatControl` method of the `ControlManager` class with a more specialized definition in the case of `TextBox` objects:

```
Public Class ControlManager

  Public Sub FormatControl(ByVal txt As TextBox)
    '*** process object as a text box
  End Sub

  Public Sub FormatControl(ByVal ctrl As Control)
    '*** process other objects as a generic control
  End Sub

End Class
```

The Visual Basic .NET compiler will dispatch the call to the implementation of `FormatControl(TextBox)` when it determines that the caller is pass-

ing a `TextBox` reference. It will call `FormatControl(Control)` when it determines that the `Control` class is the most-derived class compatible with the type of reference being passed:

```
Dim mgr As New ControlManager()

'*** this call invokes FormatControl(TextBox)
mgr.FormatControl(txtMessage)

'*** these calls invoke FormatControl(Control)
mgr.FormatControl(cmdExecuteTask)
mgr.FormatControl(chkDisplayMessage)
```

As the discussion here reveals, polymorphism is very powerful because it allows you to program in more generic terms and to substitute different compatible implementations for one another. To design, write, and use class definitions in Visual Basic .NET that benefit from polymorphism, however, you must understand a number of additional concepts. The first is the difference between an object type and a reference variable type.

Converting between Types

It's important that you distinguish between the type of object you're programming against and the type of reference you're using to interact with that object. This consideration is especially critical when you are working with classes in an inheritance hierarchy, because you will commonly access derived objects using a base class reference. For example, a `TextBox` object can be referenced through a variable of type `TextBox` or a variable of type `Control`:

```
Dim textbox1 As TextBox = txtMessage
textbox1.Text = "test 1"

Dim control1 As Control = txtMessage
control1.Text = "test 2"
```

In the preceding code, the same `TextBox` object is being accessed through two types of reference variables. The subtle but critical observation here is that an object—and the reference variables used to access that object—can be based on different types. The only requirement is that the reference variable type must be compatible with the object type. A base class reference variable is always compatible with an object created from that base class, or any class

that inherits directly or indirectly from that base class. In this example, the `control1` reference variable is compatible with a `TextBox` object because the `TextBox` class inherits (indirectly) from the `Control` class.

As you see, inheriting one class from another creates an implicit compatibility between the two types. If you can reference an object using a `TextBox` reference variable, you are guaranteed that you can also reference that object using a `Control` reference variable. This is consistent with the is-a rule because a text box "is a" control. Due to this guaranteed compatibility, the Visual Basic .NET compiler allows you to implicitly convert from the `TextBox` type to the `Control` type, even with strict type checking enabled. This technique, which is sometimes referred to as *up-casting* (i.e., casting up the inheritance hierarchy), is always legal.

Trying to convert a base class reference to a derived class reference—that is, down the inheritance hierarchy—is an entirely different matter. Known as *down-casting*, this technique is not always legal. For example, if you can reference an object using a `Control` reference variable, you cannot necessarily also reference that object using a `TextBox` reference variable. While this type of reference might be permitted in some situations, it's definitely not legal in all cases. What if the object is actually a `Button` and not a `TextBox`?

When strict type checking is enabled, the Visual Basic .NET compiler will not allow you to implicitly convert down an inheritance hierarchy (e.g., from `Control` to `TextBox`). The following code demonstrates how to convert back and forth between `Control` and `TextBox` references:

```
Dim textbox1, textbox2, textbox3 As TextBox
Dim control1, control2, control3 As Control

'*** (1) compiles with or without Option Strict
textbox1 = txtMessage
control1 = textbox1

'*** (2) doesn't compile unless Option Strict is disabled
control2 = txtMessage
textbox2 = control2   '*** error: illegal implicit conversion

'*** (3) compiles with or without Option Strict
control3 = txtMessage
textbox3 = CType(control3, TextBox)
```

No problem arises if you want to implicitly convert from the `TextBox` class to the `Control` class (1). However, the automatic compatibility between these two types works only in a single direction—upward. It is therefore illegal to perform an implicit conversion from a `Control` reference to a `TextBox` reference (2). You cannot implicitly convert to a type located downward in the inheritance hierarchy. Therefore, the conversion from `Control` to `TextBox` must be made explicitly by using the `CType` conversion operator (3).

The Visual Basic .NET compiler never worries about the type of the actual object when it decides whether to allow an implicit conversion. Instead, it always relies on the static types of the reference variables being used when deciding whether implicit conversion should be permitted.

Keep in mind that an attempt to convert between types will not always be successful. For example, suppose you attempt to convert a reference variable that is pointing to a `CheckBox` object into a reference variable of type `TextBox`:

```
Dim control1 As Control = chkDisplayMessage
Dim textbox1 As TextBox = CType(control1, TextBox)
```

This code will compile without error because the `CType` operator is used to perform the conversion explicitly. It will fail at runtime, however, because a `CheckBox` object is not compatible with the programming contract defined by the `TextBox` class. In particular, the CLR will determine that the type-cast is illegal and throw an exception. A more detailed discussion of throwing and handling exceptions is deferred until Chapter 9.

