Visual Basic .NET and the .NET Platform: An Advanced Guide

ANDREW TROELSEN

Apress™

Visual Basic .NET and the .NET Platform: An Advanced Guide
Copyright ©2002 by Andrew Troelsen

ISBN (pbk): 1-893115-26-7

Printed and bound in the United States of America 12345678910

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Karen Watterson
Technical Reviewers: Pamela Fanstill, Rob Macdonald, Dan Appleman, William Oellermann
Managing Editor: Grace Wong
Copy Editors: Anne Friedman, Nicole LeClerc
Production Editors: Janet Vail with Kari Brooks and Anne Friedman
Composition and Art Services: Impressions Book and Journal Services, Inc.
Indexer: Carol Burbo
Cover Designer: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.
Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA 94710.
Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 4

Object-Oriented Programming with VB .NET

IN THE PREVIOUS CHAPTER you were introduced to a number of core constructs of the VB .NET language. Here, you spend your time digging deeper into the details of object-based development. You begin by reviewing the famed "pillars of OOP" and then examine exactly how VB .NET contends with the notions of encapsulation, inheritance, and polymorphism. This equips you with the knowledge you need in order to build custom class hierarchies using VB .NET.

During this process, you examine some new constructs such as establishing type (rather than member) level visibility, building custom properties, and designing "sealed" classes. You also gain an understanding of the use of structured exception handling to contend with runtime errors, as opposed to the outdated "On Error Goto" mechanism of VB 6.0. This chapter wraps up with an examination of the "managed heap," including how to programmatically interact with the .NET garbage collector using the members defined by System.GC.

A Catalog of VB .NET OO-Centric Keywords

VB .NET is the first dialect of the VB language that offers full support for object oriented development techniques. Although VB 6.0 supported classes, initialize and terminate events as well as interface based programming techniques, each of these constructs were expressed in far less than ideal terms. For example, interfaces and classes were not declared using a given syntactic construct, but rather were indirectly marked by virtue of being defined within a *.cls file. In VB .NET, these same atoms are expressed using the "Class" and "Interface" keywords.

VB .NET's support for OOP goes far beyond two additional keywords however. As you will see during the course of the next three chapters, VB .NET demands a firm grounding in many OO techniques. Given that VB .NET exposes so many new concepts (especially if your current background is VB 6.0), Table 4-1 provides a high level look at the core keywords you must be ready to contend with.

Table 4-1. A Catalog of VB .NET OO-Centric Keywords

VB.NET OO-CENTRIC KEYWORD	MEANING IN LIFE
Class, Interface	Unlike VB 6.0, VB .NET provides language keywords used to define class and interface types (no more *.cls files).
MustInherit	This keyword marks a class as an "abstract base class" that is used to hold common behaviors for derived types, but is not directly creatable.
Namespace	As you saw in Chapters 2 and 3, namespaces are a way to logically group related types under a shared name.
Property, Sub, Function, Event, Delegate, WithEvents, RaiseEvent	As with VB 6.0, VB .NET class types may support any number of members. For the most part, VB .NET makes use of these keywords in very similar ways. You see the minor differences as you move through the next handful of chapters.
Sub New()	Your custom types (classes and structures) may support any number of overloaded New() methods. As you will see, these members serve as a type's "constructor."
Overridable, Overrides MustOverride, NotOverridable	Unlike VB 6.0, your custom classes may define "overridable" methods that can be "overridden" in a derived class. These keywords are the backbone of VB .NET's support for classic polymorphism.
Me, MyBase, MyClass	These keywords allow you to programmatically reference the current type as well as base class functionality.
Shadows	Derived types can "shadow" (i.e., hide) members of its base class, in order to provide a custom behavior.
Overloads	VB .NET allows you to create types that support multiple methods of the same exact name (but varying parameters). The "Overloads" keyword marks such members, but as you will see, this keyword is technically optional in most cases.
Shared	Shared members are data types and/or methods that are (pardon the redundancy) shared among all objects of the same type.
Public, Private, Protected, Friend	Types (and members of types) support various levels of visibility. These keywords represent the full set visibility options.
Inherits, Implements	These keywords allow you to derive a new class from a specified base class as well as implement any number of interfaces.

Formal Definition of the VB .NET Class

If you have been "doing objects" in another programming language, you are no doubt aware of the roll of class definitions. Formally, a class is nothing more than a custom UDT (user defined type) that is composed of data (often called attributes or properties) and functions that operate on this data (often called methods in OOP-speak). The power of object-based languages is that by grouping data and functionality in a single UDT, you are able to model your software types after real-world entities.

For example, assume you are interested in modeling a generic employee. At minimum, you may want to build a class that maintains the name, current pay, and employee ID for each worker. In addition, the Employee class defines one method named GiveBonus(), which increases an individual's current pay by some amount, and another named DisplayStats(), which prints out the relevant statistics for this individual (Figure 4-1).

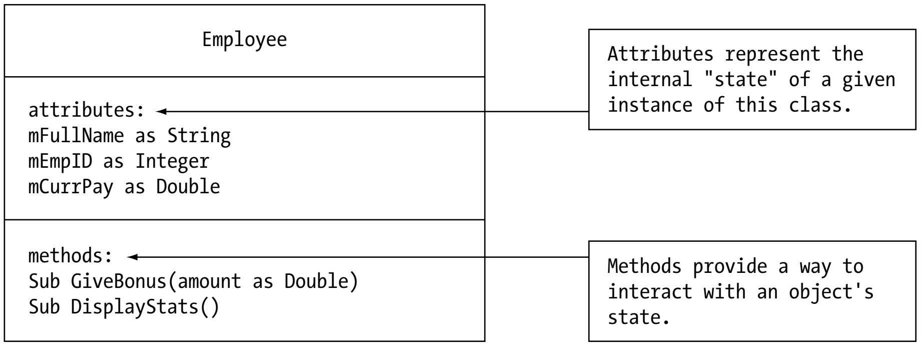

Employee
attributes:
mFullName as String
mEmpID as Integer
mCurrPay as Double
methods:
Sub GiveBonus(amount as Double)
Sub DisplayStats()
Attributes represent the internal "state" of a given instance of this class.
Methods provide a way to interact with an object's state.

Figure 4-1. A simple class definition

As you recall from Chapter 3, VB .NET classes can define any number of *constructors*. These special class methods provide a simple way for an object user to create an instance of a given class with an initial look and feel. As you know, every VB .NET class is endowed with a freebee default constructor. The role of the default constructor is to ensure that all state data is set to an initial safe value. In addition to the default constructor, you are also free to define as many custom constructors as you feel are necessary. To get the ball rolling, here is our first crack at the Employee class:

```
' The initial class definition.
class Employee
    ' Private state data.
    Private mFullName as String
    Private mEmpID as Integer
    Private mCurrPay as Double
    ' Constructors.
    Public Sub New()
    End Sub
    Public Sub New(fullName as String, empID as Integer, currPay as Double)
        mFullName = fullName
        mEmpID = empID
        mCurrPay = currPay
    End Sub
    ' Bump the pay for this employee.
    Public Sub GiveBonus(amount as Double)
        mCurrPay += amount
    End Sub
    ' Show stats of this employee.
    Public Sub DisplayStats()
        Console.WriteLine("Name: {0}", mFullName)
        Console.WriteLine("Pay: {0}", mCurrPay)
        Console.WriteLine("ID: {0}", mEmpID)
    End Sub
End Class
```

Notice the empty implementation of the default constructor:

```
Class Employee
    . . .
    ' Default constructor.
    Public Sub New()
    End Sub
End Class
```

Recall that in VB .NET, if you choose to include custom constructors in a class definition, the default constructor is *silently removed*. Therefore, if you want to allow the object user to create an instance of your class such as:

```
' Calls the default constructor.
Dim e as New Employee()
```

you need to explicitly redefine the default constructor for your class. If you forget to do so, you generate compile-time errors. Triggering the logic behind a constructor is self-explanatory. Recall that unlike VB 6.0, objects are created at the *exact point* in which the New keyword is used. Therefore, the following Employee declarations behave identically under VB .NET.

```
' Call some custom ctors (two identical approaches)
Sub Main()
    Dim e As Employee = New Employee("Joe", 80, 30000)
    e.GiveBonus(200)
    e.DisplayStats()
    Dim e2 As Employee
    e2 = New Employee("Beth", 81, 50000)
    e2.GiveBonus(1000)
    e2.DisplayStats()
End Sub
```

SOURCE CODE *The Employees project that you examine during the course of this chapter is included under the Chapter 4 subdirectory.*

Self-Reference in VB .NET

One common way to name member variables of a class is to attach the letter "m" as a prefix to the data point in question (i.e., mFullName, mEmpID and mCurrPay). Assume for a moment that the private data points of the Employee class are instead named fullName, empID and currPay. In the implementation of your custom constructor, you would suddenly have a name clash, given that the incoming parameter names are exactly the same! Like VB 6.0, VB .NET supports the "Me" keyword, which can be used within a class definition to refer to the members and data points of the defining type. Thus, you could avoid the name clash like so:

```
Class Employee
    ' Private state data.
    Private fullName As String
    Private empID As Integer
    Private currPay As Double
. . .
    Public Sub New(ByVal fullName As String, ByVal empID As Integer, _
    ByVal currPay As Double)
        Me.fullName = fullName
        Me.empID = empID
        Me.currPay = currPay
    End Sub
    . . .
End Class
```

This particular VB .NET keyword is used whenever you want to make reference to the current object instance. C#, Java, and C++ developers can equate the VB .NET "Me" keyword with the "this" keyword, which is used for the same purpose.

In this example, you made use of "Me" in your custom constructor to avoid clashes between the parameter names and names of your internal state variables. Of course, another approach would be to change the names for each parameter (or member variable) and avoid the name clash altogether (but I am sure you get the point). Also, be aware that *shared* type members cannot access the "Me" keyword. This should make perfect sense, given that shared member functions operate on the class (not object) level.

Forwarding Constructor Calls Using "Me"

Another usage of the VB .NET "Me" keyword is to force one constructor to call another. Consider the following example:

```
Class Employee
    Public Sub New(ByVal fullName As String, _
    ByVal empID As Integer, ByVal currPay As Double)
        Me.fullName = fullName
        Me.empID = empID
        Me.currPay = currPay
    End Sub
    ' If the user calls this ctor, forward to the 3-arg version
    ' using arbitrary values. . .
    Public Sub New(ByVal fullName As String)
        Me.New(fullName, IDGenerator.GetNewEmpID(), 3333)
    End Sub
. . .
End Class
```

First, notice that this iteration of the Employee class defines two custom constructors, the second of which requires a single parameter (the individual's name). However, to fully construct a new Employee, you want to ensure you have a proper Employee ID and rate of pay. Assume you have a custom class (IDGenerator) that defines a shared method named GetNewEmpID() for this very purpose. Once you gather the correct set of start-up parameters, you forward the creation request to the three-argument constructor. If you did not forward the call, you would need to add redundant initialization code to each constructor.

Member Overloading

During the last few chapters you have examined the details of VB .NET class constructors. As you have seen, it is quite common for a single class type to support any number of constructors, each of which differs by the number and type of parameters. Technically speaking, when a class defines a member of the same exact name (such as Sub New()) that differs only by the parameter set, you have "overloaded" the member.

As you have already seen, overloaded constructors can be quite helpful when you want to provide a set of construction routines that can each be accessed using the New keyword. When you overload constructors for your class types, you do not have to mark each Sub New() with an additional keyword. As long as each constructor maintains a distinct parameter list, vbc.exe is able to resolve the correct version to call.

Constructors are not the only members that can be overloaded however. In reality, any VB .NET subroutine or function may be overloaded in the same manner. Again, the key is to ensure that each version of the method has a distinct set of arguments (members differing only by return type are *not* unique enough). VB .NET defines the Overloads keyword that can be used when you want to explicitly mark a member as overloaded. This is optional. vbc.exe assumes you are overloading if it finds identically named methods with varying arguments. Assume you have added the following overloaded member to the Employee class:

```
' An overloaded method (Overloads keyword is optional).
Public Overloads Function GetStartDate(ByVal id As Integer) as Integer
    ' Look up start date using employee ID.
End Function
Public Overloads Function GetStartDate(ByVal ssn As String) as String
    ' Look up start date using SSN.
End Function
```

Here, you have a single method, GetStartDate(), which differs only by the incoming argument. This is very helpful in the eyes of the object user, given that he or she can write the following code:

```
' Calling an overloaded member.
fred.GetStartDate("111-11-2233")
jane.GetStartDate(8344)
```

rather than the more VB 6.0-centric approach of having two distinctly named members:

```
fred.GetStartDateUsingSSN("111-11-2233")
jane.GetStartDateUsingEmpID(8344)
```

Another approach taken by many VB 6.0 programmers was to have a discrete method taking a Variant data type. Using this approach, the method was in charge of determining the underlying type and value of said Variant, and acting accordingly. Of course, you are aware that the Variant data type is slow and not very type safe (at all). In fact, under VB .NET, the Variant data type is dead.

Defining the Default Public Interface

Once you have established a class' internal state data and constructor set, your next step is to flesh out the details of the *default public interface* to the class. The term refers to the set of public members that is accessible from an object instance. From an object user's point of view, the default public interface is the set of items that are accessible using the VB .NET dot operator. From the class builder's point of view, the default public interface is any item declared in a class using the Public keyword. In VB .NET, the default interface of a class may be populated by any of the following members:

- Methods: Named units of work that model some behaviors of a class.
- Properties: Accessor and mutator functions in disguise.
- Fields: Public data (although this is typically a bad idea, VB .NET supports them).

As you see in Chapter 6, the default public interface of a class may also be configured to support custom events. For the time being, let's concentrate on the use of properties, methods, and field data.

Specifying Type Level Visibility: Public and Friendly Types

Before you get too far along in your employee example, you must understand how to establish visibility levels for your custom types. In the previous chapter, you were introduced to the following class definition:

```
Class HelloClass
     ' Any number of methods with any number of parameters. . .
     ' Default and/or custom constructors. . .
End Class
```

Recall that each member defined by a class must establish its level of visibility using the Public, Private, Protected, or Friend keywords. In the same vein, VB .NET classes also need to specify their levels of visibility. The distinction is that *method visibility* is used to constrain which members can be accessed from a given object instance, and *class visibility* is used to establish which parts of the system can create the types themselves (in some ways analogous to the VB 6.0 Instancing property).

