Principles of Object-Oriented Programming

In general, when learning a new programming language, you will spend a fair part of your effort learning the syntax of that language: how to declare variables, how to control the flow of execution, and so on. However, in order to write quality code, you also need to understand the principles and methodologies behind the language. C# is a fully object-oriented language, so in order to create well-designed C# code you need to get to grips with its object-oriented features, and that means learning about *object-oriented programming* (*OOP*).

In OOP, we aim to write easily maintainable and reusable pieces of code that can perform collectively very complex tasks. However, the whole structure of an object-oriented program is very different from the structure of an equivalent program written in a procedural language. In this appendix, we introduce the principles of object-oriented programming. Although we refer to some C# syntax (since we'll be presenting the examples in C#) throughout this appendix, the emphasis is on learning those principles that apply to OOP in general, no matter which language you are using.

OOP is an extremely powerful methodology. Once you've familiarized yourself with writing your code using OOP, you will probably wonder how you ever got by without it. You'll find that, unlike procedural languages, OOP gives your code an intuitive, "natural" structure. Even Visual Basic 6, which implements a few object-oriented features, cannot keep up with true OOP.

We'll start by discussing the nature of an object before moving on to examine the concept of *inheritance*. Inheritance, which is at the heart of OOP, enables you to conveniently reuse the code for classes. We'll show you how to use inheritance in your programs both from a conceptual and C# point of view.

A Note for Visual Basic 6 Programmers

If you are a skilled Visual Basic 6 developer but do not have C++ or Java experience, then many of the concepts in this chapter will strike you as "foreign." Visual Basic does allow you to code something that is often referred to as an object: the Visual Basic class module. Some texts even refer to this as involving OOP, although this bears little resemblance to the original concepts of OOP. It is more accurate to say that Visual Basic implements a few of the more basic features of OOP. A VB class module is essentially a COM component but wrapped in a way that hides much of what it does. In particular it does not support inheritance of its methods in the same way that inheritance is used in C# and conventional OOP.

Because of its support for a different kind of inheritance (implementation inheritance), C# classes are much more powerful than VB class modules and are often used very differently. If you want to write good C# .NET applications and assemblies, you must read this appendix. Objects and inheritance are not just new language features. In a well-designed object-oriented program, the whole architecture of the program is often arranged around inheritance. Once you're comfortable with the concept of OOP, you'll be structuring your programs in a completely different way from what you have done in Visual Basic—and your programs will be easier for others to maintain as a result. However, if you already feel comfortable with manipulating objects in Visual Basic, but have not yet used inheritance, you might want to skip ahead to the section on inheritance.

Note that whenever we refer to Visual Basic in this appendix, we are more specifically referring to Visual Basic 6.

What Is an Object?

In everyday life, an object is anything that is identifiably a single material item. An object can be a car, a house, a book, a document, or a paycheck. For our purposes, we're going to extend that concept a bit and think of an object as anything that is a single item that you might want to represent in a program. We'll therefore also include living "objects," such as a person, an employee, or a customer, as well as more abstract "objects," such as a company, a database, or a country.

Thinking about objects in this way not only enables us to write code that models the real world; it also enables us to break up a large program into smaller, more manageable units. The idea really comes from the concept of a *black box* that you might have encountered in school science.

The idea of a black box is that there are a lot of objects in life that you are able to use but of which you don't understand the mechanism. Take for example a car radio. Most people don't know exactly how a car radio works; however, they do know what it does and how to operate it. Furthermore, they can take out the radio, plug in a different one and it'll do basically the same thing, even though the internal workings of it might be completely different. Black boxes formalize the idea that there's a difference between what something does and how it works, and that two objects can do the same thing but work differently on the inside.

Replacing one object with another does have some subtle effects. Car radios might have different knobs and switches, and they might project different sound qualities, but the basic function is unchanged. Another important point is that the basic user interface is unchanged—you plug one car stereo into the slot in much the same way as you would another.

If you understand all that, then you basically understand OOP, because OOP is about applying these same concepts to computer programming. If, in other areas of our lives, we use objects that have a well-designed interface that we are familiar with, and we know how to use them, but don't care how they work, why not do the same thing in your programs? In other words, break each program into lots of units and design each unit to perform a clearly specified role within the program. That's basically what an object is.

If you start thinking about your programs this way, you gain quite a few advantages. You'll find it becomes easier to design the programs. The architecture of the programs becomes more intuitive and easier to understand because it more closely reflects whatever it is that the program is abstracting from real life. It becomes easier for multiple developers to work together, since they can work on different objects in the code; all they need to know is what an object can do and how to interface with it. They don't have to worry about the details of how the underlying code works.

Objects in Programming

Now that we've established what an object is conceptually (and in everyday life), we can discuss more specifically how to apply these concepts to programming.

If you've programmed on Windows before, chances are you're already familiar with objects. For example, think about the various controls that you can place in Windows, including text boxes, list boxes, buttons, and so on. Microsoft has written these controls for you, so that you don't need to know, for example, how a text box works internally. You just know that it does certain things. For example, you can set its `Text` property to display text on screen, or you can set its `Width` property to have the text box resize itself.

In programming, we need to distinguish between a *class* and an *object*. A class is the generic definition of what an object is—a template. For example, a class could be "car radio"—the abstract idea of a car radio. The class specifies what properties an object must have to qualify as a car radio.

Class Members

So far, we've emphasized that there are two sides to an object: what it does, which is usually publicly known, and how it works, which is usually hidden. In programming, the "what it does" is normally represented in the first instance by *methods*, which are blocks of functionality that you can use. A method is just C# parlance for a function. The "how it works" is represented both by methods and by any data (variables) that the object stores. In Java and C++, this data is described as member variables, while in Visual Basic this data would be represented by any module-level variables in the class module. In C# the terminology is *fields*. In general, a class is defined by its fields and methods.

We'll also use the term *member* by itself to denote anything that is part of a class, be it a field, method, or any of the other items just mentioned that can be defined within a class.

Defining a Class

The easiest way to understand how to code a class is by looking at an example. In the following sections, we're going to develop a simple class called `Authenticator`. We'll assume we're in the process of writing a large application, which at some point requires users to log in and supply a password. `Authenticator` is the name of the class that will handle this aspect of the program. We won't worry about the rest of the application—we'll just concentrate on writing this class. However, we will also write a small piece of test harness code to verify that `Authenticator` works as intended.