If you aren't sure whether an attempt to convert a reference downward to a more-derived type will succeed, you can always perform a runtime test using the `TypeOf` operator. For example, you can test whether a `Control` reference variable points to a `TextBox`-compatible object using the following code:

```
Public Sub FormatControl(ByVal ctrl As Control)
  '*** check whether object is a TextBox
  If TypeOf ctrl Is TextBox Then  '*** safe to convert!
    Dim txt As TextBox = CType(ctrl, TextBox)
    '*** program against control as TextBox
    txt.TextAlign = HorizontalAlignment.Center
  End If
End Sub
```

It's not always necessary to write code that performs this kind of runtime test. You could instead overload the `FormatControl` method with a more specialized implementation to handle the case of a `TextBox` object. Nevertheless, sometimes you cannot predict the type of object you'll be dealing with until runtime.

Consider the situation in which you'd like to enumerate through all the controls on a form and perform a specific action only with the text boxes. The form provides a `Controls` collection that you can enumerate through using a `For Each` loop, but you cannot determine which controls are `TextBox` objects until runtime. This scenario offers a perfect example of when you should perform a runtime test using the `TypeOf` operator:

```
'*** code in a form's event handler
Dim ctrl As Control
For Each ctrl In Me.Controls
  If TypeOf ctrl Is TextBox Then
    Dim txt As TextBox = CType(ctrl, TextBox)
    '*** program against control as TextBox
    txt.TextAlign = HorizontalAlignment.Center
  End If
Next
```

You have just seen how polymorphism works with a set of classes defined in an existing inheritance hierarchy like the Windows Forms framework. Now it's time to turn your attention to how you can create an inheritance hierarchy of your own and produce the same effect. As part of this discussion, we'll continue with the example started earlier in this chapter, which included the `Human` base class and the `Programmer` derived class (see Figure 5.1).

Replacing Methods in a Derived Class

Now that you have seen the high-level motivation for polymorphism, it's time for a low-level explanation of the mechanics of how members are accessed at runtime. To use inheritance to add polymorphic behavior to your designs, you must first understand the differences between *static binding* and *dynamic binding*.

Static binding and dynamic binding represent two different ways in which a client can invoke a method or property of an object. Static binding

is usually more straightforward and results in better performance. Dynamic binding yields greater flexibility and is the underlying mechanism that allows a derived class to replace behavior inherited from a base class. The latter type of binding is one of the crucial ingredients of polymorphic programming.

Static Binding

LISTING 5.3: A simple base class with a statically bound method

```
Public Class Human
  Public Function Speak() As String
    Return "I am a human"
  End Function
End Class

Public Class Programmer : Inherits Human
  '*** inherits Speak from base class
End Class
```

First, let's tackle static binding. Suppose you are designing the `Human` and `Programmer` classes and come up with the two class definitions shown in Listing 5.3. Now imagine someone writes code that creates a `Programmer` object and invokes the `Speak` method. The caller can invoke the `Speak` method through a reference variable of type `Programmer` or through a reference variable of type `Human`:

```
Dim programmer1 As Programmer = New Programmer()
Dim human1 As Human

Dim message1, message2 As String
message1 = programmer1.Speak()  '*** calls Human.Speak()
message2 = human1.Speak()       '*** calls Human.Speak()

Console.WriteLine(message1)     '*** "I am a human"
Console.WriteLine(message2)     '*** "I am a human"
```

Both method invocations work via static binding, which is the default method invocation mechanism used by Visual Basic .NET and by the CLR. Static binding is based on the type of the reference variable, not the type of object being referenced. For example, if you are accessing a `Programmer` object through a reference variable of type `Human`, the type information

from the definition of `Human` is used to bind the client to the correct member definition.

Another important aspect of static binding is that the decision about where to locate the definition for the method being invoked is always made at compile time—that is, statically. This decision-making process is different than in dynamic binding, in which the decision about where to locate the method definition is made at runtime. With static binding, the compiler can perform more of the work at compile time. For this reason, static binding can provide a measurable performance advantage over dynamic binding.

Dynamic Binding and Overridable Methods

One of the most valuable features of inheritance is the fact that the designer of a base class can provide default behavior that can optionally be replaced by a derived class author. For example, a base class author can provide a default implementation for a method. A derived class author can then choose whether to use the base class implementation or to write a more specialized implementation. In some cases, a derived class author even has the option of extending the base class implementation by adding extra code to the derived class.

Using a derived class to replace the implementation of a method or a property defined in a base class is known as *overriding*. Method overriding relies on dynamic binding, not static binding. Dynamic binding takes into account the type of an object at runtime, which gives the CLR a chance to locate and call the overriding method. In this way dynamic binding supports more flexible code, albeit with a runtime cost. Interestingly, dynamic binding is the default in the Java programming language and, in fact, the only option in Java in most cases.