A VB .NET class can be marked by one of two visibility keywords: Public or Friend. Public classes may be created by any other objects within the same binary (assembly), as well as by other binaries (e.g., another assembly). Therefore, HelloClass could be redefined as follows:

```
' We are now creatable by types outside this assembly.
Public Class HelloClass
    ' Any number of methods with any number of parameters...
    ' Default and/or custom constructors...
End Class
```

By default, if you do not explicitly mark the visibility level of a class, it is implicitly set to "Friend." Friend classes can only be created by objects living within the same assembly, and are not accessible from outside the assembly's bounds. As you might suspect, internal items can be viewed as "helper types" used by an assembly's types to help the Public classes get their work done:

```
 ' Internal classes can only be used by other types within the same assembly.
Friend Class HelloClassHelper
     ...
End Class
```

Classes are not the only UDT that can accept a visibility attribute. As you recall, a type is simply a generic term used to refer to classes, structures, enumerations, interfaces, and delegates. Any .NET type can be assigned public or internal visibility. For example:

```
' Any type may be assigned Public or Friend visibility.
Friend Structure X            ' Cannot be used outside this assembly.
    Private myX as Integer
    Public Function GetMyX() as Integer
        Return myX
    End Function
End Structure
Friend Enum Letters   ' Cannot be used outside this assembly.
    a = 0
    b = 1
    c = 2
```

```
End Enum
Public Class HelloClass   ' May be used outside this assembly.
...
End Class
```

Logically, the previously defined types can be envisioned as shown in Figure 4-2.

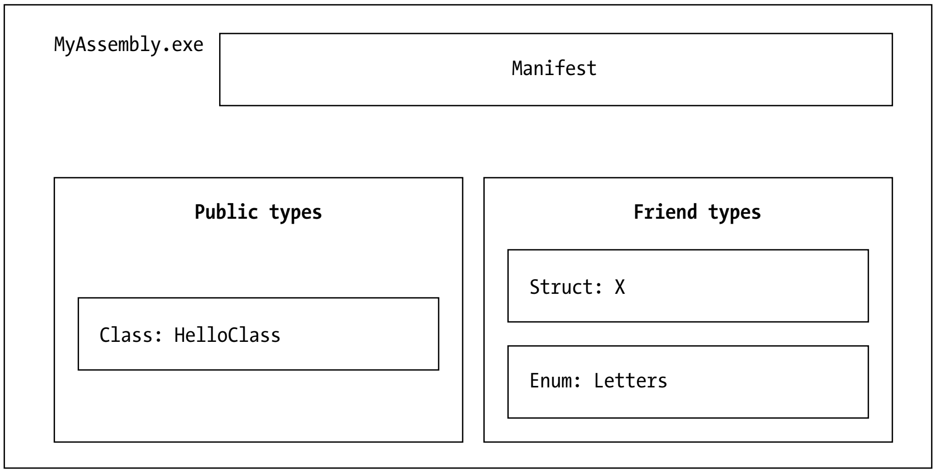


Figure 4-2. Friend and Public types

Chapter 7 drills into the specifics of composing .NET assemblies. Until then, just understand that all of your types may be defined as Public (accessible by the outside world) or Friend (not directly accessible by the outside world).

Pillars of OOP

VB .NET is a newcomer to the world of object-oriented languages (OOLs). Java, C++, Object Pascal, and (to some extent) Visual Basic 6.0 are but a small sample of the popularity of the object paradigm. Regardless of exactly when a given OOL came into existence, all object-based languages contend with three core principles of object-oriented programming, often called the famed “pillars of OOP.”

- *Encapsulation:* How well does this language hide an object’s internal implementation?
- *Inheritance:* How does this language promote code reuse?
- *Polymorphism:* How does this language let me treat related objects in a similar way?

As you are most likely already aware, VB 6.0 did not support each pillar of object technology. Specifically, VB 6.0 lacked classic inheritance (and therefore lacked classical polymorphism). VB .NET on the other hand supports each aspect of OOP, and is on par with any other modern OO language (such as C#, Java, C++, and Delphi). Before digging into the syntactic details of each pillar, it is important that you understand the basic role of each. Therefore, here is a brisk high-level rundown.

Encapsulation Services

The first pillar of OOP is called *encapsulation*. This trait boils down to the language's ability to hide unnecessary implementation details from the object user. For example, assume you have created a class named DBReader (database reader), which has two primary methods: Open() and Close():

```
' The database reader encapsulates the details of opening and closing a database...
Dim f as DBReader = New DBReader()
f.Open("C:\foo.mdf")
     ' Do something with data file. . .
f.Close()
```

The fictitious DBReader class has encapsulated the inner details of locating, loading, manipulating, and closing the data file. Object users love encapsulation, as this pillar of OOP keeps programming task simpler. There is no need to worry about the numerous lines of code that are working behind the scenes to carry out the work of the DBReader class. All you do is create an instance and send the appropriate messages (e.g., "open the file named foo.mdf").

Closely related to the notion of encapsulating programming logic is the idea of data hiding. As you know, an object's state data should ideally be specified as Private. In this way, the outside world must ask politely in order to change or obtain the underlying value. This is a good thing, as publicly declared data points can easily become corrupted (hopefully by accident rather than intent!)

Inheritance: The "is-a" and "has-a" Relationships

The next pillar of OOP, inheritance, boils down to the languages' ability to allow you to build new class definitions based on existing class definitions. In essence, inheritance allows you to extend the behavior of a base (or "parent") class by inheriting core functionality into a subclass (also called a "child class"). Figure 4-3 shows a simple example.

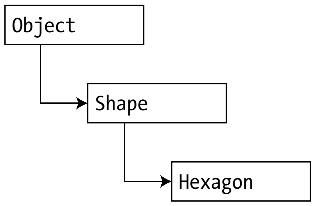


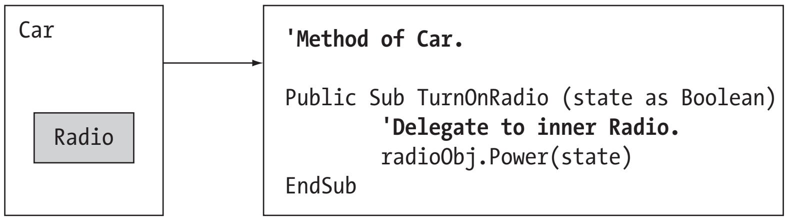
Figure 4-3. The "is-a" relationship

As you are aware, System.Object is always the topmost node in any .NET hierarchy. The Shape class extends Object. You can assume that Shape defines some number of properties, fields, methods, and events that are common to all shapes. The Hexagon class extends Shape, and inherits the core functionality defined by Shape and Object, as well as defines additional hexagon related details of its own (whatever those may be).

You can read this diagram as "A hexagon is-a shape that is-a object." When you have classes related by this form of inheritance, you establish "is-a" relationships between types. The is-a relationship is often termed *classical inheritance*.

There is another form of code reuse in the world of OOP: the containment/delegation model (also known as the "has-a" relationship). This form of reuse (used exclusively by VB 6.0 and classic COM) is not used to establish base/subclass relationships. Rather, a given class can contain another class and expose part or all of its functionality to the outside world.

For example, if you are modeling an automobile, you might want to express the idea that a car "has-a" radio. It would be illogical to attempt to derive the Car class from a Radio, or vice versa (a Car "is-a" Radio? I think not!). Rather, you have two independent classes working together, where the *outer* (or containing) class creates and exposes the *inner* (or contained) class' functionality (Figure 4-4).



```
'Method of Car.

Public Sub TurnOnRadio (state as Boolean)
        'Delegate to inner Radio.
        radioObj.Power(state)
EndSub
```

Figure 4-4. The "has-a" relationship

Here, the outer object (Car) is responsible for creating the inner (Radio) object. If the Car wants to make the Radio's behavior accessible from a Car instance, it must extend its public interface. Notice that the object user has no clue that the Car class is making use of an inner object.

```
' The inner Radio is encapsulated by the outer Car class.
Dim viper as New Car()
viper.TurnOnRadio(False)     ' Delegates request to inner Radio object.
```

Polymorphism: Classical and Ad Hoc

The final pillar of OOP is *polymorphism*. This trait captures a language's ability to treat related objects the same way. Like inheritance, polymorphism falls under two camps: Classical and ad hoc. Classical polymorphism can only take place in languages that also support classical inheritance. If this is the case (as it is in VB .NET), it becomes possible for a base class to define a set of members that can be *overridden* by a subclass. When subclasses override the behavior defined by a base class, they are essentially redefining how they respond to the same message.

To illustrate classical polymorphism, let's revisit the shapes hierarchy. Assume that the Shape class has defined a function named Draw(), taking no parameters and returning nothing. Given the fact that every shape needs to render itself in a unique manner, subclasses (such as Hexagon and Circle) are free to reinterpret this method to their own liking (Figure 4-5).

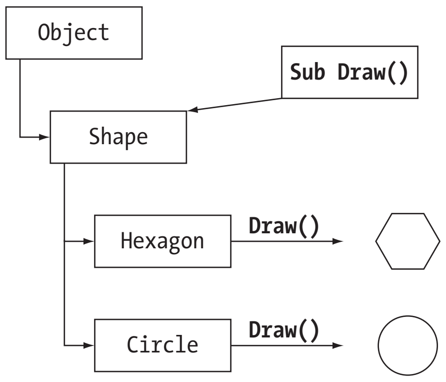


Figure 4-5. Classical polymorphism

Classical polymorphism allows a base class to enforce a given behavior on all descendents. From Figure 4-5 you can assume that any type derived from the Shapes class has the ability to be rendered. This is a great boon to any language because you are able to avoid creating redundant methods to perform a similar operation (e.g., DrawCircle(), DrawRectangle(), DrawHexagon(), and so forth).

Next, you have *ad hoc polymorphism*. This flavor of polymorphism (also used exclusively by VB 6.0) allows objects that are *not* related by classical inheritance to be treated in a similar manner, provided that every object has a method of the exact same signature (that is, method name, parameter list, and return type). Languages that support ad hoc polymorphism employ a technique called *late* *binding* to discover at runtime the underlying type of a given object. Based on this discovery, the correct method is invoked. As an illustration, first ponder Figure 4-6.

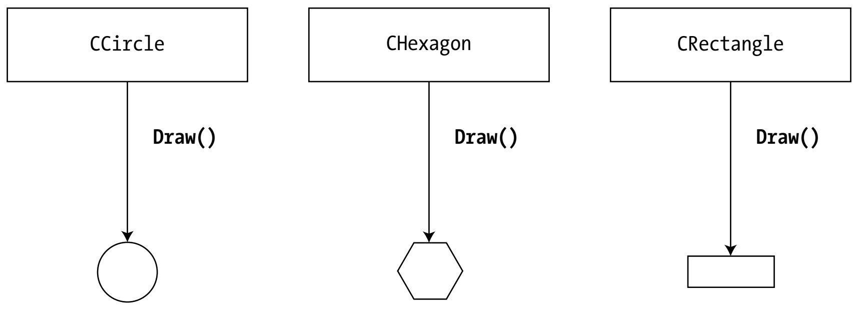


Figure 4-6. Ad hoc polymorphism

Notice how there is no common base class between the CCircle, CHexagon, and CRectangle classes. However, each class supports an identical Draw() method. Until the advent of VB .NET, Visual Basic did not support classical polymorphism (or classical inheritance for that matter), forcing developers to make due with the following ad hoc functionality. To illustrate what this boils down to syntactically, consider the following Visual Basic 6.0 code:

```
' Visual Basic 6.0 code below!
' First create an array of Object data types, setting each to an object reference.
Dim objArr(3) as Object
Set objArr(0) = New CCircle
Set objArr(1) = New CHexagon
Set objArr(2) = New CCircle
Set objArr(3) = New CRectangle

' Now loop over the array, asking each object to render itself.
Dim i as Integer
For i = 0 to 3
     objArr(i).Draw     ' Late binding. . .
Next i
```

In this code block, you begin by creating an array of generic Object data types (which is an intrinsic Visual Basic 6.0 type capable of holding any object reference, and has nothing to do with System.Object). As you iterate over the array at runtime, each shape is asked to render itself. Again, the key difference is that you have no common base class that contains a default implementation of the Draw() method. As an alternative to ad hoc polymorphism, VB 6.0 (as well as VB .NET) support interface-based polymorphism is examined a bit later in this text.

To wrap up this review of the pillars of OOP, recall that every object-oriented language needs to address how it contends with encapsulation, polymorphism, and inheritance. As you may already suspect, VB .NET completely supports each pillar of object technology, including both flavors of inheritance (is-a and has-a) as well as classical and ad hoc polymorphism. Now that you have the theory in your minds, the bulk of this chapter explores the exact VB .NET syntax that represents each trait.

The First Pillar: VB .NET's Encapsulation Services

The concept of encapsulation revolves around the notion that an object's internal data should not be directly accessible from an object instance. Rather, if an object user wants to alter the state of an object, the user does so indirectly using accessor and mutator methods. In VB .NET, encapsulation is enforced at the syntactic level using the Public, Private, Friend, and Protected keywords. To illustrate, assume you have created the following class definition:

```
' A class with a single field.
Public Class Book
     Public numberOfPages as Integer
End Class
```

When a class defines points of public data, these items are termed *fields*. The problem with field data is that the items have no ability to "understand" if the current value to which they are assigned is valid with regard to the current business rules of the system. As you know, the upper range of a VB .NET Integer is quite large (2,147,483,647). Therefore, the compiler allows the following assignment:

```
' Humm. . .
Sub Main()
     Dim miniNovel as New Book()
     miniNovel.numberOfPages = 30000000
End Sub
```

Although you have not overflowed the boundaries of an Integer data type, it should be clear that a mini-novel with a page count of 30,000,000 pages is a bit unreasonable. As you can see, fields do not provide a way to trap logical upper (or lower) limits. If your current system has a business rule that states a book must be between 1 and 2000 pages, you are at a loss to enforce this programmatically.

Because of this, public fields typically have no place in a production level class definition.

Encapsulation provides a way to preserve the integrity of state data. Rather than defining public fields (which can easily foster data corruption), you should get in the habit of defining *private data*, which are indirectly manipulated using one of two main techniques:

- Define a pair of traditional accessor and mutator methods.
- Define a named property.

Additionally, VB .NET supports special keywords "ReadOnly" and "WriteOnly," which also deliver a level of data protection. Whichever technique you choose, the point is that a well-encapsulated class should hide the details of how it operates from the prying eyes of the outside world. This is often termed "black box" programming. The beauty of this approach is that an object is free to change how a given method is implemented under the hood. It does this without breaking any existing code making use of it, provided that the signature of the method remains constant.