`Authenticator` allows us to do two things: set a new password, and check whether a password is valid. The C# code we need to define the class looks like this:

```
public class Authenticator
{
   private string password  = "";

   public bool IsPasswordCorrect(string tryPassword)
   {
      return (tryPassword == password) ? true : false;
   }

   public bool ChangePassword(string oldPassword, string newPassword)
   {
      if (oldPassword == password)
      {
         password = newPassword;
         return true;
      }
      else
         return false;
   }
}
```

The keyword `class` in C# indicates that we are going to define a new class (type of object). The word immediately following `class` is the name we're going to use for this class. Then the actual definition of the object—consisting of variables (fields) and methods—follows in braces. In this example, the definition consists of one field, `password`, and two methods, `IsPasswordCorrect()` and `ChangePassword()`.

Access Modifiers

The only field in `Authenticator`, `password`, stores the current password (initially an empty string when an `Authenticator` object is created) and is marked by the keyword `private`. This means that it is not visible outside the class, only to code that is part of the `Authenticator` class itself. Marking a field or method as `private` effectively ensures that that field or method will be part of the internal working of the class, as opposed to the external interface. The advantage of this is that if you decide to change the internal working (perhaps you later decide not to store `password` as a string but to use some other more specialized data type), you can just make the change without breaking the code outside the `Authenticator` class definition—nothing from outside of this class can access this field.

Any code that uses the `Authenticator` class can only access the methods that have been marked with the keyword `public`—in this case the `IsPasswordCorrect()` and `ChangePassword()` methods. Both of these methods have been implemented in such a way that nothing will be done (other than returning `true` or `false`) unless the calling code supplies the current correct password, as you'd expect for software that implements security. The implementations of these functions access the `password` field, but that's fine because this code forms part of the `Authenticator` class itself. Notice that these public functions simultaneously give us the interface to the external world (in other words, any other code that uses the `Authenticator` class) and define what the `Authenticator` class does, as viewed by the rest of the world.

> ***`private`*** *and* ***`public`*** *are not the only access modifiers available to define what code is allowed to know about the existence of a member. Later in this appendix we'll discuss* ***`protected`****, which makes the member available to this class and certain related classes. C# also allows members to be declared as* ***`internal`*** *and* ***`protected internal`****, which restrict access to other code within the same assembly.*

Instantiating and Using Objects

The easiest way to understand how to use a class in our code is to think of the class as a new type of variable. You're used to the predefined variable types—such as `int`, `float`, `double`, and so on. By defining the `Authenticator` class, we've effectively told the compiler that there's a new type of variable called an `Authenticator`. The class definition contains everything the compiler needs to know to be able to process this variable type. Therefore, just as the compiler knows that a `double` contains a floating-point number stored in a certain format (which enables you to add `doubles`, for example), we've told the compiler that a variable of type `Authenticator` contains a `string` and allows you to call the `IsPasswordCorrect()` and `ChangePassword()` methods.

> *Although we've described a class as a new type of variable, the more common terminology is data type, or simply type.*

Creating a user-defined variable (an object) is known as *instantiation*, because we create an *instance* of the object. An instance is simply any particular occurrence of the object. So, if our `Authenticator` object is another kind of variable, we should be able to use it just like any other variable— and we can, as demonstrated in the following example.

Create the `MainEntryPoint` class, as shown in the following code sample, and place it in the `Wrox.ProCSharp.OOProg` namespace along with the `Authenticator` class we created earlier:

```
using System;

namespace Wrox.ProCSharp.OOProg
{
   class MainEntryPoint
   {
      static void Main()
      {
         Authenticator myAccess = new Authenticator();
         bool done;
         done = myAccess.ChangePassword("", "MyNewPassword");
```

```
            if (done == true)
               Console.WriteLine("Password for myAccess changed");
            else
               Console.WriteLine("Failed to change password for myAccess");

            done = myAccess.ChangePassword("", "AnotherPassword");
            if (done == true)
               Console.WriteLine("Password for myAccess changed");
            else
               Console.WriteLine("Failed to change password for myAccess");

            if (myAccess.IsPasswordCorrect("WhatPassword"))
               Console.WriteLine("Verified myAccess\' password");
            else
               Console.WriteLine("Failed to verify myAccess\' password");
         }
      }

      public class Authenticator
      {
         // implementation as shown earlier
      }

   }
```

The `MainEntryPoint` class is a class like `Authenticator`—it can have its own members (that is, its own fields, methods, and so on). However, we've chosen to use this class solely as a container for the program entry point, the `Main()` method. Doing it this way means that the `Authenticator` class can sit as a class in its own right that can be used in other programs (either by cutting and pasting the code or by compiling it separately into an assembly). `MainEntryPoint` only really exists as a class because of the syntactical requirement of C# that even the program's main entry point has to be defined within a class, rather than being defined as an independent function.

Since all the action is happening in the `Main()` method, let's take a closer look at it. The first line of interest is:

```
            Authenticator myAccess = new Authenticator();
```

Here we are declaring and instantiating a new `Authenticator` object instance. Don't worry about `= new Authenticator()` for now—it's part of C# syntax, and is there because in C#, classes are always accessed by reference. We could actually use the following line if we just wanted to declare a new `Authenticator` object called `myAccess`:

```
   Authenticator myAccess;
```

This declaration can hold a reference to an `Authenticator` object, without actually creating any object (in much the same way that the line `Dim obj As Object` in Visual Basic doesn't actually create any object). The `new` operator in C# is what actually instantiates an `Authenticator` object.

Calling class methods is done using the period symbol (`.`) appended to the name of the variable:

```
done = myAccess.ChangePassword("", "MyNewPassword");
```

Here we have called the `ChangePassword()` method on the `myAccess` instance and fed the return value into the `done` Boolean variable. We can retrieve class fields in a similar way. Note, however, that we cannot do this:

```
string myAccessPassword = myAccess.password;
```

This code will actually cause a compilation error, because the `password` field was marked as `private`, so other code outside the `Authenticator` class cannot access it. If we changed the `password` field to be `public`, then the previous line would compile and feed the value of `password` into the string variable.

You should note that if you are accessing member methods or fields from inside the same class, you can simply give the name of the member directly.

Now that you understand how to instantiate objects, call class methods, and retrieve public fields, the logic in the `Main()` method should be pretty clear. If we save this code as Authenticator.cs and then compile and run it, we will get this:

```
Authenticator
Password for myAccess changed
Failed to change password for myAccess
Failed to verify myAccess' password
```

There are a couple of points to note from the code. First, you'll notice that so far we're not doing anything new compared to what we would do when coding a Visual Basic class module; nor do we do anything that differs from the basic C# syntax that we cover in the first part of this book. All we wanted to do here is make sure that you are clear about the concepts behind classes.