An example will illustrate why dynamic binding is so important. Imagine you have written the following client-side code that uses the programming contract defined by the `Human` class shown in Listing 5.3:

```
Public Class Reporter
  Public Sub InterviewHuman(ByVal target As Human)
    Dim Message As String = target.Speak()
    MessageBox.Show(Message)
  End Sub
End Class
```

A caller could invoke the `InterviewHuman` method by passing a `Human` object or by passing a `Programmer` object. Whichever type of object is passed to the `InterviewHuman` method must, however, provide an implementation of the `Speak` method as defined by the programming contract of the `Human` class.

An important aspect of polymorphism is that any `Human`-derived class should be able to provide a behavior for the `Speak` method that differs from the behavior of other `Human`-derived classes. For example, a `Programmer` class should be able to provide an implementation of `Speak` that is different from that in the `Human` class. You cannot take advantage of these different behaviors if static binding is being used. Because the `InterviewHuman` method programs against the `Human` type, static binding would result in every call to `Speak` being dispatched to the implementation defined in the `Human` class. Therefore, true polymorphic behavior is not possible with static binding. Your class design must contain methods that are invoked through dynamic binding.

Of course, the features of dynamic binding don't apply to all kinds of class members. When you add methods and properties to a base class, you have the option of defining them to be invoked through either static binding or dynamic binding. You don't have the same option when you add fields to a base class. In the CTS, fields can be accessed only through static binding. In other words, method and properties can be declared as *overridable* but fields cannot. This restriction gives public methods and properties yet another advantage over public fields from a design perspective.

Now that you've learned the fundamental concepts behind dynamic binding, it's time to see the Visual Basic .NET syntax that's required to support it. You enable dynamic binding by defining overridable methods and properties.

LISTING 5.4: A simple base class with a dynamically bound method

```
Public Class Human
  Public Overridable Function Speak() As String
    '*** default implementation
    Return "I am a human"
  End Function
End Class
```

The first requirement to enable overriding is that a base class must define the method or property as overridable. To do so, you declare the member definition using the `Overridable` keyword (equivalent to the `virtual` keyword in C# and C++), as shown in Listing 5.4. It's important to understand the implications of defining a method with the `Overridable` keyword. In this case, it means that every invocation of the `Speak` method through a reference variable of type `Human` will result in dynamic binding. Also, classes that inherit from `Human` will have the option of overriding the `Speak` method to provide a more specialized implementation.

Because a dynamically bound call is potentially slower than a statically bound call, it makes sense that a base class author must ask for it explicitly. Languages such as Visual Basic .NET and C# require a base class author to be very explicit about declaring methods that are overridable for another reason, too: When a method is overridable, the design becomes more challenging because the method overriding complicates the contract between a base class and its derived classes. We will revisit this topic later in this chapter. For now, just take it on faith that declaring a method or property as overridable increases your responsibilities as a base class author.

Let's finish our example by creating a derived class that overrides a method implementation defined within its base class. First, the derived class must contain a method with the same name and the same signature as the overridable method in its base class. Second, the overriding method must be explicitly declared to override the base class implementation using the `Overrides` keyword. In the following code fragment, the `Programmer` class overrides the `Speak` method inherited from the `Human` class of Listing 5.4:

```
Public Class Programmer : Inherits Human
  Public Overrides Function Speak() As String
    '*** overriding implementation
    Return "I am a programmer"
  End Function
End Class
```

Now that we've written a derived class definition that overrides a method implementation in its base class, we are ready to see an example of dynamic binding in action. Examine the following client-side code:

```
Dim programmer1 As Programmer = New Programmer()
Dim human1 As Human = programmer1
Dim message1, message2 As String

message1 = programmer1.Speak()  '*** calls Programmer.Speak
message2 = human1.Speak()       '*** calls Programmer.Speak

Console.WriteLine(message1)     '*** "I am a programmer"
Console.WriteLine(message2)     '*** "I am a programmer"
```

As this code demonstrates, it doesn't matter whether you access the `Programmer` object through a reference variable of type `Programmer` or of type `Human`. The dynamic binding scheme employed by the CLR always locates the appropriate method implementation by looking up the inheritance hierarchy for the most-derived class that holds a definition for the method in question. In the preceding code, `Programmer` is the most-derived class that contains an implementation of the `Speak` method.

Chaining Calls from a Derived Class to a Base Class

When you override a method, it's fairly common practice to chain a call from your overriding implementation in the derived class to the overridden implementation in the base class. This technique allows you to leverage the implementation provided by the base class and extend it with extra code written in the derived class. Consider the following reimplementation of the `Programmer` class:

```
Public Class Programmer : Inherits Human
  Public Overrides Function Speak() As String
    '*** chain call to Speak method in base class
    Return MyBase.Speak() & " who is a programmer"
  End Function
End Class
```

The Visual Basic .NET keyword `MyBase` is used in a derived class to explicitly access public or protected members in its base class. In this example, the `Programmer` definition of `Speak` makes an explicit call to the `Human` implementation of `Speak`. This approach allows the derived class author to reuse and extend the method implementation provided by the base class author.