Enforcing Encapsulation Using Traditional Accessors and Mutators

Let's return to your existing Employee class. If you want the outside world to interact with your private string representing a worker's full name, tradition dictates defining an *accessor* (get method) and *mutator* (set method). For example:

```
' Traditional accessor and mutator for a point of private data.
Public Class Employee
     Private mFullName as String
...
    ' Accessor.
    Public Function GetFullName() As String
        Return mFullName
    End Function

    ' Mutator
    Public Sub SetFullName(ByVal n As String)
        ' Remove any illegal characters (!,@,#,$,%),
        ' check maximum length or case before making assignment.
        mFullName = n
    End Sub
End Class
```

This technique requires two uniquely named methods to operate on a single data point. The calling logic is as follows:

```
' Accessor/mutator usage.
Sub Main()
     Dim p as new Employee()
     p.SetFullName("Fred")
     Console.WriteLine("Employee is named: {0}", p.GetFullName())
     ' Error! Can't access private data from an object instance.
     ' p.mFullName
End Sub
```

Another Form of Encapsulation: Class Properties

In addition to traditional accessor and mutator methods, classes (as well as structures and interfaces) can also support *properties*. Visual Basic and COM programmers have long used properties to simulate publicly accessible points of data (that is, fields). Under the hood however, properties resolve to a pair of hidden internal methods. Rather than requiring the user to call two discrete methods to get and set the state data, the user is able to call what appears to be a single named field:

```
' Representing a person's ID as a property
Sub Main()
     Dim p as New Employee()
     ' Set the value.
     p.EmpID = 81
     ' Get the value.
     Console.WriteLine("Employee ID is: {0}", p.EmpID)
End Sub
```

Properties always map to "real" accessor and mutator methods. Therefore, as a Class designer you are able to perform any internal logic necessary before making the value assignment (e.g., uppercase the value, scrub the value for illegal characters, check the bounds of a numerical value, and so on). Here is the VB .NET syntax behind the EmpID property:

```
' Custom property for the EmpID data point.
Public Class Employee
...
    Private mEmpID as Integer
   ' Property for the empID.
   Public Property EmpID() As Integer
        Get
            Return mEmpID
        End Get
        Set(ByVal Value As Integer)
            mEmpID = Value
        End Set
   End Property
End Class
```

Unlike VB 6.0, a property is not represented by distinct Get and Set methods. A VB .NET property is composed using a Get block (accessor) and Set block (mutator). The “Value” keyword represents the right side of the assignment. As all things in VB .NET, Value is also an object. However, the underlying type of the object depends on which sort of data it represents. In your example, the EmpID property is operating on a private integer, which, as you recall, maps to an Int32:

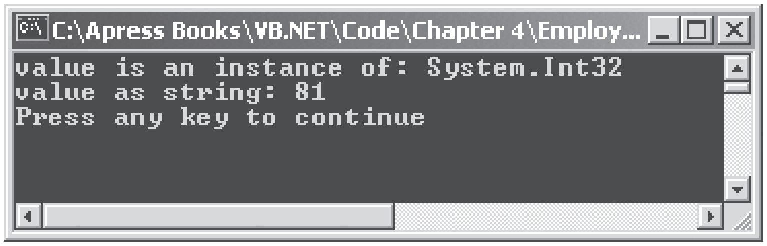
```
' Calls set, value = 81.
' 81 is an instance of Int32, so 'Value' is an Int32.
p.EmpID = 81
```

To illustrate, assume you have updated your set logic as follows:

```
Public Property EmpID() As Integer
    Get
        Return mEmpID
    End Get
    Set(ByVal Value As Integer)
    ' Just to prove the point.
        Console.WriteLine("value is an instance of: {0}", _
                Value.GetType())
        Console.WriteLine("value as string: {0}", _
                Value.ToString())
        mEmpID = Value
    End Set
End Property
```

When you set the property, you would see the following output (Figure 4-7).

Do be aware that you may only access the Value keyword within the scope of a Set block. Any attempt to do otherwise results in a compiler error.



```
C:\Apress Books\VB.NET\Code\Chapter 4\Employ...
value is an instance of: System.Int32
value as string: 81
Press any key to continue
```

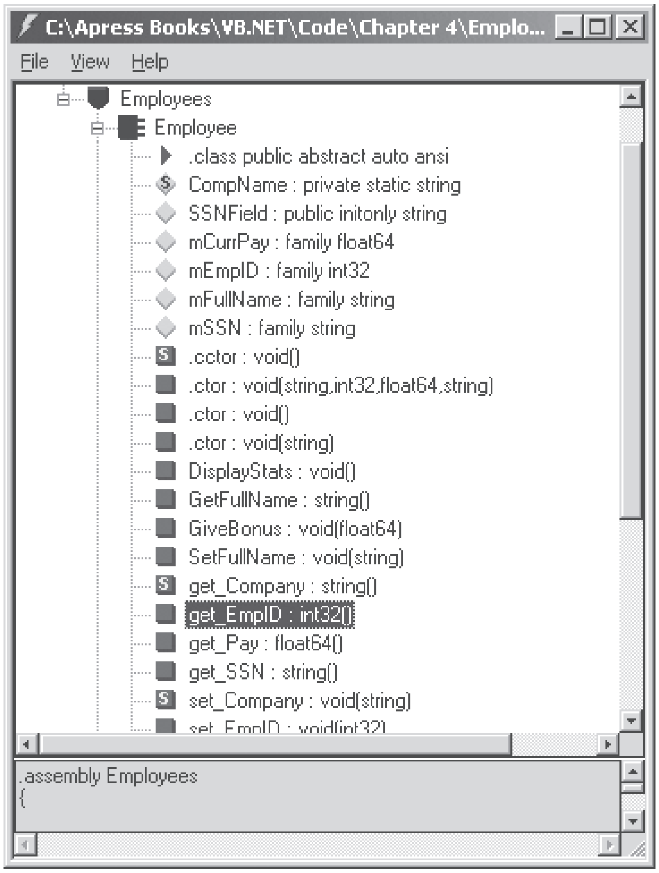
Figure 4-7. The value of "Value" when EmpID = 81

Internal Representation of VB .NET Properties

Many programmers tend to design accessor and mutator methods using "get_" and "set_" prefixes (e.g., get_Name() and set_Name()). This naming convention itself is not problematic. However, it is important to understand that under the hood, a VB .NET property is internally represented using these same prefixes. For example, if you open up the Employees.exe assembly using ILDasm.exe you see that each property actually resolves to two discrete (and hidden) methods (Figure 4-8).

Given this, realize that if you defined a class as such, you generate compiler errors:

```
' Remember, a VB .NET property really maps to a get_/set_ pair.
Public Class Employee
...
' Another property.
    Public Property SSN() As String
        Get
            Return mSSN
        End Get
        Set(ByVal Value As String)
            mSSN = Value
        End Set
    End Property
    ' ERROR! These are already defined by SSN property!
    Public Function get_SSN() As String
        Return SSN
    End Function
    Public Sub set_SSN(ByVal newVal As String)
        SSN = newVal
    End Sub
End Sub
```



```
Employees
  Employee
    .class public abstract auto ansi
    CompName : private static string
    SSNField : public initonly string
    mCurrPay : family float64
    mEmpID : family int32
    mFullName : family string
    mSSN : family string
    .cctor : void()
    .ctor : void(string,int32,float64,string)
    .ctor : void()
    .ctor : void(string)
    DisplayStats : void()
    GetFullName : string()
    GiveBonus : void(float64)
    SetFullName : void(string)
    get_Company : string()
    get_EmpID : int32()
    get_Pay : float64()
    get_SSN : string()
    set_Company : void(string)
    set_EmpID : void(int32)
```

```
.assembly Employees
{
```

Figure 4-8. Properties map to hidden get_ and set_ methods

On a related note, understand that the reverse of this situation is *not true.* Meaning, if you define two methods named get_X() and set_X() in a given class, you cannot write syntax that references a property named X:

```
' Assume Foo has two methods named get_X() and set_X() but not a
' literal VB .NET property definition.
Dim f as New Foo()
f.X = 100                 ' Error! ! Must be defined as VB .NET property, not
                          ' set_X().
Console.WriteLine(f.X)    ' Error! ! Must also be a VB .NET property, not
                          ' get_X().
```

Read-Only, Write-Only, and Shared Properties

To wrap up our investigation of VB .NET properties, there are a few loose ends to contend with. First, recall that EmpID was established as a read/write property. When building custom properties, you may want to configure a read-only property. To do so, simply build a property without a corresponding Set block, and make use of the ReadOnly keyword. Likewise, if you want to have a write-only property, omit the Get block, and make use of the WriteOnly keyword. Unlike VB 6.0, the ReadOnly and WriteOnly keywords are required in a property definition, in order to enforce readability. To illustrate, here is a read-only property for our Employee class:

```
Public Class Employee
...
    ' Assume this is assigned in the class constructor...
    Private mSSN as String
    ' A read only property.
    Public ReadOnly Property ReadOnlySSN() As String
        Get
            Return mSSN
        End Get
    End Property     ...
End Class
```

VB .NET also supports *shared properties* (which must operate on *shared data*). Recall that shared types are bound to a given class, not an instance (object) of that class. For example, assume that the Employee type defines a point of shared data to represent the name of the organization employing these workers. You may define a shared (e.g., class level) property as follows:

```
' Shared properties must operate on shared data!
Public Class Employee

    ' A shared property.
    Private Shared CompName As String
    Public Shared Property Company() As String
        Get
            Return CompName
        End Get
        Set(ByVal Value As String)
            CompName = Value
        End Set
    End Property
     ...
End Class
```

Shared properties are manipulated in the same manner as shared methods, as seen here:

```
' Set and get the name of the company that employs these people. . .
Public Sub Main()
     Employee.Company = "Intertech, Inc"
     Console.WriteLine("These folks work at {0}", Employee.Company)
     . . .
End Sub
```

Shared Constructors

As an interesting sidebar, consider the use of shared constructors. This may seem strange given that the "constructor" is understood as a method called on a new *object* variable. Nevertheless, VB .NET supports the use of shared constructors that serve no other purpose than to assign initial values to shared data. Syntactically, shared constructors are odd in that they *cannot* take a visibility modifier (but must take the Shared keyword). To illustrate, if you wanted to ensure that the name of the static CompName field was always assigned to "Intertech, Inc" on creation, you would write:

```
' Shared constructors are used to initialize shared data.
Public Class Employee
     Private Shared CompName as String
    . . ..
    Shared Sub New()
       CompName = "Intertech, Inc"
    End Sub
End Class
```

If you invoke the Employee.Company property, there is no need to assign an initial value within the Main() method, as the shared constructor does so automatically:

```
' Automatically set to "Intertech, Inc" via the shared constructor.
Public Sub Main()
. . .
     Console.WriteLine("These folks work at {0}", Employee.Company)
End Sub
```

To wrap up the examination of VB .NET properties, understand that these syntactic entities are used for the same purpose as a classical accessor/mutator pair. The benefit of properties is that the users of your objects are able to manipulate the internal data point using a single named item.

Pseudo-Encapsulation: Creating Read-Only Fields

Closely related to read-only properties is the notion of read-only *fields*. As you know, a field is a point of public data. Typically speaking, public data is a bad thing because the object user has a fairly good chance of making an illogical assignment. Read-only fields offer data preservation that is established using the ReadOnly keyword:

```
Public Class Employee
    ...
    ' Read only field.
    Public ReadOnly SSNField As String = "111-11-1111"
End Class
```

As you can guess, any attempt to make assignments to a field marked ReadOnly results in a compiler error.

Shared Read-Only Fields

Shared read-only fields are also permissible. This can be helpful if you want to create a number of constant values bound to a given class. In this light, ReadOnly seems to be a close cousin to the Const keyword. The difference is that the value assigned to Const must be resolved at compile time. The value of ReadOnly shared fields, however, may be computed at *runtime*.

For example, assume a type named Car that needs to establish a set of tires at runtime. You can create a new class (Tire) that consists of a number of shared ReadOnly fields:

```
' The Tire class has a number of readonly fields.
Public Class Tire
    Public Shared ReadOnly GoodStone As Tire = New Tire(90)
    Public Shared ReadOnly FireYear As Tire = New Tire(100)
    Public Shared ReadOnly ReadyLyne As Tire = New Tire(43)
    Public Shared ReadOnly Blimpy As Tire = New Tire(83)
    Private manufactureID As Integer
    Public ReadOnly Property MakeID() As Integer
```

```
        Get
            Return manufactureID
        End Get
    End Property
    Public Sub New(ByVal ID As Integer)
        manufactureID = ID
    End Sub
End Class

Public Class Car
    ' What sort of tires do I have?
    Public tireType As Tire = Tire.Blimpy ' Returns a new Tire.
End Class
```

Here is an example of working with these new types:

```
' Make use of a dynamically created readonly field.
Module Module1
    Sub Main()
        Dim c As Car = New Car()
        ' Prints out "Manufacture ID of tires: 83
        Console.WriteLine("Manufacture ID of tires: {0}", c.tireType.MakeID)
    End Sub
End Module
```

SOURCE CODE *The SharedReadOnly project is included under the Chapter 4 subdirectory.*

The Second Pillar: VB .NET's Inheritance Support

Now that you understand how to create a single well-encapsulated class, it is time to turn your attention to building a family of related classes. As mentioned, inheritance is the aspect of OOP that facilitates reuse of implementation. Inheritance comes in two flavors: Classical inheritance (the is-a relationship) and the containment/delegation model (the has-a relationship). Let's begin by examining the classical is-a model.

When you establish is-a relationships between classes, you are building a dependency between types. The basic idea behind classical inheritance is that new classes may leverage (and extend) the functionality of other classes. To illustrate, assume that you want to define two additional classes to the Employee project, representing sales people and managers. The hierarchy looks something like what you see in Figure 4-9.

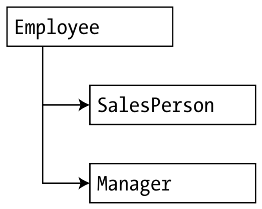


Figure 4-9. The employee hierarchy

As illustrated in Figure 4-9, you can see that a SalesPerson is-a Employee (as is a Manager—at least in a perfect world). In the classical inheritance model, base classes (such as Employee) are used to define general characteristics that are common to all descendents. Subclasses (such as SalesPerson and Manager) extend this general functionality while adding more specific behaviors to the class.

In VB .NET, extending a class is accomplished using the "Inherits" keyword. Therefore, you can syntactically model these relationships as follows:

```
' Add two new subclasses to the Employees namespace.
Public Class Manager
    Inherits Employee
     ' Managers need to know their number of stock options.
     Private numberOfOptions as Long
     Public Property NumbOpts() as Long
        Get
            Return numberOfOptions
        End Get
        Set
            numberOfOptions = Value
        End Set
     End Property
End Class

Public Class SalesPerson
Inherits Employee
    ' Sales people need to know their number of sales.
    Private numberOfSales as Long
    Public Property NumbSales() as Long
        Get
            Return numberOfSales
        End Get
        Set
            numberOfSales = Value
        End Set
    End Property
End Class
```

Notice how each subclass has extended the base class behavior by adding a custom property that operates on an underlying private point of data. Because you have established an is-a relationship, SalesPerson and Manager have *automatically* inherited all public members of the Employee base class. To illustrate:

```
' Create a subclass and access base class functionality.
Public Sub Main()
     ' Make a sales person.
     Dim stan as SalesPerson = New SalesPerson()
     ' These members are inherited from the Employee base class.
     stan.EmpID = 100
     stan.SetFullName("Stan the Man")
     ' This is defined by the SalesPerson subclass.
     stan.NumbSales = 42
End Sub
```

Needless to say, a child class *cannot* directly access private members defined by its parent class. On a related note, when the object user creates an instance of a subclass, encapsulation of private data is ensured:

```
' Error!! Instance of child class cannot allow access to a base class' private data!
Dim stan as SalesPerson = New SalesPerson()
stan.currPay
```

Controlling Base Class Creation

Currently, SalesPerson and Manager can only be created using the default class constructor. With this in mind, consider the following line of code:

```
' Create a subclass using a custom constructor.
Dim chucky as Manager = New Manager("Chucky", 92, 100000, "333-23-2322", 9000)
```

Here, you are creating an instance of the Manager class using a custom constructor. If you look at the argument list, you can clearly see that most of these values should be stored in the member variables defined by the Employee base class. Assuming you have a number of mutator methods (or class properties), you could write the following logic:

```
' If you do not say otherwise, a subclass constructor automatically calls the
' default constructor of its base class.
```

```
Public Sub New(fullName as String, empID as Integer, _
    currPay as Double, ssn as String, numbOfOpts as Long)
    ' This point of data belongs with us!
    numberOfOptions = numbOfOpts
    ' Assume the base class defines the following mutator methods.
    SetEmpID(empID)
    SetFullName(fullName)
    SetSSN(ssn)
    SetPay(currPay)
End Sub
```

Although this is technically permissible, it is not optimal. First, like most OO languages, the base class constructor (in this case the default constructor) is called automatically *before* the logic of the custom Manager constructor is executed. After this point, the current implementation accesses four public members of the employee base class to establish its state. Thus, you have really made six hits during the creation of this derived object!