Second, the previous example uses the `Authenticator` class directly in other code within the same source file. You'll often want to write classes that are used by other projects that you or others work on. In order to do this, you write the class in exactly the same way, but compile the code for the class into a library, as explained in Chapter 13.

Using Static Members

You may have noticed in our example that the `Main()` method was declared as `static`. In this section we are going to discuss what effect this `static` keyword has.

Creating static fields

It's important to understand that by default each instance of a class (each object) has its own set of all the fields you've defined in the class. For example, in the following snippet the instances `karli` and `julian` each contain their own string called `password`:

```
Authenticator julian = new Authenticator();
Authenticator karli = new Authenticator();
karli.ChangePassword("OldKarliPassword", "NewKarliPassword");
julian.ChangePassword("OldJulianPassword", "NewJulianPassword");
```

Changing the password in `karli` has no effect on the password in `julian`, and vice versa (unless the two references happen to be pointing to the same address in memory, which is something we'll come to later). This situation resembles Figure A-1.

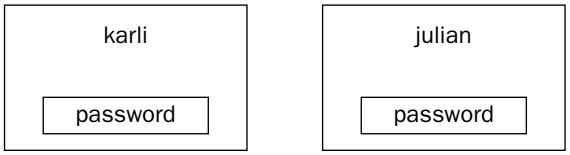

Figure A-1

There are some cases in which this might not be the behavior you want. For example, suppose we want to define a minimum length for all `password`s (and therefore for all of the `password` fields in all instances) in our `Authenticator` class. We do not want each password to have its own minimum length. Therefore, we really want the minimum length to be stored only once in memory, no matter how many instances of `Authenticator` we create.

To indicate that a field should only be stored once, no matter how many instances of the class we create, we place the keyword `static` in front of the field declaration in our code:

```
public class Authenticator
{

   private static uint minPasswordLength = 6;

   private string password = "";
```

Storing a copy of `minPasswordLength` with each `Authenticator` instance not only wastes memory but also causes problems if we want to be able to change its value! By declaring the field as `static`, we ensure that it is only stored once, and this field is shared among all instances of the class. Note that in this code snippet we also set an initial value. Fields declared with the `static` keyword are referred to as *static fields* or *static data*, while fields that are not declared as static are referred to as *instance fields* or *instance data*. Another way of looking at this is that an instance field belongs to an object, while a static field belongs to the class.

> **VB developers shouldn't confuse static fields with static variables in Visual Basic, which are variables whose values remain between invocations of a method.**

If a field has been declared as `static`, then it exists when your program is running from the moment that the particular module or assembly containing the definition of the class is loaded—that is as soon as your code tries to use something from that assembly, so you can always guarantee a static variable is there when you want to refer to it. This is independent of whether you actually create any instances of that class. By contrast, instance fields only exist when there are variables of that class currently in scope—one set of instance fields for each variable.

In some ways static fields perform the same functions as global variables performed for older languages such as C and FORTRAN.

You should note that the `static` keyword is independent of the accessibility of the member to which it applies. A class member can be `public static` or `private static`.

Creating static methods

As we explained in our `Authenticator` example, by default a method such as `ChangePassword()` is called against a particular instance, as indicated by the name of the variable in front of the period (`.`) operator. That method then implicitly has access to all the members (fields, methods, and so on) of that particular instance.

However, just as with fields, it is possible to declare methods as `static`, provided that they do not attempt to access any instance data or other instance methods. For example, we might want to provide a method to allow users to view the minimum password length:

```
public class Authenticator
{
   private static uint minPasswordLength = 6;

   public static uint GetMinPasswordLength()
   {
      return minPasswordLength;
   }

...
```

You can download the code for `Authenticator` with this modification from the Wrox Press Web site (`www.wrox.com`) as the `Authenticator2` sample.

In our earlier `Authenticator` example, the `Main()` method of the `MainEntryPoint` class is declared as `static`. This allows it to be invoked as the entry point to the program, despite the fact that no instance of the `MainEntryPoint` class was ever created.

Accessing static members

The fact that static methods and fields are associated with a class rather than an object is reflected in how you access them. Instead of specifying the name of a variable before the `.` operator, you specify the name of the class, like this:

```
Console.WriteLine(Authenticator.GetMinPasswordLength());
```

Also notice that in the above code we access the `Console.WriteLine()` method by specifying the name of the class, `Console`. That is because `WriteLine()` is a static method too—we don't need to instantiate a `Console` object to use `WriteLine()`.

How instance and static methods are implemented in memory

We said earlier that each object stores its own copy of a class's instance fields. This is, however, not the case for methods. If each object had its own copy of the code for a method, it would waste a lot of mem-

ory, since the code for the methods remains the same across all object instances. Therefore, instance methods, just like static methods, are stored only once, and associated with the class as a whole. Later on, we'll discuss other types of class members (constructors, properties, and so on) that contain code rather than data and follow the same logic.

Figure A-2 shows how instance and static methods are implemented in memory.

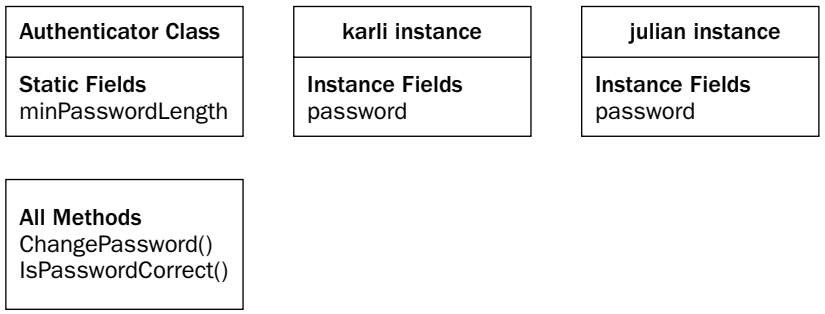

Figure A-2

If instance methods are only stored once, how is a method able to access the correct copy of each field? In other words, how can the compiler generate code that accesses Karli's password with the first method call and Julian's with the second in the following example?

```
karli.ChangePassword("OldKarliPassword", "NewKarliPassword");
julian.ChangePassword("OldJulianPassword", "NewJulianPassword");
```

The answer is that instance methods actually take an extra implicit parameter, which is a reference to where in memory the relevant class instance is stored. You can almost think of this code example as the user-friendly version that you have to write, because that's how C# syntax works. However, what's actually happening in your compiled code is this:

```
ChangePassword(karli, "OldKarliPassword", "NewKarliPassword");
ChangePassword(julian, "OldJulianPassword", "NewJulianPassword");
```

Declaring a method as `static` makes calling it slightly more efficient, because it will not be passed this extra parameter. On the other hand, if a method is declared as `static`, but attempts to access any instance data, the compiler will raise an error for the obvious reason that you can't access instance data unless you have the address of a class instance!