As shown in the preceding example, the `MyBase` keyword allows a derived class author to chain a call to the base class author's implementa-

tion. A chained call doesn't have to be made at the beginning of the derived class implementation, however. It can be made at any point in the overriding implementation.

Design Issues with Overridable Methods

You've just seen the syntax for creating overridable methods. You've also seen the syntax for overriding a method and for chaining a call to an overridden base class implementation. As a result of the discussion, you might have concluded that the syntax required for method overriding isn't especially complicated.

In reality, mastering the syntax is the easy part. Making sure you get the semantics correct is a much tougher job. Anyone who has managed a large software project using inheritance and method overriding can tell you that managing the semantics of overridable methods and properties requires a high level of expertise and attention to detail.

An overridable method complicates the programming contract of a base class because a derived class author can use any of three possible approaches:

- A derived class author can inherit a base class implementation and reuse it without modification.
- A derived class author can provide an overriding implementation that chains a call back to the base class implementation.
- A derived class author can provide an overriding implementation that does not chain a call back to the base class implementation.

Consider these three approaches for dealing with an overridable method from a design perspective. You might say that there are really three options: reusing, extending, and replacing. When a derived class inherits a method, it *reuses* the base class implementation. When a derived class overrides a method and chains a call back the base class, it *extends* the base class implementation. When a derived class overrides a method and does not chain a call back the base class, it *replaces* the base class implementation.

While the CLR's support for method overriding allows for reusing, extending, and replacing, many overridable methods have semantics that do not support all three approaches. The overridable `Finalize` method of the `Object` class, for instance, is a real-world example of an overridable

method that does not allow replacing. If you elect to override the `Finalize` method in a user-defined class, your implementation must chain a call back to the `Finalize` implementation of the base class. If you fail to chain a call to the base class implementation, you have broken the semantic contract of this overridable method and your code will likely exhibit problematic behavior. Chapter 10 discusses when and how to override the `Finalize` method; for now, just recognize that replacing the implementation for an overridable method creates problems.

As you can see, some overridable methods only support reusing or extending the base class implementation. An overridable method may also have semantics that allow for reusing and replacing yet disallow extending. In general, the semantics of overridable methods and properties require extra attention.

The semantics involved with chaining can become even more complicated because the semantics of some overridable methods require an overriding implementation to chain a call back to the base class implementation at a specific point in time. For example, the semantics of one overridable method might require overriding method implementations to chain a call to the base class implementation before doing any work in the derived class. The semantics of another overridable method might require overriding method implementations to chain a call to the base class implementation after all work has been completed in the derived class implementation.

This discussion should lead you to two important observations:

- The semantics of method and property overriding are often sensitive to whether an overriding method should chain a call to its base class.
- The semantics of overriding can be affected by whether the chained call should be made at the beginning or at the end of the overriding method or property implementation.

If you must ever design a base class, it is your responsibility to document the semantics for each overridable method and property. Your documentation should specify for each overridable method and property whether chaining a call back to your base class implementation is required. You should also point out any occasion where a chained call must be made at the beginning or at the end of the overriding implementation in the derived class.

Even if you never design or write a base class definition, you must keep these rules in mind. As a .NET programmer, you will almost certainly encounter situations in which you must create classes that inherit from one of the base classes provided by the .NET Framework.

When you are working with inheritance, semantic errors can be much more challenging to find than syntax errors. The compiler will catch syntax errors and identify their exact locations in your code, but it cannot catch semantic errors. This factor makes semantic errors related to inheritance far more difficult to locate. Making sure the semantics for overridable methods are well defined and adhered to requires a lot of discipline. It may also require coordination across different development teams.

Declaring a Method as `NotOverridable`

Recall that a class created with Visual Basic .NET is inheritable by default. If you create a class named `Programmer` that inherits from `Human`, another programmer can create a third class, `SeniorProgrammer`, that inherits from your derived class:

```
Public Class SeniorProgrammer : Inherits Programmer
  '*** can this class override Speak?
End Class
```

Given the class definitions for `Human`, `Programmer`, and `SeniorProgrammer` (which now form the inheritance hierarchy shown in Figure 5.1), ask yourself the following question: Should the author of `SeniorProgrammer` be able to override the `Programmer` implementation of `Speak`? The answer is yes. A method that is declared with the `Overrides` keyword is itself overridable. The author of `SeniorProgrammer` can override the implementation of `Speak` in `Programmer` with the following code:

```
Public Class SeniorProgrammer : Inherits Programmer
  Public Overrides Function Speak() As String
    '*** overriding implementation
  End Function
End Class
```

You can take this example one step further by creating a class named `SuperSeniorProgrammer` that inherits from `SeniorProgrammer`. `Super-`

SeniorProgrammer would be able to override the SeniorProgrammer definition of the Speak method with yet another implementation.

If you take this example to the logical extreme, you can create as many classes as you want in the inheritance hierarchy, with each class inheriting from the one above it and overriding the Speak method with a new implementation. There isn't really a theoretical limitation on how many levels you can design in an inheritance hierarchy. In reality, practical limitations often determine how many levels of inheritance you should allow. A few examples will demonstrate how you can limit the use of inheritance to keep a complicated design from getting out of hand.