To help optimize the creation of a derived class, implement your subclass constructors to explicitly call an appropriate custom base class constructor, rather than the default. In this way, you are able to call an appropriate constructor to initialize state data, and increase the efficiency of an object's creation in the process. Let's retrofit the custom constructor to do this very thing:

```
' This time, use the VB .NET 'MyBase' keyword to call
' a custom constructor on the base class.
Public Sub New(ByVal FullName As String, ByVal empID As Integer, _
    ByVal currPay As Double, ByVal ssn As String, ByVal numbOfOpts As Long)
        MyBase.New(FullName, empID, currPay, ssn)
        ' This point of data belongs with us!
        numberOfOptions = numbOfOpts
End Sub
```

Here, you make use of the VB .NET "MyBase" keyword. In this situation, you are explicitly calling the four-argument constructor defined by Employee and saving yourself unnecessary calls during the creation of the child class. The SalesPerson constructor looks almost identical:

```
' As a general rule, all subclasses should explicitly call an appropriate
' base class constructor.
Public Sub New(ByVal fName As String, ByVal empID As Integer, _
    ByVal currPay As Double, ByVal ssn As String, ByVal numbOfSales As Long)
        MyBase.new(fName, empID, currPay, ssn)
        numberOfSales = numbOfSales
End Sub
```

Also be aware that you may use the MyBase keyword any time a subclass wants to access a Public or Protected member defined by a parent class. Use of this keyword is not limited to constructor logic (you see additional examples throughout this chapter).

Regarding Multiple Base Classes

It is important to keep in mind that VB .NET demands that a given class have *exactly one* direct base class. Therefore, it is not possible to have a single type with two or more base classes (this technique is known as multiple inheritance or simply, MI). As you will see in Chapter 5, VB .NET does allow a given type to implement any number of discrete interfaces. In this way, a VB .NET class can exhibit a number of behaviors while avoiding the problems associated with classic MI. On a related note, it is permissible to configure a single *interface* to derive from multiple *interfaces* (again, details to come in Chapter 5).

Keeping Family Secrets: The "Protected" Keyword

As you already know, Public items are directly accessible from any subclass. Private items cannot be accessed from any object beyond the object that has indeed defined the Private data point. VB .NET takes the lead of many other modern day object languages and provides an additional level of accessibility: Protected.

When a base class defines protected data or protected methods, it is able to create a set of members that can be *accessed directly* by each descendent. If you want to allow the SalesPerson and Manager child classes to directly access the data sector defined by Employee, you can update the original Employee class definition as follows:

```
' Protected state data.
Public Class Employee
     ' Child classes can directly access this information. Object users cannot.
     Protected mFullName as String
     Protected mEmpID as Integer
     Protected mCurrPay as Double
     Protected mSSN as String
...
End Class
```

However, as far as the object user is concerned, protected data is private. Therefore, the following is illegal:

```
' Error! Can't access protected data from object instance
Dim emp as Employee = New Employee()
emp.ssn = "111-11-1111"
```

Preventing Inheritance: "Sealed" Classes

Classical inheritance is a wonderful thing. When you establish base class/subclass relationships, you are able to leverage the behavior of existing types. However, what if you want to define a class that cannot (for whatever reason) be subclassed? Classes of this typed are generally called “sealed” in that they prevent the chain of inheritance from continuing. For example, assume you have added yet another class to your employee namespaces, which extends the existing SalesPerson type. Consider Figure 4-10.

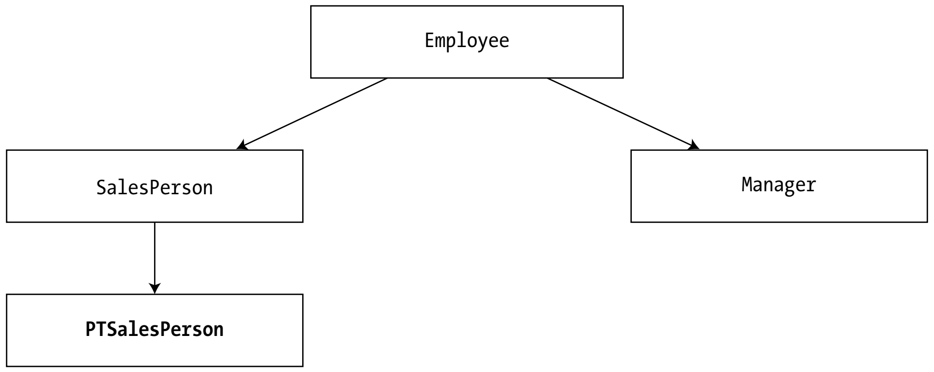


Figure 4-10. The extended employee hierarchy

PTSalesPerson is a class representing (of course) a part-time salesperson. For the sake of argument, let’s say that you want to ensure that no other developer is able to subclass from PTSalesPerson (after all, how much more part-time can you get than “part-time”?). To prevent others from extending a class, make use of the VB .NET “NotInheritable” keyword.

```
' Ensure that PTSalesPerson cannot act as a base class to others.
Public NotInheritable Class PTSalesPerson
    Inherits SalesPerson
    ' Other interesting members. . .
    Public Sub New(ByVal FullName As String, ByVal empID As Integer, _
        ByVal currPay As Double, ByVal ssn As String, ByVal numbOfSales As Long)
        MyBase.New(FullName, empID, currPay, ssn, numbOfSales)
    End Sub
End Class
```

Because PTSalesPerson is sealed, it cannot serve as a base class to any other type. For example, if you attempted to extend PTSalesPerson, you receive a compiler error.

```
' Compiler error! PTSalesPerson is sealed and cannot be extended!
Public Class ReallyPTSalesPerson
   Inherits PTSalesPerson ' Error!
End Class
```

By and large, the NotInheritable keyword is most useful when creating standalone utility classes. As an example, the String class defined in the System namespace has been explicitly sealed. Therefore, you cannot create some new class deriving from System.String. If you want to build a class that leverages the functionality of a sealed class your only option is to make use of the containment/delegation model (speaking of which. . .)

Programming for Containment/Delegation

As noted earlier in this chapter, inheritance comes in two flavors. You have just examined the classical is-a relationship. To conclude the exploration of the second pillar of OOP, let's examine the has-a relationship (also known as the containment/delegation model). Assume you have created a simple VB .NET class modeling a radio:

```
' This type will function as a contained class.
Public Class Radio
    Public Sub TurnOn(ByVal state As Boolean)
        If (state) Then
            Console.WriteLine("Jamming. . .")
        Else
            Console.WriteLine("Quiet time. . .")
        End If
    End Sub
End Class
```

Now assume you are interested in modeling an automobile. The Car class maintains a set of state data (the car's pet name, current speed, and maximum speed) all of which may be set using a custom constructor. Here is the initial definition:

```
' This class will function as the 'outer' class.
Public Class Car
    ' Internal state data
    ' (assume related public properties).
    Private currSpeed As Integer
    Private maxSpeed As Integer
    Private petName As String
. . .
    ' Is the car alive or dead?
    Private dead As Boolean
    Public Sub New()
        maxSpeed = 100
        dead = False
    End Sub
    Public Sub New(ByVal name As String, ByVal max As Integer, _
    ByVal curr As Integer)
        currSpeed = curr
        maxSpeed = max
        petName = name
    End Sub
    Public Sub SpeedUp(ByVal delta As Integer)
        ' If the car is dead, just say so. . .
        If (dead) Then
            Console.WriteLine(petName & " is out of order. . .")
        Else ' Not dead, speed up.
            currSpeed += delta
            If (currSpeed >= maxSpeed) Then
                Console.WriteLine(petName & " has overheated. . .")
                dead = True
            Else
                Console.WriteLine("CurrSpeed = " & currSpeed)
            End If
        End If
    End Sub
End Class
```

At this point you have two independent classes. Obviously, it would be rather odd to establish an is-a relationship between the two entities. However, it should be clear that some sort of relation between the two could be established. In short, you would like to express the idea that the Car 'has-a' Radio. A class that wants to contain another class is often termed the "parent" class. The contained class is termed a "child" class. To begin, you can update the Car class definition as follows:

```
' A Car has-a Radio.
Public Class Car
...
     ' The contained Radio.
     Private theMusicBox as Radio
...
End Class
```

Notice how the outer Car class has declared the Radio object as Private. This of course is a good thing, as you have preserved encapsulation. However, the next obvious question is: How can the outside world interact with child objects? It should be clear that it is the responsibility of the outer Car class to create the child Radio class. Although the outer class may create any child objects whenever it sees fit, the most common place to do so is in the constructor set:

```
' Outer classes are responsible for creating any child objects.
Public Class Car
...
     ' The contained Radio.
     Private theMusicBox as Radio
     Public Sub New()
          maxSpeed = 100
          dead = false
          ' Outer class creates the contained class(es) on start-up.
          ' NOTE:  If we did not, theMusicBox would
          ' begin life as a null reference.
          theMusicBox = New Radio()
     End Sub
     Public Sub New(name as String, max as Integer, curr as Integer)
          currSpeed = curr
          maxSpeed = max
          petName = name
          dead = false
          theMusicBox = New Radio()
     End Sub
...
End Class
```

Alternately, you could make use of the VB .NET initializer syntax as follows:

```
' A Car has-a Radio.
Public Class Car
...
```

```
     ' The contained Radio.
     Private theMusicBox as Radio = New Radio()
...
End Class
```

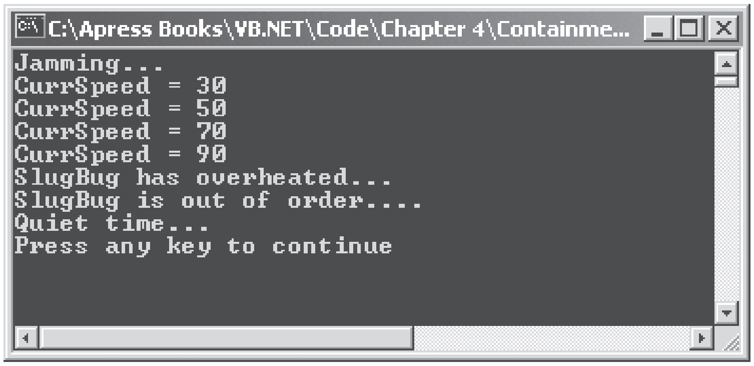
At this point, you have successfully contained another object. However, to expose the functionality of the inner class to the outside world requires *delegation*. Delegation is simply the act of adding members to the parent class that make use of the child classes' functionality. For example:

```
' Outer classes extend their public interface to provide access to inner classes.
Public Class Car
...
    Public Sub CrankTunes(ByVal state As Boolean)
        ' Tell the radio play (or not).
        theMusicBox.TurnOn(state)
    End Sub
End Class
```

In the following code, notice how the object user is able to interact with the hidden inner object indirectly, and is totally unaware of the fact that the Car class is making use of a Private Radio instance:

```
' Take this car for a test drive.
Module Module1
    Sub Main()
        ' Make a car.
        Dim c1 As Car
        c1 = New Car("SlugBug", 100, 10)
        ' Jam some tunes.
        c1.CrankTunes(True)
        ' Speed up.
        Dim i As Integer
        For i = 0 To 5
            c1.SpeedUp(20)
        Next
        ' Shut down.
        c1.CrankTunes(False)
    End Sub
End Module
```

Figure 4-11 shows the output.



```
C:\Apress Books\VB.NET\Code\Chapter 4\Containme...
Jamming...
CurrSpeed = 30
CurrSpeed = 50
CurrSpeed = 70
CurrSpeed = 90
SlugBug has overheated...
SlugBug is out of order....
Quiet time...
Press any key to continue
```

Figure 4-11. Our contained Radio in action

SOURCE CODE *The Containment project is included under the Chapter 4 subdirectory.*

Nested Type Definitions

Before examining the final pillar of OOP (polymorphism), let's explore a programming technique termed *nested classes*. In VB .NET, it is possible to define a type directly within the scope of another type. The syntax is quite straightforward:

```
' Nesting class types.
Public Class MyClass
     ' Members of outer class.
     . . .
     Public Class MyNestedClass
          ' Members of nested class.
          . . .
     End Class
End Class
```

Although the syntax is clean, understanding *why* you might do this is not readily apparent. Typically, a nested type is regarded only as a helper type of the outer class, and is not intended for use by the outside world. This is slightly along the lines of the "has-a" relationship, however in the case of nested types, you are in greater control of the inner type's visibility. In this light, nested types also help enforce encapsulation services.