This means that in our `Authenticator` sample we could not declare `ChangePassword()` or `IsPasswordCorrect()` as `static`, because both of these methods access the password field, which is not `static`.

Interestingly, although the hidden parameter that comes with instance methods is never declared explicitly, you do actually have access to it in your code. You can get to it using the keyword `this`. We can rewrite the code for the `ChangePassword()` method as follows:

```
        public bool ChangePassword(string oldPassword, string newPassword)
        {

            if (oldPassword == this.password)

            {

                this.password = newPassword;

                return true;
            }
            else
                return false;
        }
```

Generally, you wouldn't write your code like this unless you have to distinguish between variable names. All we've achieved here is to make the method longer and slightly harder to read.

A Note About Reference Types

Before we leave the discussion of classes, we ought to point out one potential gotcha that can occur in C# because C# regards all classes as reference types. This can have some unexpected effects when it comes to comparing instances of classes for equality and setting instances of classes equal to each other. For example, look at this code:

```
Authenticator User1;
Authenticator User2 = new Authenticator();
Authenticator User3 = new Authenticator();
User1 = User2;
User2.ChangePassword ("", "Tardis");  // This sets password for User1 as well!
User3.ChangePassword ("", "Tardis");
if (User2 == User3)
{
   // contents of this if block will NOT be executed even though
   // objects referred to by User2 and User3 are contain identical values,
   // because the variables refer to different objects
}
if (User2 == User1)
{
   // any code here will be executed because User1 and User2 refer
   // to the same memory
}
```

In this code we declare three variables of type `Authenticator`: `User1`, `User2`, and `User3`. However, we only instantiate two objects of the `Authenticator` class, because we only use the `new` operator twice. Then we set the variable `User1` equal to `User2`. Unlike with a value type, this does not copy any of the contents of `User2`. Rather, it means that `User1` is set to refer to the same memory as `User2` is referring to. What that means is that any changes we make to `User2` also affect `User1`, because they are not separate objects; both variables refer to the same data. We can also say that they *point to* the same data, and the actual data referred to is sometimes described as the *referent*. So when we set the password of `User2` to `Tardis`, we are implicitly also setting the password of `User1` to `Tardis`. This is very different from how value types behave.

The situation gets even less intuitive when we try to compare `User2` and `User3` in the next statement:

```
if (User2 == User3)
```

You might expect that this condition returns `true`, since `User2` and `User3` have both been set to the same password, so both instances contain identical data. The comparison operator for reference types, however, doesn't compare the contents of the data by default—it simply tests to see whether the two references are referring to the same address in memory. Because they are not, this test returns `false`, which means anything inside this `if` block will not be executed. By contrast, comparing `User2` with `User1` returns `true` because these variables do point to the same address in memory.

Note that this behavior does not apply to strings, because the == operator has been overloaded for strings. Comparing two strings with == always compares string content. (Any other behavior for strings would be extremely confusing!)

Overloading Methods

To *overload* a method is to create several methods each with the same name, but each with a different signature. The reason why you might want to use overloading is best explained with an example. Consider how in C# we write data to the command line, using the `Console.WriteLine()` method. For example, if we want to display the value of an integer, we can write this:

```
int x = 10;
Console.WriteLine(x);
```

To display a string we can write:

```
string message = "Hello";
Console.WriteLine(message);
```

Even though we are passing different data types to the same method, both of these examples compile. This is because there are actually lots of `Console.WriteLine()` methods, but each has a different signature—one of them takes `int` as a parameter, while another one takes `string`, and so on. There is even a two-parameter overload of the method that allows for formatted output and lets you write code like this:

```
string Message = "Hello";

Console.WriteLine("The message is {0}", Message);
```

Obviously, Microsoft provides all of these `Console.WriteLine()` methods because it realizes that there are many different data types of which you might want to display the value.

Method overloading is very useful, but there are some pitfalls to be aware of when using it. Suppose we write:

```
short y = 10;
Console.WriteLine(y);
```

A quick look at the documentation reveals that no overload of `WriteLine()` takes `short`. So what will the compiler do? In principle, it could generate code that converts `short` to `int` and call the `int` version of `Console.WriteLine()`. Or it could convert `short` to `long` and call `Console.WriteLine(long)`. It could even convert `short` to `string`.

In this situation, each language will have a set of rules for what conversion will be the one that is actually performed (for C#, the conversion to `int` is the preferred one). However you can see the potential for confusion. For this reason, if you define method overloads, you need to take care to do so in a way that won't cause any unpredictable results.

When to use overloading

Generally, you should consider overloading a method when you need a number of methods that take different parameters, but conceptually do the same thing, as with `Console.WriteLine()` above. The situations in which you will normally use overloading are explained in the following subsections.

Optional parameters

One common use of method overloads is to allow certain parameters to a method to be optional and to have default values if the client code does not specify their values explicitly. For example, consider this code:

```
public void DoSomething(int x, int y)
{
   // do whatever
}

public void DoSomething(int x)
{
   DoSomething(x, 10);
}
```

These overloads allow client code to call `DoSomething()`, supplying one required parameter and one optional parameter. If the optional parameter isn't supplied, we effectively assume the second `int` is `10`. Most modern compilers will also inline method calls in this situation so there is no performance loss. This is certainly true of the .NET JIT compiler.

> *Some languages, including Visual Basic and C++, allow default parameters to be specified explicitly in function declarations, with a syntax that looks like public void DoSomething(int X, int Y=10). C# does not allow this; in C# you have to simulate default parameters by providing multiple overloads of methods as shown in the previous example.*

Different input types

We have already discussed this very common reason for defining overloads in the `Console.WriteLine()` example.

Different output types

This situation is far less common; however, occasionally you might have a method that calculates or obtains some quantity, and depending on the circumstances, you might want this to be returned in more

than one way. For example, in an airline company, you might have a class that represents aircraft timetables, and you might want to define a method that tells you where an aircraft should be at a particular time. Depending on the situation, you might want the method to return either a string description of the position ("over Atlantic Ocean en route to London") or the latitude and longitude of the position.