Suppose you've created a definition for Programmer by inheriting from Human. From your perspective, you are the beneficiary of inheritance because you were able to reuse code from Human and you saved yourself a good deal of time in doing so. However, if you allow other programmers to inherit from your derived class, you must also live up to the responsibilities of a base class author. That includes documenting the semantics for all of your overridable methods.

When you override a method using the Overrides keyword, your method definition becomes overridable by default. You can reverse this default behavior by adding the NotOverridable keyword before the Overrides keyword. This technique is used here to prevent the continued overriding of the Speak method:

```
Public Class Programmer : Inherits Human
  Public NotOverridable Overrides Function Speak() As String
    '*** overriding implementation
  End Function
End Class

Class SeniorProgrammer : Inherits Programmer
  '*** this class cannot override Speak
End Class
```

The author of SeniorProgrammer is no longer allowed to override the Speak method. As this example illustrates, when you declare an overriding method implementation with the NotOverridable keyword, that choice simplifies your design. You don't have to worry about other classes inheriting from your class and breaking the semantics of your method.

Using the `NotOverridable` keyword allows you to disallow overriding on a method-by-method or a property-by-property basis, but another option can make life even easier. Recall that you can disallow inheriting altogether by using the `NotInheritable` keyword. This keyword is applicable to base classes as well as derived classes such as `Programmer`:

```
Public NotInheritable Class Programmer : Inherits Human
  Public Overrides Function Speak() As String
    '*** overriding implementation
  End Function
End Class
```

Now classes may no longer inherit from `Programmer`. This choice really simplifies things because you don't have to worry about a contract of behavior between `Programmer` and derived classes. Sometimes it makes sense to define overridden methods and properties as `NotOverridable`; at other times it's better to define a derived class as `NotInheritable`. Both techniques simplify the overall design of a derived class.

Most software developers agree that keeping a design as simple as possible is beneficial. But there's another good reason to apply the `NotOverridable` and `NotInheritable` keywords whenever you can: They can also improve performance.

Recall that overridable methods require the use of dynamic binding and, therefore, may incur a runtime cost. Judicious use of the `NotOverridable` and `NotInheritable` keywords allows the Visual Basic .NET compiler to employ static binding rather than dynamic binding at times, thereby reducing execution time.

For example, imagine `Programmer` is defined with the `NotInheritable` keyword. The Visual Basic .NET compiler can make the assumption that a reference variable of type `Programmer` will only reference an object created from the `Programmer` class. That is, the client will never use a `Programmer` reference variable to access an object of some class derived from `Programmer`. Because `Programmer` is sealed, a `Programmer` reference variable can only be used to access objects created from `Programmer`. There is no opportunity for polymorphism and, consequently, no need to use dynamic binding. In such a case, the compiler will optimize calls by using static binding instead of dynamic binding.

MyBase versus MyClass versus Me

While we're on the topic of static binding versus dynamic binding, it makes sense to discuss some subtle differences between the keywords `Me`, `MyClass`, and `MyBase`. All three can be used inside a method implementation of a class to refer to a class member, but they can exhibit quite different behavior.

Listing 5.5 summarizes the class definitions we have discussed so far: `Human`, `Programmer`, and `SeniorProgrammer`. Study the listing, and determine which methods are invoked using static binding and which are invoked using dynamic binding.

LISTING 5.5: An inheritance hierarchy with statically and dynamically bound methods

```
Public Class Human
  Public Overridable Function Speak() As String
    Return "I am a human"
  End Function
End Class

Public Class Programmer : Inherits Human
  Public Overrides Function Speak() As String
    Return "I am a programmer"
  End Function

  Public Sub GetInfo()
    '*** what happens when you call Speak from this method?
  End Sub
End Class

Public Class SeniorProgrammer : Inherits Programmer
  Public Overrides Function Speak() As String
    Return "I am a senior programmer"
  End Function
End Class
```

Listing 5.5 includes three different definitions of the `Speak` method. The `Programmer` class overrides the definition of `Speak` from its base class, then is itself overridden again by the derived class `SeniorProgrammer`. Notice that the `Programmer` class now contains an additional method named `GetInfo`. Imagine you wrote the following definition for this method:

```
'*** method definition in Programmer class
Public Sub GetInfo()
  Dim message1, message2, message3, message4 As String
  message1 = MyBase.Speak()
  message2 = MyClass.Speak()
  message3 = Me.Speak()
  message4 = Speak()

  Console.WriteLine(message1)   '*** ?
  Console.WriteLine(message2)   '*** ?
  Console.WriteLine(message3)   '*** ?
  Console.WriteLine(message4)   '*** ?
End Sub
```

As you can see, there are four different ways to call the `Speak` method. The question is, What does the method output? The answer: The output depends on the type of object. First, suppose you call `GetInfo` using a reference variable of type `Human`:

```
Dim h1 As Human = New Human
h1.GetInfo()
```