To illustrate, you can redesign your current Car application by representing the Radio as a nested type. By doing so, you are assuming the outside world does not need to directly create a Radio. Here is the update:

```
' The Car is nesting the Radio. Everything else is as before.
Public Class Car
...
     ' A nested, private radio. Cannot be created by the outside world.
     Private Class Radio
          Public Sub TurnOn(ByVal state As Boolean)
              If (state) Then
                  Console.WriteLine("Jamming. . .")
              Else
                  Console.WriteLine("Quiet time. . .")
              End If
        End Sub
    End Class
     ' The outer class can make instances of nested types.
     Private theMusicBox as Radio
...
End Class
```

Notice that the Car type is able to create object instances of any nested item. Also notice that this class has been declared a *private* type. In VB .NET, nested types may be declared private as well as public. Recall, however, that classes that are directly within a namespace (e.g., nonnested types) cannot be defined as private. As far as the object user is concerned, the Car type works as before. Because of the private, nested nature of the Radio, the following is now illegal:

```
' Can't do it outside the scope of the Car class!
Dim r as Radio = New Radio()
```

SOURCE CODE *The Nested project is included under the Chapter 4 subdirectory.*

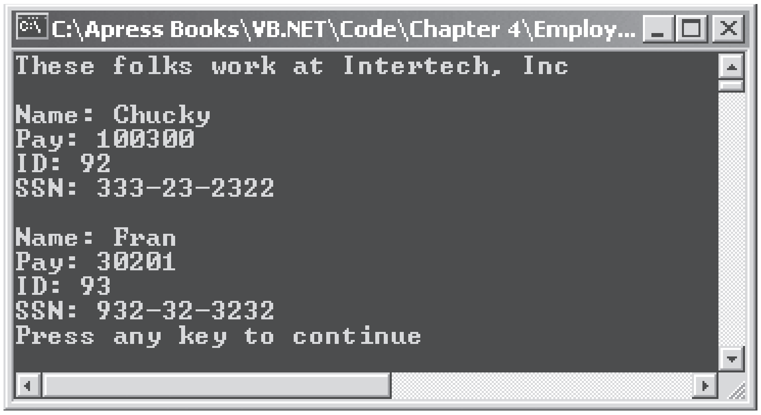
The Third Pillar: VB .NET's Polymorphic Support

Assume the Employee base class has implemented the GiveBonus() method as follows:

```
' Employee defines a new method that gives a bonus to a given employee.
Public Class Employee
...
     Public Sub GiveBonus(amount as Double)
          mCurrPay += amount
     End Sub
End Class
```

Because this method has been defined as public, you can now give bonuses to salespersons and managers (see Figure 4-12 for output):

```
' Give each child class a bonus.
Dim chucky as Manager= new Manager("Chucky", 92, 100000, "333-23-2322", 9000)
chucky.GiveBonus(300)
chucky.DisplayStats()
Dim fran as SalesPerson = new SalesPerson("Fran", 93, 30000, "932-32-3232", 31)
fran.GiveBonus(200)
fran.DisplayStats()
```



```
C:\Apress Books\VB.NET\Code\Chapter 4\Employ...
These folks work at Intertech, Inc

Name: Chucky
Pay: 100300
ID: 92
SSN: 333-23-2322

Name: Fran
Pay: 30201
ID: 93
SSN: 932-32-3232
Press any key to continue
```

Figure 4-12. The current employee hierarchy does not implement polymorphism

The problem with the current design is that the inherited GiveBonus() method operates *identically* for each subclass. Ideally, the bonus of a salesperson should take into account the number of sales. Perhaps managers should gain additional stock options in conjunction with a monetary bump in salary. Given this, you are suddenly faced with an interesting question: "How can related objects respond differently to the same request?"

Polymorphism is the final pillar of OOP, which provides a way for a subclass to redefine how it responds to a method defined by its base class. To retrofit your current design, you need to understand the use of the VB .NET "Overridable" and "Overrides" keywords. When a base class wants to define a method that may be overridden by a subclass, it must specify the method as Overridable (which is typically referred to as a "virtual method"):

```
Public Class Employee
    ' GiveBonus() has a default implementation, however
    ' child classes are free to override this behavior.
    Public Overridable Sub GiveBonus(ByVal amount As Double)
        mCurrPay += amount
    End Sub
. . .
End Class
```

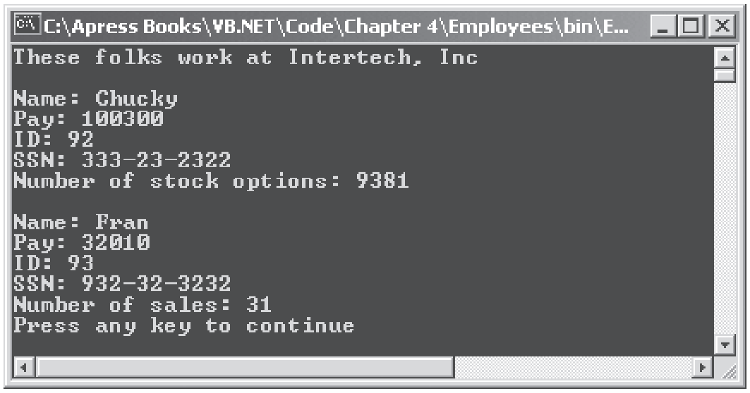
If a subclass wants to redefine a virtual method, it may change the method in question using the Overrides keyword. For example:

```
Public Class SalesPerson
Inherits Employee
     ' A sales person's bonus is influenced by the number of sales.
    Public Overrides Sub GiveBonus(ByVal amount As Double)
        Dim salesBonus As Integer
        If (numberOfSales >= 0 And numberOfSales <= 100) Then
            salesBonus = 10
        ElseIf (numberOfSales >= 101 And numberOfSales <= 200) Then
            salesBonus = 15
        Else
            salesBonus = 20  ' Anything greater than 200.
        End If
        MyBase.GiveBonus(amount * salesBonus)
    End Sub
. . .
End Class

Public Class Manager
Inherits Employee
     Private r as Random = new Random()
     ' Managers get some number of new stock options, in addition to raw cash.
    Public Overrides Sub GiveBonus(ByVal amount As Double)
        ' Increase salary.
        MyBase.GiveBonus(amount)
        ' And give some new stock options. . .
        numberOfOptions += r.Next(500)
    End Sub
. . .
End Class
```

Notice how each overridden method is free to leverage the default behavior using the MyBase keyword. In this way, you have no need to completely reimplement the logic behind GiveBonus(), but can reuse (and extend) the default behavior of the parent class.

Also assume that Employee.DisplayStats() has been declared as Overridable, and has been overridden by each subclass to account for displaying the number of sales (for sales folks) and current stock options (for managers). Now that each subclass can interpret what these virtual methods means to itself, each object instance behaves as a more independent entity (see Figure 4-13 for output):

```
' A better bonus system through polymorphism.
Dim chucky as Manager = New Manager("Chucky", 92, 100000, "333-23-2322", 9000)
chucky.GiveBonus(300)
chucky.DisplayStats()
Dim fran as SalesPerson = New SalesPerson("Fran", 93, 3000, "932-32-3232", 31)
fran.GiveBonus(200)
fran.DisplayStats()
```



```
These folks work at Intertech, Inc

Name: Chucky
Pay: 100300
ID: 92
SSN: 333-23-2322
Number of stock options: 9381

Name: Fran
Pay: 32010
ID: 93
SSN: 932-32-3232
Number of sales: 31
Press any key to continue
```

Figure 4-13. A better bonus system (thanks to polymorphism)

Excellent! At this point you are not only able to establish is-a and has-a relationships among related classes, but also have injected polymorphic activity into your employee hierarchy. As you may suspect, the story of polymorphism goes beyond simply overriding base-class behavior.

Defining (and Understanding) Abstract Classes

Currently, the Employee base class has been designed to supply protected member variables for its descendents, as well as supply two Overridable methods (GiveBonus() and DisplayStats()) that may be overridden by a given descendent. While this is all well and good, there is a rather odd byproduct of the current design: You can directly create instances of the Employee base class:

```
' What exactly does this mean?
Dim X as Employee = New Employee()
```

Now think this one through. The only real purpose of the Employee base class is to define default state data and implementations for a given subclass. In all likelihood, you did not intend anyone to create a direct instance of this class. The Employee type itself is too general a concept. A far better design is to prevent the ability to directly create a new Employee instance. In VB .NET, this is facilitated by using the "MustInherit" keyword (classes that are defined using the MustInherit keyword are termed *abstract base classes*):

```
' Update the Employee class as abstract to prevent direct instantiation.
Public MustInherit Class Employee
     ' Same public interface and state data as before. . .
End Class
```

If you do not attempt to create an instance of the Employee class, you are issued a compile time error.

```
' Error! Can't create an instance of an abstract class.
Dim X as Employee = New Employee()
```

Enforcing Polymorphic Activity: Abstract Methods

Once a class has been defined as an abstract base class, it may define any number of *abstract members*. Abstract methods can be used whenever you want to define a method that *does not* supply a default implementation. By doing so, you enforce a polymorphic trait on each descendent, leaving them to contend with the task of providing the details behind your abstract methods.

The first logical question you might have is: "Why would I ever want to do this?" To understand the role of abstract methods, let's return to the shapes hierarchy seen earlier in this chapter (Figure 4-14).

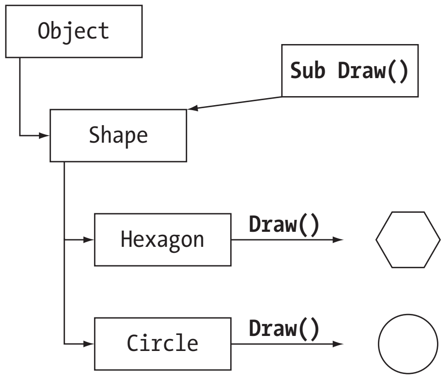


Figure 4-14. Our current shapes hierarchy

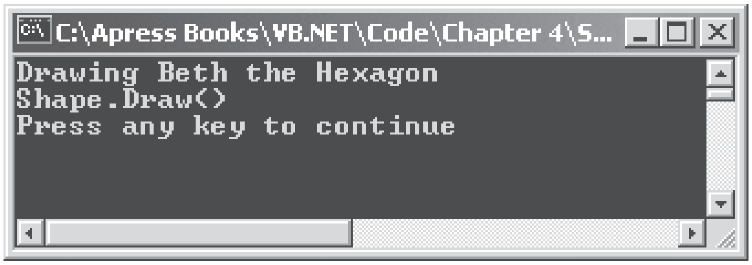
Much like the Employee hierarchy, you should be able to tell that you don't want to allow the object user to create an instance of Shape directly. To illustrate, update your initial classes as follows:

```
Public MustInherit Class Shape
    Protected mPetName As String
    ' Constructors.
    Public Sub New()
        mPetName = "NoName"
    End Sub
    Public Sub New(ByVal s As String)
        mPetName = s
    End Sub
    ' Child classes inherit this member.
    Public Overridable Sub Draw()
        Console.WriteLine("Shape.Draw()")
    End Sub
    Public Property PetName() As String
        Get
            Return mPetName
        End Get
        Set(ByVal Value As String)
            mPetName = Value
        End Set
    End Property
End Class

' Circle does NOT override Draw().
Public Class Circle
Inherits Shape
    Public Sub New()
    End Sub
    Public Sub New(name as String)
        MyBase.New(name)
    End Sub
End Class

' Hexagon DOES override Draw().
Public Class Hexagon
    Inherits Shape
    Public Sub New()
    End Sub
    Public Sub New(ByVal name As String)
```

```
        MyBase.New(name)
    End Sub
    Public Overrides Sub Draw()
        Console.WriteLine("Drawing {0} the Hexagon", PetName)
    End Sub
End Class
```

Notice that the Shape class has defined an Overridable method named Draw(). As you have just seen, subclasses are free to redefine the behavior of an Overridable method using the Overrides keyword (as in the case of the Hexagon class). The point of abstract methods becomes crystal clear when you understand that subclasses are *not required* to override virtual methods (as in the case of Circle). Therefore, if you create an instance of the Hexagon and Circle types, you find that the Hexagon understands how to draw itself correctly. The Circle, however, is more than a bit confused (see Figure 4-15 for output):

```
' The Circle object did not override the base class implementation of Draw().
Sub Main()
    ' Make and draw a hex.
    Dim Hex As Hexagon = New Hexagon("Beth")
    Hex.Draw()
    Dim cir As Circle = New Circle("Cindy")
    ' Humm. Using base class implementation.
    cir.Draw()
End Sub
```



```
Drawing Beth the Hexagon
Shape.Draw()
Press any key to continue
```

Figure 4-15. Overridable methods do not have to be overridden

Clearly this is not a very intelligent design. To enforce that each child object defines what Draw() means to itself, you can simply establish Draw() as an abstract method of the Shape class using the “MustOverride” keyword, which by definition means you provide no default implementation whatsoever:

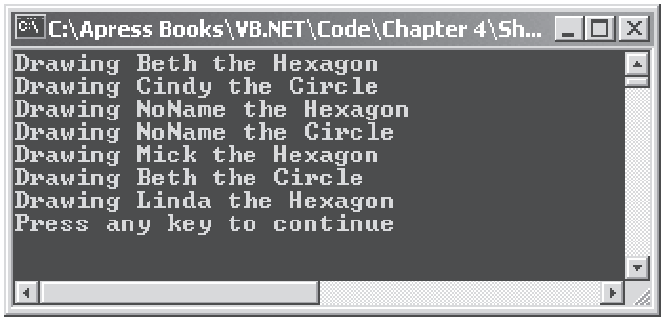
```
' Force all kids to figure out how to be rendered.
' Abstract base class.
Public MustInherit Class Shape
    Protected mPetName As String
    ' All child objects must define for themselves what
    ' it means to be drawn.
    Public MustOverride Sub Draw()  ' No 'End Sub' for abstract methods
. . .
End Class
```

Given this, you are now obligated to implement Draw() in our Circle class:

```
' If we did not implement the abstract Draw() method, Circle would also be
' considered abstract, and could not be directly created!
Public Class Circle
    Inherits Shape
    Public Sub New()
    End Sub
    Public Sub New(ByVal name As String)
        MyBase.New(name)
    End Sub
    Public Overrides Sub Draw()
        Console.WriteLine("Drawing {0} the Circle", PetName)
    End Sub
End Class
```

To illustrate the full story of polymorphism, consider the following code (Figure 4-16 shows the output):

```
' Create an array of various Shapes.
Module Module1
    Sub Main()
        ' The base class reference trick.
        Dim s as Shape() = {New Hexagon(), New Circle(), New Hexagon("Mick"), _
        New Circle("Beth"), New Hexagon("Linda")}
        Dim i As Integer
        For i = 0 To UBound(s)
            s(i).Draw()
        Next
    End Sub
End Module
```



```
Drawing Beth the Hexagon
Drawing Cindy the Circle
Drawing NoName the Hexagon
Drawing NoName the Circle
Drawing Mick the Hexagon
Drawing Beth the Circle
Drawing Linda the Hexagon
Press any key to continue
```

Figure 4-16. Better! Abstract methods must be overridden

This illustrates polymorphism at its finest. Recall that when you mark a class as MustInherit, you are unable to create a *direct instance* of that type. However, you can freely store references to any subclass within an abstract base variable. As you iterate over the array of Shape references, it is at runtime that the correct type is determined. At this point, the correct method is invoked. You may be thinking "hey! This is a lot like interface-based programming!" You are correct. However realize that abstract base classes can do far more than simply define abstract methods. They are also able to define any number of concrete methods that may be leveraged by a subclass.

Shadowing Class Members

VB .NET provides a facility that is the logical opposite of method overriding: method hiding. Assume you are in the process of building a brand new class named Oval. Given that an Oval is-a type of Circle, you may want to extend the Shapes hierarchy as shown in Figure 4-17.