You cannot distinguish overloads using the return type of a method. However, you can do so using `out` parameters. So you could define these:

```
void GetAircraftLocation(DateTime Time, out string Location)
{
...
}

void GetAircraftLocation(DateTime Time, out float Latitude, out float Longitude)
{
...
}
```

Note, however, that in most cases using overloads to obtain different `out` parameters does not lead to an architecturally neat design. In the above example, a better design would perhaps involve defining a `Location` struct that contains the location string as well as the latitude and longitude and returning this from the method call, hence avoiding the need for overloads.

Properties

Earlier we mentioned that a class is defined by its fields and methods. However, classes can also contain other types of class members, including constructors, indexers, properties, delegates, and events. For the most part these other items are used only in more advanced situations and are not essential to understanding the principles of object-oriented design. For that reason, we will only discuss properties, which are extremely common and can significantly simplify the external user interface exposed by classes, in this appendix. The other class members are introduced in Part I. Properties are in extremely common use, however, and can significantly simplify the external user interface exposed by classes. For this reason, we'll discuss them here.

> *Visual Basic programmers will find that C# properties correspond almost exactly to properties in VB class modules and are used in just the same way.*

Properties exist for the situation in which you want to make a method call look like a field. You can see what a property is by looking at the `minPasswordLength` field in our `Authenticator` class. Let's extend the class so that users can read and modify this field without having to use a `GetMinPasswordLength()` method like the one we introduced earlier.

A property is a method or pair of methods that are exposed to the outside world as if they are fields. To create a property for the minimum password length we modify the code for the `Authenticator` class as follows:

```
        public static uint MinPasswordLength
        {
           get
           {
```

```
            return minPasswordLength;
         }
         set
         {
            minPasswordLength = value;
         }
      }
```

As we can see from this, we define a property in much the same way as a field, except that after the name of the property, we have a code block enclosed by curly braces. In the code block there may be two methods called `get` and `set`. These are known as the *get accessor* and the *set accessor*. Note that although no parameter is explicitly mentioned in the definition of the `set` accessor, there is an implicit parameter passed in, and referred to by the name `value`. Also, the `get` accessor always returns the same data type as the property was declared as (in this case `uint`).

Now, to retrieve the value of `minPasswordLength`, we use this syntax:

```
uint i = Authenticator.MinPasswordLength;
```

What will actually happen here is that `MinPasswordLength` property's `get` accessor is called. In this case, this method is implemented to simply return the value of the `minPasswordLength` field.

To set the `MinPasswordLength` field using the property, we use the following code:

```
Authenticator.MinPasswordLength = 7;
```

This code causes the `MinPasswordLength`'s `set` accessor to be called, which is implemented to assign the required value (7) to the `minPasswordLength` field. We mentioned earlier that the `set` accessor has an implicit parameter, called `value`.

Note that in this particular example, the property in question happens to be static. In general that is not necessary. Just as for methods, you will normally declare properties as static only if they refer to static data.

Data encapsulation

You may wonder what the point of all the above code is. Wouldn't it have been easier to make the `minPasswordLength` field `public`, so that we could access it directly and not have to bother about any properties? The answer is that fields represent the internal data of an object, so they are an integral part of the functionality of an object. Now, in OOP, we aim to make it so that users of objects only need to know what an object does, not how it does it. So making fields directly accessible to users defeats the ideology behind OOP.

Ideology is all very well, but there must be practical reasons behind it. One reason is this: if we make fields directly visible to external users, we lose control over what they do to the fields. They might modify the fields in such a way as to break the intended functionality of the object (give the fields inappropriate values, let's say). However, if we use properties to control access to a field, this is not a problem because we can add functionality to the property that checks for inappropriate values. Related to this, we can also provide read-only properties by omitting the `set` accessor completely. The principle of hiding fields from client code in this way is known as *data encapsulation*.

You should only use properties to do something that appears only to set or retrieve a value; in all other instances use methods. That means that the `set` accessor must only take one parameter and return a void, while the `get` accessor cannot take any parameters. For example, it would not be possible to rewrite the `IsPasswordValid()` method in the `Authenticator` class as a property. The parameter types and return value for this method are not of the correct type.

Introducing Inheritance

One characteristic of objects in everyday life is that they tend to come in families of related things that share aspects of their design. A sofa is just like an armchair, except that it can seat more than one person. A CD-ROM does the same sort of thing as a cassette tape, but with extra direct-access facilities. Likewise, many cars differ in body style and size, but internally their engines and other components are built in much the same way, often using the same components.

This is an example of *implementation inheritance*, and the equivalent in OOP would be some classes (`EscortCar`, `OrionCar`, and `FiestaCar`, perhaps?), which not only expose methods with the same names, but actually the same methods, in the sense that when you call the methods you are running the same code.

Let's now extend this example. Say that I swapped my Escort for another Escort that has a diesel engine. Both cars have exactly the same body shell (the user interface is the same) but under the hood, the engines are different. That's an example of *interface inheritance*, and the equivalent in computer programming would be two classes (`EscortCar` and `EscortDieselCar`) that happen to expose methods that have the same names, purposes, and signatures, but different implementations.

> *Java or C++ developers who are familiar with COM will recognize that implementation inheritance is the kind of inheritance that is supported by Java/C++ and other traditional object-oriented languages, while the more restricted interface inheritance was the only form of inheritance that was supported by COM and COM objects. Visual Basic supports only interface inheritance through the* ***`Implements`*** *keyword. The great thing about C# is it supports both types of inheritance.*

As far as C# programming is concerned, we're looking at the issue of how to define a new class, while reusing features from an existing class. The benefits are twofold: First, inheritance provides a convenient way to reuse existing, fully tested code in different contexts, thereby saving a lot of coding time; second, inheritance can provide even more structure to your programs by giving a finer degree of granularity to your classes.

At this point we're going to move on to a new coding example, based on a cell phone company, that will demonstrate how implementation inheritance works in a C# program. Inheritance of classes in C# is always implementation inheritance.

Using Inheritance in C#

The example we'll use to demonstrate inheritance is going to be of a fictitious cell phone company, which we'll call Mortimer Phones. We're going to develop a class that represents a customer account and is responsible for calculating that customer's phone bill. This is a much longer, more complex example than the `Authenticator` class, and as it develops we'll quickly find that one simple class is not adequate; instead, we will need a number of related classes, and inheritance is the solution to our challenge.