This code fails to compile because the `Human` class does not contain a method called `GetInfo`—just making sure you were awake! Next, suppose you call `GetInfo` using a reference variable of type `Programmer` that refers to a `Programmer` object:

```
Dim p1 As Programmer = New Programmer
p1.GetInfo()
```

The method call outputs the following to the console window:

```
I am a human
I am a programmer
I am a programmer
I am a programmer
```

That should make sense, because the base class of a `Programmer` is `Human`. Finally, suppose you call `GetInfo` using a reference variable of type `Programmer` that refers to a `SeniorProgrammer` object:

```
Dim p2 As Programmer = New SeniorProgrammer()
p2.GetInfo()
```

Here is the resulting output:

```
I am a human
I am a programmer
I am a senior programmer
I am a senior programmer
```

The explanation of this result is a little more subtle. While the object is of type `SeniorProgrammer`, the method being called is defined inside the `Programmer` class. Therefore, this example illustrates a case where the `Programmer` class has other classes both above it and below it in the inheritance hierarchy that can affect what happens at runtime.

What happens when this call to `p2.GetInfo` executes?

- When `GetInfo` calls `MyBase.Speak`, the Visual Basic .NET compiler uses static binding to invoke the implementation of `Speak` within the base class—in this case the `Human` class, because `Programmer` inherits from `Human`.
- When it calls `MyClass.Speak`, the compiler use static binding to invoke the implementation of `Speak` in the calling method's class—in this case `Programmer` because `GetInfo` is defined within `Programmer`.
- When it calls `Me.Speak`, the compiler uses dynamic binding to invoke the most-derived implementation of `Speak`—in this case it is defined in `SeniorProgrammer`.

If you call `Speak` without using one of these three keywords, it has the exact same effect as calling `Me.Speak`—namely, it uses dynamic binding.

Calls through the `MyBase` and `MyClass` keywords always result in static binding to the base class and the current class, respectively. Calls through the `Me` keyword result in dynamic binding whenever the method being called is declared as overridable. Each of these keywords can be useful in certain scenarios.

Shadowing Methods

While most uses of static binding are relatively straightforward, this is not always the case. In certain situations, static binding can become complex and non-intuitive. In particular, it can be tricky when a base class and a derived class have one or more member definitions with the same name. An example will demonstrate this point.

Suppose we return to Listing 5.3, where the `Human` class defines `Speak` as a statically bound method. What would happen if the `Programmer` class also supplied a method called `Speak`? In other words:

```
Public Class Human
  Public Function Speak() As String
    Return "I am a human"
  End Function
End Class

Public Class Programmer : Inherits Human
  Public Function Speak() As String
    Return "I am a programmer"
  End Function
End Class
```

Both class definitions contain a method named `Speak` with the same calling signature. When a member in a derived class is defined in this manner with the same name as a non-overridable member in its base class, the technique is called *member shadowing*. That is, the `Programmer` class definition of `Speak` shadows the `Human` class definition of `Speak`.

LISTING 5.6: A derived class that shadows a method of its base class

```
Public Class Human
  Public Function Speak() As String
    Return "I am a human"
  End Function
End Class

Public Class Programmer : Inherits Human
  Public Shadows Function Speak() As String
    Return "I am a programmer"
  End Function
End Class
```

The Visual Basic .NET compiler produces a compile-time warning when you shadow an inherited member. This warning is meant to raise a red flag so you can avoid shadowing if you have stumbled upon it accidentally. If you want to deliberately shadow a member from a base class, you can suppress the compiler warning by making your intentions explicit with the `Shadows` keyword, as shown in Listing 5.6.

In a few rare situations, an experienced class designer may decide to use shadowing. The most common scenario where shadowing occurs is when the base class author adds new members to a later version. Imagine that you created the `Programmer` class by inheriting from an earlier version of the `Human` class that did not contain a `Speak` method. Therefore, at the time when you added the public `Speak` method to the `Programmer` class, it did not conflict with any of the methods inherited from its base class.

What would happen if the author of the `Human` class decided to add a public `Speak` method in a later version of the class? You would then face the dilemma of either removing the `Speak` method from the `Programmer` class or shadowing the `Speak` method from the `Human` class. A few other scenarios call for shadowing, but this one is probably the most common.

You should do your best to avoid shadowing members from a base class, because member shadowing creates ambiguities that make it easy for a client-side programmer to get into trouble. The problem with member shadowing is that it is based on static binding and, consequently, produces inconsistent results.

The following example will demonstrate where shadowing a member in a base class can create a good deal of confusion. Imagine you're writing client-side code in which you will create an object of type `Programmer`. Assume the `Programmer` class is defined as shown in Listing 5.6, where `Programmer` contains a `Speak` method that shadows the `Speak` method in the `Human` class.