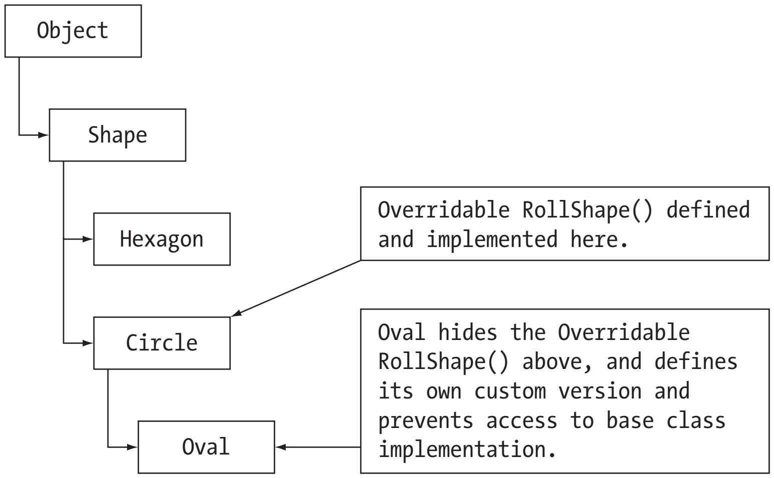

Object
Shape
Hexagon
Circle
Oval
Overridable RollShape() defined and implemented here.
Oval hides the Overridable RollShape() above, and defines its own custom version and prevents access to base class implementation.

Figure 4-17. Versioning the Draw() method

Now, for the sake of argument, assume that the Oval wants to hide the inherited version of RollShape() and prevent its code base from accessing the base class functionality? Formally, this technique is termed *versioning* a method. Syntactically, this can be accomplished using the Shadows keyword on a method-by-method basis. For example:

```
' This class extends Circle, but hides the inherited RollShape() method.
Public Class Oval
    Inherits Circle
    Public Sub New()
        MyBase.PetName = "Joe"
    End Sub
    ' Hide base class impl if they create an Oval.
    Public Shadows Sub RollShape()
        Console.WriteLine("Rolling an Oval. . .")
        Console.WriteLine("FLOP. . .")
    End Sub
End Class
```

Because you used the Shadows keyword in the definition of RollShape(), you are guaranteed that if an object user makes an instance of the Oval class and calls RollShape(), the most derived version is called. Thus:

```
' The RollShape() defined by Oval will be called.
Dim o As Oval = New Oval()
o. RollShape()
```

At this point, method hiding may seem to be little more than an interesting exercise in class design. However, this technique can be very useful when you are extending types defined within another .NET assembly. Imagine that you want to derive a new class from another class defined in a distinct .NET binary. Now, what if the binary base type defines a Draw() method that is somehow incompatible with your own Draw() method? To prevent object users from triggering a base class implementation, just shadow the member.

SOURCE CODE *The Shapes hierarchy can be found under the Chapter 4 subdirectory.*

Casting between Class Types (CType)

At this point you have created a number of class hierarchies in VB .NET. Next, you need to examine the laws of *casting* between class types. First, recall the

Employee hierarchy. The topmost member in our hierarchy is System.Object. Given the terminology of classical inheritance, everything "is-a" object. In our example, a part-time salesperson "is-a" salesperson, and so forth. Therefore, the following cast operations are legal.

```
'  A Manager 'is-a' object.
Dim o As Object = New Manager("Frank Zappa", 9, 40000, "111-11-1111", 5)
'  A Manager 'is-a' Employee too.
Dim e As Employee = New Manager("MoonUnit Zappa", 2, 20000, "101-11-1321", 1)
'  A PTSales dude(tte) is a Sales dude(tte)
Dim sp As SalesPerson = New PTSalesPerson("Jill", 834, 100000, "111-12-1119", 90)
```

As seen above, the first law of casting between class types is that when two classes are related by an is-a relationship, it is always safe to reference a derived class using a base class reference. This leads to some powerful programming constructs. For example, if you have a module level method such as:

```
' Fire everyone >:-)
Public Sub FireThisPerson(ByVal e As Employee)
    Console.WriteLine(e.GetFullName() & " has been fired!")
End Sub
```

You can effectively pass any descendent from the Employee class into this method. Thus:

```
' Streamline the staff.
FireThisPerson(sp)
FireThisPerson(e)
```

The following logic works as there is an implicit cast from the base class type (Employee) to the derived types. Now, what if you also wanted to fire your Manager (currently held in a System.Object reference)? If you pass the object reference into the FireThisPerson() method as follows:

```
' A Manager 'is-a' object.
Dim o As Object = New Manager("Frank Zappa", 9, 40000, "111-11-1111", 5)
FireThisPerson(o)     ' Error!
```

you are issued a compiler error (if you have Option Strict enabled, which of course you do)! The reason for the error is because you cannot automatically receive access from a base type (in this case System.Object) to a derived type (in this case Employee) without first performing an explicit cast.

This is the second law of casting: You must explicitly downcast using the VB .NET CType() function. CType() takes two parameters. The first parameter is the base class type you currently have access to. The second parameter is the name of the derived type you want to have access to. The value returned from CType() is the result of the downward cast. Thus, the previous problem can be avoided as follows:

```
' Error! Must explicitly cast when moving from base to derived class!
' FireThisPerson(o) ' No!
' OK.
FireThisPerson(CType(o, Manager))
```

As you will see in the next chapter, CType() is also the safe (and preferred) way of obtaining an interface reference from a type. Furthermore, CType() may operate safely on numerical types but don't forget you have a number of related conversion functions at your disposal (CInt() and so on).

Exception Handling

Error handling among Windows developers has grown into a confused mishmash of techniques over the years. Many programmers roll their own error handling logic within the context of a given application. For example, a development team may define a set of constants that represent known error conditions, and make use of them as method return values. In addition to this ad hoc technique, the Window's API defines a number of error codes that come by way of #defines, HRESULTs, and far too many variations on the simple Boolean. Furthermore, many COM developers have made use of a small set of standard COM interfaces (e.g., ISupportErrorInfo, IErrorInfo, ICreateErrorInfo) to return meaningful error information to a COM client (although VB 6.0 hides the process from view using the "On Error Goto" syntax and intrinsic Err object).

The obvious problem with the previous techniques is the tremendous lack of symmetry. Each approach is tailored to a given technology, a given language, and perhaps a given project. In order to put an end to this madness, the .NET platform provides exactly *one* technique to send and trap runtime errors: Structured Exception Handling (SEH).

The beauty of this approach is that developers now have a well-defined approach to error handling, which is common to all languages targeting the .NET universe. Therefore, the way in which a VB .NET programmer handles errors is conceptually identical to that of a C# programmer, a C++ programmer using managed extensions (MC++), and so forth. As an added bonus it is also possible to throw and catch exceptions across binaries, AppDomains (defined in Chapter 7), and machines in a language-independent manner.

To begin to understand how to program using exceptions, you must first realize that exceptions are indeed objects. All system-defined and user-defined exceptions derive from System.Exception (which in turn derive from System.Object). Here is a breakdown of some of the interesting members defined by the Exception class (Table 4-2):

Table 4-2. Core Members of the System.Exception Type

SYSTEM.EXCEPTION PROPERTY	**MEANING IN LIFE**
HelpLink	This property returns a URL to a help file describing the error in gory detail.
Message	This read-only property returns the textual description of a given error.
Source	This property returns the name of the object (or possibly the application) that sent the error.
StackTrace	This read-only property contains a string that identifies the sequence of calls that triggered the error.
InnerException	The InnerException property can be used to preserve the error details between a series of exceptions. For example, assume the object user triggers method A. During the invocation of method A, an exception is triggered (and caught). Method A can save this exception using the InnerException property and throw a new (more specific) exception ("method A bombed"). Thus, the caller is able to fully understand the flow of error logic by catching the error and investigate the "inner" exception.

Throwing an Exception

To illustrate the use of System.Exception, let's revisit the Car class defined earlier in this chapter, in particular, the SpeedUp() method. Here is the current implementation:

```
' Currently, SpeedUp() reports errors using console IO.
Public Sub SpeedUp(ByVal delta As Integer)
    ' If the car is dead, just say so. . .
    If (dead) Then
        Console.WriteLine(petName & " is out of order. . .")
    Else ' Not dead, speed up.
        currSpeed += delta
        If (currSpeed >= maxSpeed) Then
            Console.WriteLine(petName & " has overheated. . .")
            dead = True
        Else
```

```
            Console.WriteLine("CurrSpeed = " & currSpeed)
        End If
    End If
End Sub
```

To illustrate, let's retrofit SpeedUp() to throw an exception if the user attempts to speed up the automobile after it has met its maker (dead = True). First, you create and configure a new instance of the Exception class. When you want to pass the error back to the caller, make use of the VB .NET *Throw* keyword. Here is an example:

```
' This time, throw an exception if the user speeds up a trashed automobile.
Public Sub SpeedUp(delta as Integer)
     If(dead) Then
          Throw New Exception("This car is already dead")
     Else
     . . .
End Sub
```

Before examining how to handle this incoming exception, a few points. First of all, when you build custom classes, it is always up to you to decide exactly what constitutes an exception. Here, you are making the assumption that if the program attempts to increase the speed of a car that has expired, the custom Exception should be thrown to indicate the SpeedUp() method cannot continue. Alternately, you could implement SpeedUp() to recover automatically without needing to throw an exception.

By and large, exceptions should be thrown only when a more "terminal" condition has been met (can't connect to a data source, a file that was to be opened is missing, an external device is not responding, or whatnot). Deciding exactly what constitutes throwing an exception is a design issue you must always contend with. For now, assume that asking a doomed automobile to increase its speed justifies a cause for an exception.

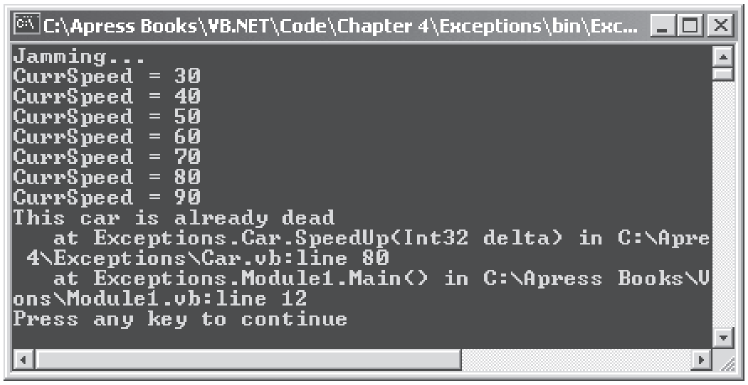
Next, understand that the .NET runtime libraries already define a number of predefined exceptions. For example, the System namespace defines numerous exceptions such as ArgumentOutOfRangeException, IndexOutOfRangeException, StackOverflowException, and so forth. Other namespaces define additional exceptions that reflect the behavior of that namespace (e.g., System.Drawing.Printing defines printing exceptions and System.IO defines IO based exceptions).

Catching Exceptions

Because the SpeedUp() method is able to throw an exception object, you need to be ready to handle the error should it occur. When you call a method that may throw an exception, you should establish a Try/Catch block to wrap the call. Here is the simplest form:

```
' Speed up the car safely...
Sub Main()
    ' Make a car.
    Dim buddha As Car = New Car("Buddha", 100, 20)
    buddha.CrankTunes(True)
    ' Try to rev the engine hard!
    Try
        Dim i As Integer
        For i = 0 To 10
            buddha.SpeedUp(10)
        Next
    Catch e As Exception
        Console.WriteLine(e.Message)
        Console.WriteLine(e.StackTrace)
    End Try
End Sub
```

In essence, a Try block is a section of code that is on the lookout for any exception that may be encountered during the flow of execution. If an exception is detected, the flow of program execution is sent to the next available Catch block. On the other hand, if the code within a Try block does not trigger an exception, the Catch block is skipped entirely, and all is right with the world. Figure 4-18 shows a test run of the handled error.



```
Jamming...
CurrSpeed = 30
CurrSpeed = 40
CurrSpeed = 50
CurrSpeed = 60
CurrSpeed = 70
CurrSpeed = 80
CurrSpeed = 90
This car is already dead
   at Exceptions.Car.SpeedUp(Int32 delta) in C:\Apre
 4\Exceptions\Car.vb:line 80
   at Exceptions.Module1.Main() in C:\Apress Books\V
ons\Module1.vb:line 12
Press any key to continue
```

Figure 4-18. Dealing with the error using structured exception handling

Notice how this Catch block explicitly specifies the exception it is willing to catch. In VB .NET (as well as numerous other languages targeting the .NET platform) it is also permissible to configure a Catch block that does not explicitly define a specific exception. Thus, we could implement the Try/Catch block as follows:

```
' A generic catch.
. . .
Catch
     Console.WriteLine("Something bad happened. . .")
. . .
```

Obviously, this is not the most descriptive manner in which to handle runtime exceptions, given that there's no way to obtain meaningful information about the error that occurred. Nevertheless, VB .NET does allow for such a construct.

Building Custom Exceptions

Although you could simply throw instances of System.Exception to signal a runtime error, it is sometimes advantageous to build a custom class that encapsulates the details of your problem. Let's examine two possible approaches.

Take One

Assume you want to build a custom exception to represent the error of speeding up a doomed automobile. To begin, create a new class derived from System.Exception (by convention, custom exceptions should end with an "–Exception" suffix). After this point, you are free to include any custom properties, methods, or fields that can be used from within the catch block of the calling logic. You are also free to override any Overridable members defined by your parent classes:

```
' This custom exception describes the details of the car-is-dead condition.
Public Class CarIsDeadException
    Inherits System.Exception
    ' This custom exception maintains the name of the doomed car.
    Private carName As String
    Public Sub New()
    End Sub
    Public Sub New(ByVal carName As String)
        Me.carName = carName
    End Sub
    ' Override the Exception.Message property.
    Public Overrides ReadOnly Property Message() As String
        Get
            Dim msg As String = MyBase.Message
            msg &= carName & " has bought the farm. . ."
            Return msg
        End Get
    End Property
End Class
```

Here, the CarIsDeadException type maintains a private data member that holds the name of the car that threw the exception. You have also added two constructors to the class, and overrode the read-only Message property in order to include the pet name of the car in the error description. Throwing this error from within SpeedUp() should be self-explanatory:

```
' Throw the custom exception.
' This time, throw an exception if the user speeds up a trashed automobile.
Public Sub SpeedUp(delta as Integer)
    If(dead) Then
         Throw New CarIsDeadException(Me.petName)
    Else
    . . .
End Sub
```

Catching the error is just as easy:

```
Try
    . . .
 Catch e As CarIsDeadException
        Console.WriteLine(e.Message)
    . . .
End Try
```

In this scenario, you may not need to build a custom exception class, given that you are free to set the Message property directly using the Exception type. Typically, you only need to create custom exceptions when the error is tightly bound to the class issuing the error (for example, a File class that throws a number of file-related errors, a Car class that throws a number of automobile-centric exceptions and so forth). Nevertheless, at this point you should understand the basic process of constructing a custom exception type.