We're going to write a class that works out the monthly bill for each customer of Mortimer Phones. The class is called `Customer`, and each instance of this class represents one customer's account. In terms of public interface, the class contains two properties:

- `Name` represents the customer's name (read-write).
- `Balance` represents the amount owed (read-only).

The class also has two methods:

- `RecordPayment()`, which is called to indicate that the customer has paid a certain amount of their bill.
- `RecordCall()`, which is called when the customer has made a phone call. It works out the cost of the call and adds it to that customer's balance.

The `RecordCall()` method is potentially quite a complex function when applied to the real world, since it would involve figuring out the type of call from the number called, then applying the appropriate rate, and keeping a history of the calls. To keep things simple, we'll assume there are just two types of calls: calls to landlines, and calls to other cell phones, and that each of these are charged at a flat rate of 2 cents a minute for landlines and 30 cents a minute for other cell phones. Our `RecordCall` method will simply be passed the type of call as a parameter, and we won't worry about keeping a call history.

With this simplification, we can look at the code for the project. The project is a console application, and the first thing in it is an enumeration for the types of call:

```
namespace Wrox.ProCSharp.OOProg
{
   using System;

   public enum TypeOfCall
   {
      CallToCellPhone, CallToLandline
   }
```

Now, let's look at the definition of the `Customer` class:

```
   public class Customer
   {
      private string name;
      private decimal balance;

      public string Name
      {
         get
         {
            return name;
         }
         set
         {
            name = value;
         }
```

```
        }

        public decimal Balance
        {
           get
           {
              return balance;
           }
        }

        public void RecordPayment(decimal amountPaid)
        {
           balance -= amountPaid;
        }

        public void RecordCall(TypeOfCall callType, uint nMinutes)
        {
           switch (callType)
           {
              case TypeOfCall.CallToLandline:
                 balance += (0.02M * nMinutes);
                 break;
              case TypeOfCall.CallToCellPhone:
                 balance += (0.30M * nMinutes);
                 break;
              default:
                 break;
           }
        }
     }
```

This code should be reasonably self-explanatory. Note that we hardcode the call charges of 2 cents per minute (landline) and 30 cents per minute (pay-as-you-go charges for a cell phone) into the program. In real life, they'd more likely to be read in from a relational database, or some file that allows the values to be changed easily.

Now let's add some code in the program's `Main()` method that displays the amounts of bills currently owed:

```
     public class MainEntryPoint
     {
        public static void Main()
        {
           Customer arabel = new Customer();
           arabel.Name = "Arabel Jones";
           Customer mrJones = new Customer();
           mrJones.Name = "Ben Jones";
           arabel.RecordCall(TypeOfCall.CallToLandline, 20);
           arabel.RecordCall(TypeOfCall.CallToCellPhone, 5);
           mrJones.RecordCall(TypeOfCall.CallToLandline, 10);
           Console.WriteLine("{0,-20} owes ${1:F2}", arabel.Name, arabel.Balance);
           Console.WriteLine("{0,-20} owes ${1:F2}", mrJones.Name, mrJones.Balance);
        }
     }
  }
```

Running this code gives the following results:

```
MortimerPhones
Arabel Jones          owes $1.90
Ben Jones             owes $0.20
```

Adding inheritance

Currently, the Mortimer Phones example is heavily simplified. In particular, it only has one call plan for all customers, which is not even remotely realistic. Many people are registered under a call plan for which they pay a fixed rate each month, but there are many other plans.

The way we're working at the moment, if we try to take all of the different call plans into account, our `RecordCall()` method is going to end up containing various nested `switch` statements and looks something like this (assuming the `CallPlan` field is an enumeration):

```
public void RecordCall(TypeOfCall callType, uint nMinutes)
{
   switch (callplan)
   case CallPlan.CallPlan1:
   {
      switch (callType)
      {
      case TypeOfCall.CallToLandline:
         // work out amount

      case TypeOfCall.CallToCellPhone:

         // work out amount
         // other cases
    // etc.

   }
   case CallPlan.CallPlan2:
   {
      switch (callType)
      {

// etc.

}
```

That is not a satisfactory solution. Small `switch` statements are nice, but huge `switch` statements with large numbers of options—and in particular embedded `switch` statements—make for code that is difficult to follow. It also means that whenever a new call plan is introduced the code for the method will have to be changed. This could accidentally introduce new bugs into the parts of the code responsible for processing existing call plans.

The problem has really to do with the way the code for the different call plans is mixed up in a `switch` statement. If we could cleanly separate the code for the different call plans then the problem would be solved. This is one of the issues that inheritance addresses.

We want to separate the code for different types of customers. We'll start by defining a new class that represents customers on a new call plan. We'll name this call plan Nevermore60. Nevermore60 is designed for customers who use their cell phones a lot. Customers on this call plan pay a higher rate of 50 cents a minute for the first 60 minutes of calls to other cell phones, then a reduced rate of 20 cents a minute for all additional calls, so if they make a large enough number of calls they save money compared to the previous call plan.

We'll save actually implementing the new payment calculations for a little while longer, and we'll initially define `Nevermore60Customer` like this:

```
public class Nevermore60Customer : Customer
{

}
```

In other words, the class has no methods, no properties, nothing of its own. On the other hand, it's defined in a slightly different way from how we've defined any classes before. After the class name is a colon, followed by the name of our earlier class, `Customer`. This tells the compiler that `Nevermore60Customer` is *derived* from `Customer`. That means that every member in `Customer` also exists in `Nevermore60Customer`. Alternatively, to use the correct terminology, each member of `Customer` is *inherited* in `Nevermore60Customer`. Also, `Nevermore60Customer` is said to be a *derived class*, while `Customer` is said to be the *base class*. You'll also sometimes encounter derived classes referred to as subclasses, and base classes as super-classes or parent classes.

Since we've not yet put anything else in the `Nevermore60Customer` class, it is effectively an exact copy of the definition of the `Customer` class. We can create instances of and call methods against the `Nevermore60Customer` class, just as we could with `Customer`. To see this, we'll modify one of the customers, `Arabel`, to be a `Nevermore60Customer`:

```
public static void Main()
{

   Nevermore60Customer arabel = new Nevermore60Customer();

   ...
}
```

In this code, we've changed just one line, the declaration of `Arabel`, to make this customer a `Nevermore60Customer` instance. All the method calls remain the same, and this code produces exactly the same results as our earlier code. If you want to try this out, it's the `MortimerPhones2` code sample (which is part of the sample download file, available at `www.wrox.com`).