To understand what's going on, you must remember how static binding works: The reference variable's type controls method invocation. Now look at the following code:

```
Dim programmer1 As Programmer = New Programmer
Dim human1 As Human

Dim message1, message2 As String
message1 = programmer1.Speak()  '*** calls Programmer.Speak()
message2 = human1.Speak()       '*** calls Human.Speak()
```

The reference variable named `programmer1` is of type `Programmer`. Therefore, invoking the `Speak` method through `programmer1` will result in invoking the implementation defined in the `Programmer` class. Likewise, the reference variable named `human1` is of type `Human`. Therefore, invoking

the `Speak` method through `human1` will result in invoking the implementation defined in the `Human` class. The strange thing about this example is that a single object responds in different ways to a call of `Speak` depending on the type of reference that is used to access the object. Dynamic binding produces much more intuitive results because the type of object—not the type of reference—determines which method is actually executed.

To make matters worse, it is legal to shadow an overridable method. However, shadowing an overridable method is something you rarely want to do. This possibility is mentioned here only as a warning that sloppy syntax can result in shadowing by mistake. This kind of mistake is likely to lead to trouble. For example, what happens when a base class defines an overridable method, and a derived class author attempts to override it but forgets to use the `Overrides` keyword? The compiler produces a warning but still compiles your code as if you had used the `Shadows` keyword:

```
Public Class Human
  Public Overridable Function Speak() As String
    '*** default implementation
  End Function
End Class

Public Class Programmer : Inherits Human
  '*** author forgot to use Overrides keyword
  Public Function Speak() As String
      '*** method shadows Human.Speak
  End Function
End Class
```

Shadowing Overloaded Methods and Properties

Shadowing can become even more complicated when it involves methods and properties that have been overloaded. Recall that the name for a method or property can be overloaded with multiple implementations that differ in terms of their parameter lists. Let's look at an example in which the `Human` class contains two overloaded methods named `Speak`, and then the `Programmer` class inherits from `Human` and defines `Speak` so that it shadows one of the inherited methods:

```
Public Class Human
  Public Function Speak() As String
    Return "I am a human"
  End Function
```

```
  Public Function Speak(ByVal message As String) As String
    Return "I am a human who says " & message
  End Function
End Class

Public Class Programmer : Inherits Human
  Public Function Speak() As String
    Return "I am a programmer"
  End Function
End Class
```

In this example, the definition of the `Speak` method in the `Programmer` class will shadow the definition of the `Speak` method in the `Human` class with the matching signature. What you might not expect is that the other overloaded definition of `Speak` within `Human` is hidden as well. Thus the method with the signature `Speak(String)` is not part of the `Programmer` class definition. For this reason, the semantics of shadowing in Visual Basic .NET are sometimes referred to as *hide-by-name*.

If you try to compile these two class definitions, you will receive another compiler warning. As before, you can suppress this warning by adding the `Shadows` keyword to the definition of the `Speak` method in `Programmer`, as depicted in Listing 5.7.

LISTING 5.7: Shadowing an overloaded method

```
Public Class Human
  Public Function Speak() As String
    Return "I am a human"
  End Function

  Public Function Speak(ByVal message As String) As String
    Return "I am a human who says " & message
  End Function
End Class

Public Class Programmer : Inherits Human
  Public Shadows Function Speak() As String
    Return "I am a programmer"
  End Function

  '*** hides Speak(String) from base class
End Class
```

You might ask why the Visual Basic .NET compiler requires you to use the `Shadows` keyword to suppress the compiler warning in this situation. To understand the motivation behind this requirement, ask yourself the following question: Should the definition for the method with the signature `Speak()` in the `Programmer` class hide every definition of `Speak` in the `Human` class, or should it just shadow the one with the matching signature? In this case the `Shadows` keyword indicates that every implementation of `Speak` in the `Human` class should be hidden from clients programming against the definition of `Programmer`.

There's a subtle yet important difference between shadowing a method and hiding a method. In Listing 5.7, the method `Speak()` is shadowed, whereas the method `Speak(String)` is hidden. The shadowed method is still accessible to clients through the derived class definition, but the hidden method is not. Take a look at the following client-side code to see the difference. This code creates only one object of type `Programmer`, yet accesses this same object through two different reference variables:

```
Dim programmer1 As Programmer = New Programmer
Dim human1 As Human

Dim message1, message2, message3, message4 As String

'*** access object through derived class reference
message1 = programmer1.Speak()          '*** calls Programmer.Speak()
message2 = programmer1.Speak("Hello")   '*** error: method doesn't exist

'*** access object through base class reference
message3 = human1.Speak()               '*** calls Human.Speak()
message4 = human1.Speak("Hello")        '*** calls Human.Speak(String)
```

As this example reveals, member hiding has a strange side effect. An object created from the `Programmer` class still provides a definition for `Speak(String)`—as evidenced by the fact that `human1.Speak("Hello")` works. However, the `Speak(String)` method is accessible only to clients that are accessing the object through a reference variable of type `Human`. As this example involves static binding, a call to `Speak()` through a reference variable of type `Human` will use the method definition from `Human`. Thus hiding doesn't remove a method or property definition from an object; it simply makes a member inaccessible to clients that use reference variables based on the derived class.