Take Two

Our CarIsDeadException type has overridden the Message property to configure a custom error message. This class also has an overloaded constructor that accepts the pet name of the automobile that has met its maker. When you build custom exceptions, you are able to build the type as you see fit. However, the recommended approach is to build a relatively simple type that supplies three named constructors matching the following signature:

```
Public Class CarIsDeadException
    Inherits System.Exception
    ' Constructors for this exception.
    Public Sub New()
    End Sub
    Public Sub New(ByVal message As String)
        MyBase.New(message)
    End Sub
    Public Sub New(ByVal message As String, ByVal innerEx As Exception)
        MyBase.New(message, innerEx)
    End Sub
End Class
```

Notice that this time you have *not* provided a private string to hold the pet name, and have *not* overridden the Message property. Rather, you are simply passing all the relevant information to your base class. When you want to throw an exception of this type, you would send in all necessary information as a constructor argument (the output would be identical):

```
' If the car is dead, just say so. . .
if(dead)
      ' Pass pet name and message as ctor argument.
      Throw New CarIsDeadException(Me.petName & " has bought the farm!")
End If
```

Using this design, your custom exception is little more than a semantically defined name, devoid of any unnecessary member variables (or overrides).

Handling Multiple Exceptions

As mentioned, in its simplest form, a Try block has a single corresponding Catch block. In reality, you often run into a situation where the code within a Try block could trigger numerous exceptions. For example, assume the car's SpeedUp() method not only throws an exception when you attempt to speed up a doomed automobile, but throws another if you send in an invalid parameter (for example, any number less than zero):

```
Public Sub SpeedUp(ByVal delta As Integer)
    ' Bad param?
    If (delta < 0) Then
        Throw New ArgumentOutOfRangeException("Speed must be greater than zero")
    End If
    ' If the car is dead, just say so. . .
    If (dead) Then
        ' Throw 'Car is dead' exception.
        Throw New CarIsDeadException(petName & " has bought the farm!")
    Else
        currSpeed += delta
        If (currSpeed >= maxSpeed) Then
            dead = True
        Else
            Console.WriteLine("CurrSpeed = {0}", currSpeed)
        End If
    End If
End Sub
```

The calling logic would look something like this:

```
' Here, we are on the lookout for multiple exceptions.
Try
    Dim i As Integer
     For i = 0 To 10
         buddha.SpeedUp(10)
    Next
Catch e As CarIsDeadException
    Console.WriteLine(e.Message)
    Console.WriteLine(e.StackTrace)
Catch e As ArgumentOutOfRangeException
    Console.WriteLine(e.Message)
    Console.WriteLine(e.StackTrace)
End Try
```

It is also worth pointing out that a Catch block may be adorned with a “When” condition. For example, assume you want to handle the CarIsDeadException just a bit differently if the Car that throws the exception is called by a particular pet name. If any other Car type throws a CarIsDeadException, you want to take a different plan of action:

```
Public Sub SpecialErrorForBuddha()
     Dim b As New Car("Buddha", 50, 0)
     Try
         Dim i As Integer
         For i = 0 To 10
             b.SpeedUp(10)
         Next
     Catch e As CarIsDeadException When b.PetName = "Buddha"
         Console.WriteLine("Buddha died. . .")
     Catch
         Console.WriteLine("Some car died. . .")
     End Try
End Sub
```

Given that you did indeed set the pet name of our car as Buddha, the line "Buddha died. . ." will be spit out to the console. If you passed in the string "Bill" as the first argument to the Car's constructor, you would see "Some car died. . ." printed instead. When you make use of a When condition on a Catch block, this does *not* mean you can ignore the error if the condition evaluates to False. What it does mean is you can have a finer level of granularity when handling the error.

The "Finally" Block

A Try/Catch block may also be augmented with an optional "Finally" block. The idea behind a Finally block is to ensure that any acquired resources can be cleaned up, even if an exception interferes with the normal flow of execution. For example, assume you want to always power down the car's radio before exiting Main(), regardless of any errors:

```
' Provide a manner to clean up.
 Sub Main()
    ' Make a simple car.
    Dim buddha As Car = New Car("Buddha", 100, 20)
    buddha.CrankTunes(True)
    ' Try to rev the engine hard!
    Try
        Dim i As Integer
        For i = 0 To 10
            buddha.SpeedUp(10)
        Next
    Catch e As CarIsDeadException
        Console.WriteLine(e.Message)
        Console.WriteLine(e.StackTrace)
    Catch e As ArgumentOutOfRangeException
        Console.WriteLine(e.Message)
        Console.WriteLine(e.StackTrace)
    Finally
        ' This will always happen regardless.
        buddha.CrankTunes(False)
    End Try
End Sub
```

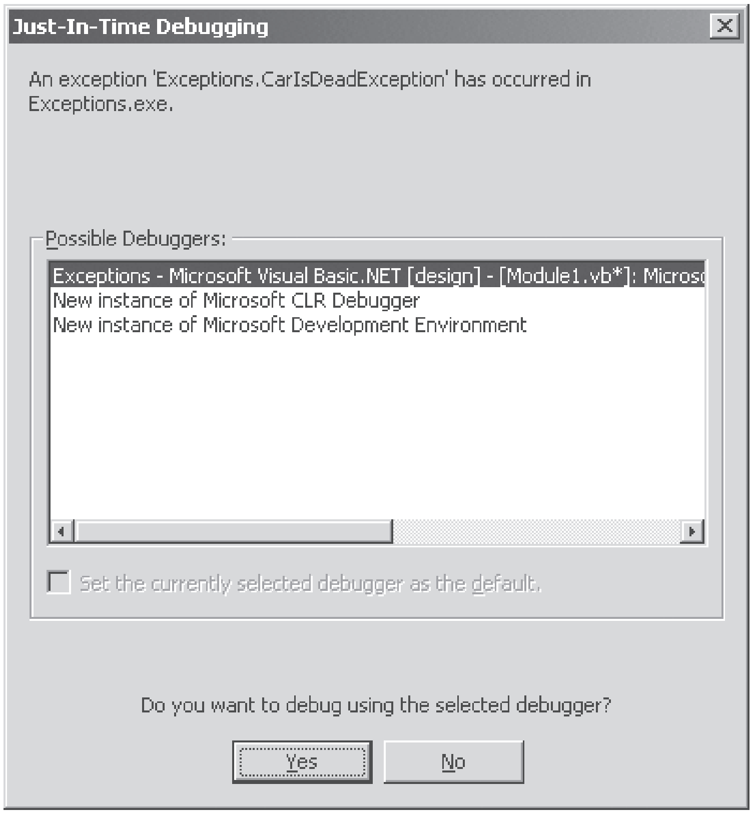
If you did not include a Finally block, the radio would *not* be turned off if an exception was caught (which may or may not be problematic). If you need to clean up any allocated memory, close down a file, detach from a data source (or whatever), you must add that code within a Finally block to ensure proper clean up. It is important to realize, that the code contained within a Finally block executes *every time* even if the logic within your try clause does not generate an exception.

Final Thoughts Regarding Exceptions

Unlike ad hoc error handling techniques, .NET exceptions cannot be ignored. One obvious question that may be on your mind is what would happen if you do not handle an exception thrown your direction? Assume that the logic in Main() that increases the speed of the Car object has no error-handling logic. The result of ignoring the generated error would be highly obstructive to the end user of your application, as the "last chance exception" dialog is displayed (Figure 4-19).

Now that you see the inherent goodness in catching exceptions, you might ask what you are to do with exceptions once they are caught. Again, this is a design issue based on your current project. In your trivial Car example, you dumped your custom message and call stack to the console. A more realistic scenario can include freeing up acquired resources or writing to a log file. The exception-handling schema is simply a pattern to follow when sending and receiving errors. What you do with them is largely up to you.

Finally, it is important to keep in mind that exceptions should only be thrown if the underlying problem is truly fatal. In other words, if you are able to recover from a user, logical, or general design error without throwing a system defined or custom exception, do so. In this light, the CarIsDeadException may be of arguable necessity. Chapter 6 revisits the SpeedUp() method, and substitutes the custom exception with a more appropriate custom event.


Just-In-Time Debugging

An exception 'Exceptions.CarIsDeadException' has occurred in Exceptions.exe.

Possible Debuggers:

Exceptions - Microsoft Visual Basic.NET [design] - [Module1.vb*]: Microso
New instance of Microsoft CLR Debugger
New instance of Microsoft Development Environment

Set the currently selected debugger as the default.

Do you want to debug using the selected debugger?

Figure 4-19. Exceptions that are not caught are a major buzz kill.

On Error Goto?

Last but not least, you have the issue of VB 6.0 error handling. To be honest, "On Error Goto" was never a very elegant way to handle runtime anomalies in your code. Nevertheless, VB .NET still supports this syntax for the sake of code migration. If you really wanted to, you are free to define a tag within a given method that can intercept the Err object. Do note that the Err object now has a method named GetException() which returns the underlying System.Exception type. To illustrate:

```
Public Sub OldStyleError()
    On Error Goto OOPS
    Dim c As New Car("Bill", 100, 0)
    Dim i As Integer
    For i = 0 To 10
        c.SpeedUp(10)
    Next
    Exit Sub
```

```
OOPS:
    Console.WriteLine(Err.Description())
    Dim e As Exception = Err.GetException()
    Console.WriteLine(e.Message)
End Sub
```

SOURCE CODE *The Exceptions project is included under the Chapter 4 subdirectory.*

Understanding Object Lifetime

As a VB .NET programmer, the rules of memory management are simple: Use the New keyword to allocate an object onto the managed heap. The .NET runtime destroys the object when it is no longer needed. Next question: How does the runtime determine when an object is no longer needed? The short (i.e., incomplete) answer is that the runtime deallocates memory when there are no longer any outstanding references to an object within the current scope (or if a reference has been explicitly set to Nothing). To illustrate:

```
' Create a local Car variable.
Sub Main()
     ' Place a car onto the managed heap.
     Dim c as Car = New Car("Viper", 200, 100)
End Sub  ' If c is the only reference to the Car object,
         ' it can be reclaimed when it drops out of scope.
```

Now, assume that your application has allocated three Car types. As long as there is enough room on the heap, you are returned a reference to each object in memory. Technically speaking, references to an object on the managed heap are called a *root*. The process can be visualized as illustrated in Figure 4-20.

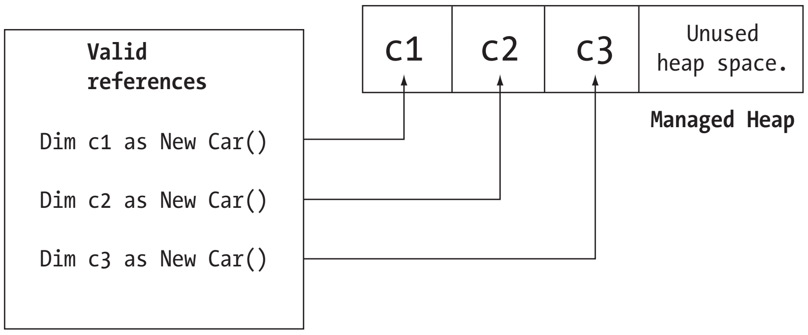


Figure 4-20. Valid references point to a location on the managed heap

As you are busy creating more and more objects, the managed heap may eventually become full. If you attempt to create a new object on a heap plump and full of active object references, an OutOfMemoryException exception is thrown. Therefore, if you want to be extremely defensive in your coding practices, you could allocate an object as follows:

```
' Try to add these cars to the managed heap and check for errors. . .
Public Sub Main()
. . .
    Dim yetAnotherCar as Car
     Try
          yetAnotherCar = New Car()
     Catch e as OutOfMemoryException
          Console.WriteLine(e.Message)
          Console.WriteLine("Managed heap is FULL! Running GC. . .")
     End Try
. . .
End Sub
```

Regardless of how defensive your object allocation logic may be, understand that when the memory allocated to the managed heap runs dry, the garbage collection algorithm kicks in automatically. At this time, all objects on the managed heap are tested for outstanding object references in your application (i.e., active roots). If the garbage collector determines that a given root is no longer used by a given application (i.e., the object has fallen out of scope or was set to Nothing), the object is marked for termination. Once the entire heap has been searched for "severed roots," the heap is swept clean, and the underlying memory is reclaimed.

Finalizing an Object Reference

VB 6.0 supplied a Terminate event, which could be handled on a class-by-class basis in order to ensure that proper clean up of the object occurred. Under VB .NET, the .NET garbage collection scheme is rather nondeterministic. .NET class types do *not* support a Terminate event which is automatically called when the object is no longer in use. In fact, you are typically unable to determine exactly when an object will be deallocated from memory. Although this approach to memory management can simplify coding efforts (allocate and forget) you are left with the unappealing byproduct of your objects possibly holding onto unmanaged resources (Hwnds, database connections, etc.) longer than necessary.

For example, if the Car type was to obtain a connection to a remote database during its lifetime, you would like to ensure that this resource released in a timely manner. One choice you face as a VB .NET class designer is to determine whether or not your classes should override the System.Object.Finalize() method (the default implementation does nothing). To illustrate, let's update our Car type:

```
Public Class Car
    Protected Overrides Sub Finalize()
        Console.WriteLine("In Finalize.")
    End Sub
. . .
End Class
```

When you place a VB .NET object onto the managed heap using the New keyword, the runtime automatically determines if your object supports a custom Finalize() method. If so, the object is marked as "finalizable." When the runtime determines a finalizable object is no longer referenced, it is placed onto a separate area of the heap named the "finalization queue." If a garbage collection occurs, each object on the finalization queue will have its Finalize() method called before deallocating the memory for the object.

Finalization Details

Assume that you have now defined some additional automobile classes (minivans, sports cars, and jeeps). Also assume that MiniVans and SportsCars do not override Finalize(), while Car and Jeep types do. When Car and Jeeps are no longer referenced, they are moved to the finalization queue and have their Finalize() method called at the next garbage collection. Internally, the process would look something like what you see in Figure 4-21.

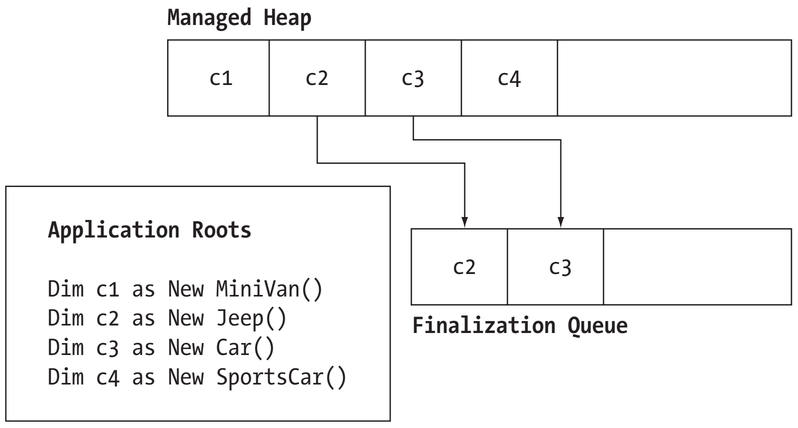


Figure 4-21. Objects that override Finalize() are placed onto the finalization queue

As you may be able to infer, classes that support custom Finalize() methods take longer to remove from memory. C1 and c4 do not override Finalize(), and can therefore be deallocated from memory immediately (if a garbage collection were to occur). C2 and c3 on the other hand, have additional overhead imposed by the call to Finalize(). Nevertheless, when you want to ensure that your objects are given a chance to release any acquired resources, you should support override Finalize() on your custom objects.