By itself, having a copy of the definition of the `Customer` class might not look very useful. The power of this comes from the fact we can now make some modifications or additions to `Nevermore60Customer`. We can instruct the compiler, "`Nevermore60Customer` is almost the same as `Customer`, but with these differences." In particular, we're going to modify the way that `Nevermore60Customer` works out the charge for each phone call according to the new tariff.

The differences we can specify in principle are:

- We can add new members (of any type: fields, methods, properties, and so on) to the derived class, where these members are not defined in the base class.
- We can replace the implementation of existing members, such as methods or properties, that are already present in the base class.

For our example, we will replace, or *override*, the `RecordCall()` method in `Customer` with a new implementation of the `RecordCall()` method in `Nevermore60Customer`. Not only that, but whenever we need to add a new call plan, we can simply create another new class derived from `Customer`, with a new override of `RecordCall()`. In this way we can add code to cope with many different call plans, while keeping the new code separate from all the existing code that is responsible for calculations using existing call plans.

> *Don't confuse method overriding with method overloading. The similarity in these names is unfortunate as they are completely different, unrelated, concepts. Method overloading has nothing to do with inheritance or virtual methods.*

So let's modify the code for the `Nevermore60Customer` class, so that it implements the new call plan. To do this we need not only to override the `RecordCall()` method, but also to add a new field that indicates the number of high cost minutes that have been used:

```
public class Nevermore60Customer : Customer
{
   private uint highCostMinutesUsed;
   public override void RecordCall(TypeOfCall callType, uint nMinutes)
   {
      switch (callType)
      {
         case TypeOfCall.CallToLandline:
            balance += (0.02M * nMinutes);
            break;
         case TypeOfCall.CallToCellPhone:
      uint highCostMinutes, lowCostMinutes;
      uint highCostMinutesToGo =
         (highCostMinutesUsed < 60) ? 60 - highCostMinutesUsed : 0;
      if (nMinutes > highCostMinutesToGo)
      {
         highCostMinutes = highCostMinutesToGo;
         lowCostMinutes = nMinutes - highCostMinutes;
      }
      else
      {
         highCostMinutes = nMinutes;
         lowCostMinutes = 0;
      }
      highCostMinutesUsed += highCostMinutes;
      balance += (0.50M * highCostMinutes + 0.20M *
         lowCostMinutes);
      break;
```

```
            default:
               break;
         }
      }
   }
```

You should note that the new field we've added, `highCostMinutesUsed`, is only stored in instances of `Nevermore60Customer`. It is not stored in instances of the base class, `Customer`. The base class itself is never implicitly modified in any way by the existence of the derived class. This must always be the case, because when you code the base class, you don't necessarily know what other derived classes might be added in the future—and you wouldn't want your code to be broken when someone adds a derived class!

As you can see, the algorithm to compute the call cost in this case is more complex, though if you follow through the logic you will see it does meet our definition for the Nevermore60 call plan. Notice that the extra keyword `override` has been added to the definition of the `RecordCall()` method. This informs the compiler that this method is actually an override of a method that is already present in the base class, and we must include this keyword.

Before this code will compile, we need to make a couple of modifications to the base class, `Customer`, too:

```
public class Customer
{
   private string name;

   protected decimal balance;


// etc.


   public virtual void RecordCall(TypeOfCall callType, uint nMinutes)

   {
      switch (callType)
```

The first change we've made is to the `balance` field. Previously it was defined with the `private` keyword, meaning that no code outside the `Customer` class could access it directly. Unfortunately this means that, even though `Nevermore60Customer` is derived from `Customer`, the code in the `Nevermore60Customer` class cannot directly access this field (even though a `balance` field is still present inside every `Nevermore60Customer` object). That would prevent `Nevermore60Customer` from being able to modify the balance when it records calls made, and so prevent the code we presented for the `Nevermore60Customer.RecordCall()` method from compiling.

The access modifier keyword `protected` solves this problem. It indicates that any class that is derived from `Customer`, as well as `Customer` itself, should be allowed access to this member. The member is still invisible, however, to code in any other class that is not derived from `Customer`. Essentially, we're assuming that, because of the close relationship between a class and its derived class, it's fine for the derived class to know a bit about the internal workings of the base class, at least as far as protected members are concerned.

There is actually a controversial point here about good programming style. Many developers would regard it as better practice to keep all fields private, and write a protected accessor method to allow derived classes to modify the balance. In this case, allowing the balance field to be protected rather than private prevents our example from becoming more complex than it already is.

The second change we've made is to the declaration of the `RecordCall()` method in the base class. We've added the keyword `virtual`. This changes the manner in which the method is called when the program is run, in a way that facilitates overriding it. C# will not allow derived classes to override a method unless that method has been declared as `virtual` in the base class. We will be looking at virtual methods and overriding later in this appendix.

Class Hierarchies and Class Design

In a procedural language, and even to some extent in a language like Visual Basic, the emphasis is very much on breaking the program down into functions. Object orientation shifts the emphasis of program design away from thinking about what functionality the program has to considering instead what objects the program consists of.

Inheritance is also an extremely important feature of object-oriented programming, and a crucial stage in the design of your program is deciding on *class hierarchies*—the relationship between your classes. In general, as with our Mortimer Phones example, you will find that you have a number of specialized objects that are particular types of more generic objects.

When you're designing classes it's normally easiest to use a diagram known as a *class hierarchy diagram*, which illustrates the relationships between the various base and derived classes in your program. Traditionally, class hierarchy diagrams are drawn with the base class at the top and arrows pointing from derived classes to their immediate base classes. For example, the hierarchy of our Mortimer Phones examples from `MortimerPhones3` onward look like what is shown in Figure A-3.

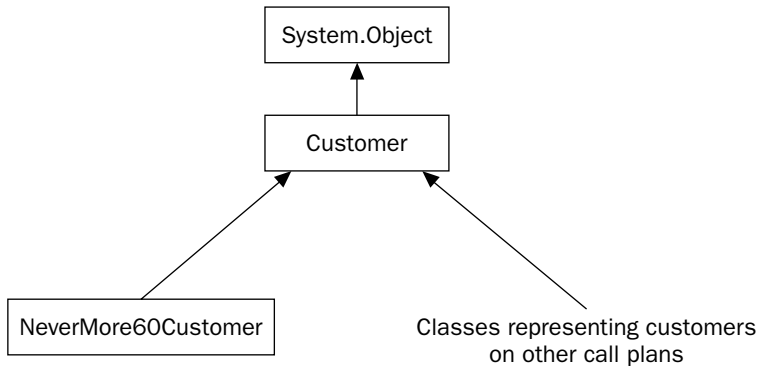

Figure A-3

The above class hierarchy diagram emphasizes that inheritance can be direct or indirect. In our example, `Nevermore60Customer` is directly derived from `Customer`, but indirectly derived from `Object`. Although the examples in our discussion are tending to focus on direct derivation, all the principles apply equally when a class indirectly derives from another class.