You've just seen how Visual Basic .NET allows you to shadow and hide methods using hide-by-name semantics with the `Shadows` keyword. It also allows you to use the `Overloads` keyword instead of the `Shadows` keyword in situations in which you would rather achieve *hide-by-signature* semantics. With this technique, you can shadow an overloaded method from a base class without hiding other method definitions of the same name. Let's revisit Listing 5.7 and make one minor modification:

```
Public Class Human
  Public Function Speak() As String
    Return "I am a human"
  End Function

  Public Function Speak(ByVal message As String) As String
    Return "I am a human who says " & message
  End Function
End Class

Public Class Programmer : Inherits Human
  Public Overloads Function Speak() As String
    Return "I am a programmer"
  End Function

  '*** inherits Speak(String) from base class
End Class
```

The only change that has been made to this code from Listing 5.7 is that the `Overloads` keyword has replaced the `Shadows` keyword in the `Programmer` class definition of `Speak()`. This change has the effect of shadowing a method by signature as opposed to hiding it by name. The result is that class `Programmer` now makes the definition of `Speak(String)` accessible to clients:

```
Dim programmer1 As Programmer = New Programmer
Dim human1 As Human

Dim message1, message2, message3, message4 As String

'*** access object through derived class reference
message1 = programmer1.Speak()          '*** calls Programmer.Speak()
message2 = programmer1.Speak("Hello")   '*** calls Human.Speak(String)

'*** access object through base class reference
message3 = human1.Speak()               '*** calls Human.Speak()
message4 = human1.Speak("Hello")        '*** calls Human.Speak(String)
```

It's now possible to call `Speak()` and `Speak(String)` using a reference variable of type `Programmer` or a reference variable of type `Human`. One of these method signatures is shadowed, and the other is inherited directly from `Human` to `Programmer`. Calls to `Speak()` are dispatched to either the `Human` class definition or the `Programmer` class definition depending on the type of reference variable used. Calls to `Speak(String)` are always dispatched to the definition in the `Human` class.

The `Overloads` keyword should be used on some occasions that do not involve any form of hiding or shadowing. For example, you might want to add a method to a derived class that shares the same name as one or more methods in its base class but doesn't match any of their parameter lists. Suppose you wanted to create a new class that inherits from our running definition of `Human` (see Listing 5.7). What if you decided to add a third method named `Speak` that had a signature that was different from the two signatures of `Speak` inherited from `Human`? This scenario does not involve either shadowing or hiding, but you can and should use the `Overloads` keyword:

```
Class Programmer : Inherits Human
  '*** inherits Speak() from base class
  '*** inherits Speak(String) from base class

  Public Overloads Function Speak(ByVal excited As Boolean) As String
    If excited Then
      Return "Oh boy, I am an excited programmer"
    Else
         Return "I am a programmer"
    End If
  End Function
End Class
```

Now the `Programmer` class supports three overloaded versions of `Speak`. Two implementations of `Speak` are inherited from `Human`, and a third implementation is added to the `Programmer` class. Notice that you would get very different results if you do not use the `Overloads` keyword in the `Speak(Boolean)` method definition of the `Programmer` class. If you omit this keyword, the Visual Basic .NET compiler would once again default to using the `Shadows` keyword. In that case, the `Programmer` class would contain only one definition of `Speak`, not three.

Clearly, a design in which members are shadowed and/or hidden has the potential to catch programmers off guard. The shadowing of fields,

methods, and properties results in multiple definitions with the same name. Confusion may arise because different types of reference variables produce inconsistent results when accessing the same object.

While most of this discussion has dealt at length with the complexities of shadowing and hiding, you most likely will not have to deal with these topics on a regular basis. In fact, the complexities discussed over the last several pages explain why most designers try their best to avoid designs involving shadowing and hiding. You are well advised to follow suit and avoid the use of shadowing and hiding when you design and implement your own classes.

SUMMARY

As you read through this chapter, one theme probably became clear: The use of inheritance increases your responsibilities both during the design phase and during the coding phase. If you decide to create a class from which other classes will inherit, you must make several extra design decisions. If you make these decisions incorrectly, you can get yourself into hot water pretty quickly. This is especially true when you need to revise a base class after other programmers have already begun to use it.

The first question you should address is whether it makes sense to use inheritance in your designs. In other words, should you be designing and writing your own base classes to be inherited by other classes? Once you've made a decision to use inheritance in this manner, you take on all the responsibilities of a base class author. It's your job to make sure that every derived class author understands the programming contract your base class has defined for its derived classes.

While you may never create your own base class, you will more than likely create custom classes that inherit from someone else's base class. It's difficult to imagine that you could program against the classes of the FCL without being required to inherit from a system-provided base class such as the `Form` class, the `Page` class, or the `WebService` class. Because so many of the class libraries in the .NET Framework have designs that involve base classes, every .NET developer needs to understand the responsibilities of a derived class author. If you don't have a solid understanding of the issues at hand, you will find it difficult to implement a derived class correctly.