Building an Ad Hoc Destruction Method

Again assume the Car class obtains resources during its lifetime. If this type overrides Finalize(), it will take longer to remove from memory than objects that do not (which may or may not be a problem). Given the fact that resources such as database connections are a precious commodity, you may not want to wait for the .NET garbage collector to trigger your Finalize() logic at "some time in the future." A logical question at this point is how you can provide a way for the object user to deallocate the resources held by an object as soon as possible.

One alternative is to define a custom ad hoc method that you can assume all objects in your system implement. Let's call this method Dispose(). The assumption is that when object users are finished using your object, they manually call Dispose() before allowing the object reference to drop out of scope. In this way, your objects can perform any amount of cleanup necessary (i.e., release a database connection) without incurring the hit of being placed on the finalization queue and without waiting for the garbage collector to trigger the class' Finalize() logic:

```
' Equipping our class with an ad hoc destructor.
Public Class Car
. . .
     ' This is a custom method we expect the object user to call manually.
     Public Sub Dispose()
          ' . . . Clean up your Internal resources.
     End Sub
End Class
```

The IDisposable Interface

In order to provide symmetry among all objects that support an explicit destruction routine, the .NET class libraries define an interface named IDisposable which (surprise, surprise) supports a single member named Dispose(). Here is the official C# definition:

```
public interface IDisposable
{
     public void Dispose();
}
```

Now, rest assured that the concepts behind interface-based programming are fully detailed in Chapter 5. Until then, understand that the recommended design pattern to follow is to implement the IDisposable interface for all types that want to support an explicit form of resource deallocation. Thus, you may update the Car type as follows:

```
Public Class Car
    Implements IDisposable
...
     ' This is still a custom method we expect the object user to call manually.
     Public Sub Dispose() Implements IDisposable.Dispose
          ' . . . Clean up your Internal resources.
     End Sub
End Class
```

Again, using this approach, you provide the object user with a way to manually dispose of acquired resources as soon as possible, and avoid the overhead of being placed on the finalization queue. As you may guess, it is possible for a single VB .NET class to support an overridden Finalize() method as well as implement the IDisposable interface. You will see this technique in just a moment.

Interacting with the Garbage Collector

Like everything in the .NET universe, you are able to interact with the garbage collector using an object reference. System.GC is the class that enables you to do so. GC is a sealed class, which, as you recall, means it cannot function as a base class to other types. You access the GC's functionality using a small set of shared members. Here is a rundown of some of the more interesting items (Table 4-3).

Table 4-3. Select Members of the System.GC Type

SYSTEM.GC MEMBER	MEANING IN LIFE
Collect()	Forces the GC to call the Finalize() method for every object on the managed heap. You can also (if you choose) specify the generation to sweep (more on generations soon).
GetGeneration()	Returns the generation to which an object currently belongs.
MaxGeneration	This property returns the maximum of generations supported on the target system.
ReRegisterForFinalize()	Sets a flag indicating that a suppressed object should be reregistered as finalizable. This (of course) assumes the object was marked as nonfinalizable using SuppressFinalize().
SuppressFinalize()	Sets a flag indicating that a given object should not have its Finalize() method called (i.e., it should be taken off the finalization queue).
GetTotalMemory()	Returns the amount of memory (in bytes) currently being used by all objects in the heap, including objects that are soon to be destroyed. This method takes a Boolean parameter, which is used to specify if a garbage collection should occur during the method invocation.

To illustrate programmatic interaction with the .NET garbage collector, let's retrofit our automobile's destruction logic as follows:

```
Public Class Car
    Implements IDisposable
    ' Internal state data. . .
    Private currSpeed As Integer
    Private maxSpeed As Integer
    Private petName As String
    ' Used to mark if we are currently disposed.
    Private disposed As Boolean
    ' Constructors. . . (removed for clarity)
    . . .
    ' This helper function will be called by
    ' explicit and implicit destruction methods.
    Private Sub CleanUpInternalResources()
```

```
        If (disposed = False) Then
            disposed = True
            Console.WriteLine("Cleaning up internal resources. . .")
        End If
    End Sub
    ' This will be called by the runtime when a GC is needed.
    Protected Overrides Sub Finalize()
        Console.WriteLine("In Finalize() for {0}!", petName)
        CleanUpInternalResources()
    End Sub
    ' Called by the client when they are done.
    Public Sub Dispose() Implements IDisposable.Dispose
        Console.WriteLine("In Dispose() for {0}!", petName)
        ' No need to finalize if user
        ' called Dispose() manually.
        CleanUpInternalResources()
        System.GC.SuppressFinalize(Me)
    End Sub
End Class
```

Notice that this iteration of the Car class supports both an overridden implementation of Finalize() as well as the IDisposable interface. In both cases, a call is made to an internal private helper method named CleanUpInternalResources(). Assume this method does some sort of clean up for Car types as long as this method has *not* been called previously (thus the need for the Private Boolean type to check for the "disposedness" of the object).

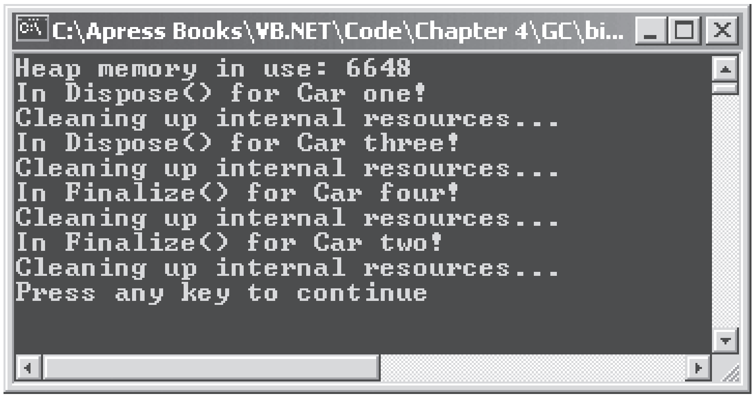
This time, the Dispose() method has been altered to call GC.SuppressFinalize(), which informs the system that it should remove the specified object from the finalization queue, as the object user has called Dispose() manually (and has therefore cleaned up any internal resources of the Car type).

To illustrate the interplay between explicit and implicit object deallocation, assume the following updated Main() method. GC.Collect() is called to force all objects on the finalization queue to have their Finalize() method triggered before this application shuts down. However, given that two of the Car types have been manually disposed by the object user, these types do not have their Finalize() methods triggered due to the call to GC.SuppressFinalize():

```
' Interacting with the GC.
Public Sub Main()
    Console.WriteLine("Heap memory in use: {0}",  _
                     System.GC.GetTotalMemory(false).ToString())
    ' Add these cars to the managed heap.
    Dim c1, c2, c3, c4 as Car
    c1 = New Car("Car one", 40, 10)
```

```
        c2 = New Car("Car two", 70, 5)
        c3 = New Car("Car three", 200, 100)
        c4 = New Car("Car four", 140, 80)
        ' Manually dispose some objects.
        ' This will tell the GC to suppress finialization.
        c1.Dispose()
        c3.Dispose()
        ' Call Finalize() for objects remaining on the finalization queue.
        System.GC.Collect()
    End Sub
```

Here is the output (Figure 4-22).



C:\Apress Books\VB.NET\Code\Chapter 4\GC\bi...

```
Heap memory in use: 6648
In Dispose() for Car one!
Cleaning up internal resources...
In Dispose() for Car three!
Cleaning up internal resources...
In Finalize() for Car four!
Cleaning up internal resources...
In Finalize() for Car two!
Cleaning up internal resources...
Press any key to continue
```

Figure 4-22. Cleaning up our resources

Garbage Collection Optimizations

The next topic of interest has to do with the notion of "generations." When the .NET garbage collector is about to mark objects for deletion, is does *not* literally walk over each and every object placed on the managed heap looking for orphaned roots. Doing so would involve considerable time, especially in larger (i.e., real-world) applications.

Recall that the GC forces a collection as soon as it determines there is not enough memory to hold a new object instance. If the GC were to search every single object in memory for severed roots, this could easily entail checking hundreds, if not thousands, of objects. In this case, you could easily envision sluggish performance.

To help optimize the collection process, every object on the heap is assigned to a given "generation." The idea behind generations is simple: The longer an object has existed on the heap, the more likely it is to stay there (such as the application level object). Conversely, objects that have been recently placed on the heap are more likely to be unreferenced by the application rather quickly (e.g., a temporary object created in some method scope). Given these assumptions, each object belongs to one of the following generations:

- Generation 0: Identifies a newly allocated object that has never been marked for collection.
- Generation 1: Identifies an object that has survived a garbage collection sweep (i.e., it was marked for collection, but was not removed due to the fact that the heap had enough free space).
- Generation 2: Identifies an object that has survived more than one sweep of the garbage collector.

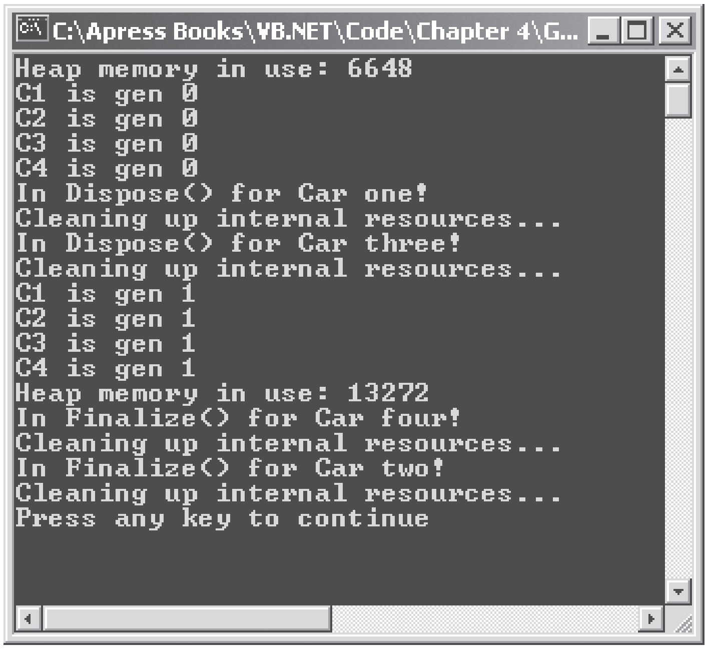
Now, when a collection occurs, the GC marks and sweeps all generation 0 objects first. If this results in the required amount of memory, the remaining objects are promoted to the next available generation. If all generation 0 objects have been removed from the heap, but more memory is still necessary, generation 1 objects are marked and swept, followed (if necessary) by generation 2 objects. In this way, the newer objects (i.e., local variables) are removed quickly while an older object is assumed to be in use. In a nutshell, the GC is able to quickly free heap space using the generation as a baseline.

Programmatically speaking, you are able to investigate the generation an object currently belongs to using GC.GetGeneration(). Furthermore, GC.Collect() does allow you to specify which generation should be checked for orphaned roots. Consider the following:

```
' Just how old are you?
Public Sub Main()
     Console.WriteLine("Heap memory in use: {0}", _
                  System.GC.GetTotalMemory(False).ToString())
     ' Add these cars to the managed heap.
     Dim c1, c2, c3, c4 as Car
     c1 = New Car("Car one", 40, 10)
     c2 = New Car("Car two", 70, 5)
     c3 = New Car("Car three", 200, 100)
     c4 = New Car("Car four", 140, 80)
     ' Display generations.
     Console.WriteLine("C1 is gen {0}", System.GC.GetGeneration(c1))
     Console.WriteLine("C2 is gen {0}", System.GC.GetGeneration(c2))
     Console.WriteLine("C3 is gen {0}", System.GC.GetGeneration(c3))
     Console.WriteLine("C4 is gen {0}", System.GC.GetGeneration(c4))
     ' Dispose some cars manually.
```

```
        c1.Dispose()
        c3.Dispose()
        ' Collect all gen 0 objects?
        System.GC.Collect(0)
        ' Display generations again (each will be promoted).
        Console.WriteLine("C1 is gen {0}", System.GC.GetGeneration(c1))
        Console.WriteLine("C2 is gen {0}", System.GC.GetGeneration(c2))
        Console.WriteLine("C3 is gen {0}", System.GC.GetGeneration(c3))
        Console.WriteLine("C4 is gen {0}", System.GC.GetGeneration(c4))
        ' Force memory to be freed for all generations.
        System.GC.Collect()    ' Calls Finalize() for each finalizable object.
        Console.WriteLine("Heap memory in use: {0}", _
                     System.GC.GetTotalMemory(False).ToString())
    End Sub
```

The output is shown in Figure 4-23. Notice that when you request a collection of generation 0, each object is promoted to generation 1, given that these objects did not need to be removed from memory (as the managed heap was not exhausted):



```
Heap memory in use: 6648
C1 is gen 0
C2 is gen 0
C3 is gen 0
C4 is gen 0
In Dispose() for Car one!
Cleaning up internal resources...
In Dispose() for Car three!
Cleaning up internal resources...
C1 is gen 1
C2 is gen 1
C3 is gen 1
C4 is gen 1
Heap memory in use: 13272
In Finalize() for Car four!
Cleaning up internal resources...
In Finalize() for Car two!
Cleaning up internal resources...
Press any key to continue
```

Figure 4-23. Interacting with the garbage collector

To close, keep in mind that your interactions with the GC should be slim-to-none. The whole point of having a managed heap is to move the responsibility of memory management from your hands into the hands of the runtime. Do remember however, that when you build classes that override Finalize(), your objects will require more time to be removed from the managed heap (due to the extra logic of the finalization queue). If you want to support an implicit means of freeing the resources used by an object, you may implement the IDisposable interface.

SOURCE CODE *The GC project is located under the Chapter 4 subdirectory.*

Summary

If you already come to the universe of .NET from another object-oriented language (such as C#, C++, Java, or Delphi), this chapter may have been more of a quick compare and contrast between your current language of choice and VB .NET. On the other hand, those of you who are exploring complete OOP concepts for the first time may have found many of the concepts presented here a bit confounding. Regardless of your background, rest assured that the information presented here is the foundation of any .NET application.

This chapter began with a review of the pillars of OOP: Encapsulation, inheritance, and polymorphism. As you have seen, VB .NET provides full support for each aspect of object orientation. In addition, the use of structured exception handling was introduced, which is *the* way to report and respond to error information in the .NET platform.

Finally, the chapter wrapped up by examining exactly how the .NET runtime frees you from manually cleaning up the memory you allocate by the virtue of a managed heap. You have also explored the interplay between Object.Finalize(), the IDisposable interface and the VB .NET destructor and examined how to programmatically interact with the garbage collector using the System.GC type.