Another example is one of the hierarchies from the .NET base classes. In Chapter 19, we see how to use the base classes that encapsulate windows (or to give them their more modern .NET terminology, forms). You may not have realized just how rich a hierarchy could be behind some of the controls that you can place on windows (see Figure A-4).

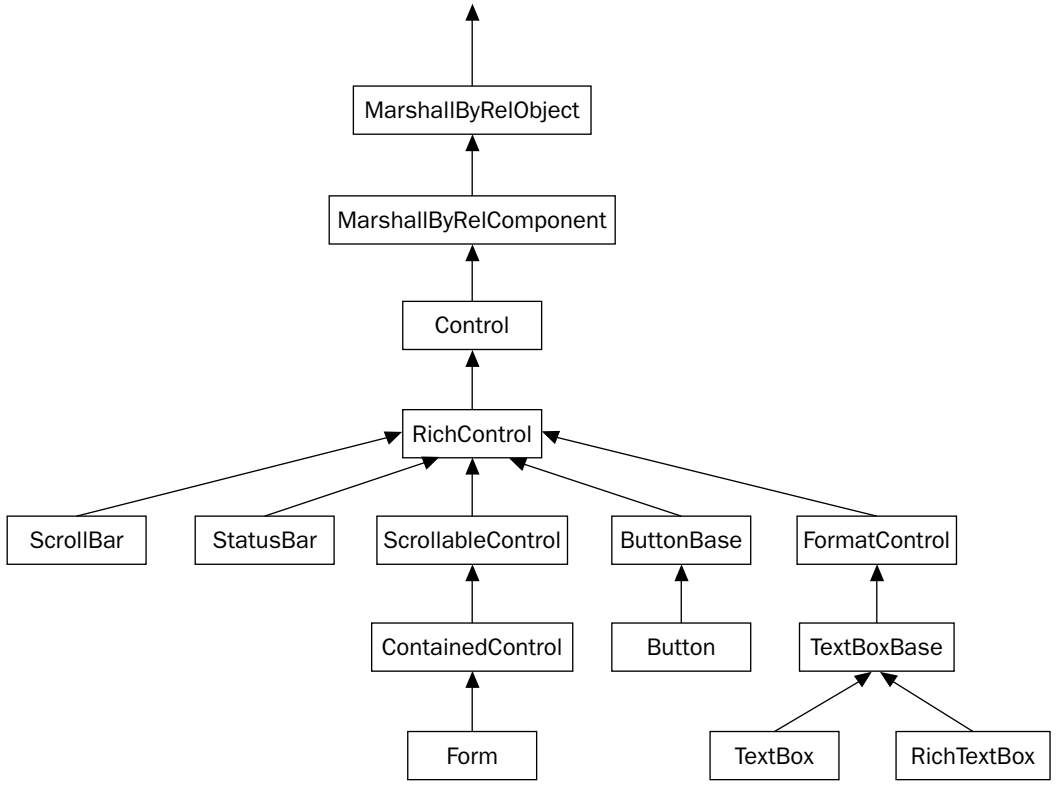

Figure A-4

The `Form` class represents the generic window, while `ScrollBar`, `StatusBar`, `Button`, `TextBox`, and `RichTextBox` represent the familiar corresponding controls. The rich hierarchy behind these classes allows a fine-tuning of what implementations of which methods can be made common to a number of different classes. Many of these classes will also implement certain interfaces, by which they can make their nature as windows known to client code.

It's also important to realize that class hierarchies are, like any other aspect of programming, an area in which there may be many possible solutions, each having its advantages and disadvantages. For our Mortimer Phones example, there may be other ways to design classes. One argument against our chosen hierarchy is that customers often change their call plans—and do we really want to have to destroy a customer object and instantiate a new one of a different class whenever that happens? Perhaps it would be better to have just one customer class, which contains a reference to a call plan object, and have a class hierarchy of call plans?

A large application will not have just one hierarchy, but will typically implement a large number of hierarchies that possibly stretches into hundreds of classes. That may sound daunting, but the alternative, before object-oriented programming came into being, was to have literally thousands of functions making up your program, with no way to group them into manageable units. Classes provide a very effective way of breaking your program into smaller sections. This not only facilitates maintenance but also makes your program easier to understand because the classes represent the actual objects that your program is representing in a very intuitive way.

It's also important with your classes to think carefully about the separation between the public interface that is presented to client code, and the private internal implementation. In general the more of a class you are able to keep private, the more modular your program will become, in the sense that you can make modifications or improvements to the internal implementation of one class and be certain that it will not break or even have any effect on any other part of the program. That's the reason that we've emphasized that member fields in particular will almost invariably be private, unless they are either constant or they form part of a struct whose main purpose is to group together a small number of fields. We haven't always kept to that rule rigidly in this appendix, but that's largely so we can keep the samples as simple as possible.

The object class

One point that you might not realize from the code is that in our Mortimer Phones examples, `Customer` is itself derived from another class, `System.Object`. This is a rule that is enforced by .NET and C#: All .NET classes must ultimately derive from a base class called `Object`. In C# code, if you write a class and do not specify a base class, the compiler will supply `System.Object` as the base class by default. This means that all objects in the .NET Framework have certain methods inherited from the `Object` class, including the `ToString()` and `GetType()` methods that are discussed in the beginning of the book. We look at the `Object` class in more detail in Chapter 10.

Single and multiple inheritance

In C#, each derived class can only inherit from one base class (although we can create as many different classes that are derived from the same base class as we want). The terminology to describe this is *single inheritance*. Some other languages, including C++, allow you to write classes that have more than one base class, which is known as *multiple inheritance*.

Polymorphism and Virtual Members

Let's go back to our Mortimer Phones example. Earlier, we encountered this line of code:

```
Nevermore60Customer arabel = new Nevermore60Customer();
```

In fact, we could have instantiated the `Nevermore60Customer` object like this as well:

```
Customer arabel = new Nevermore60Customer();
```

Because `Nevermore60Customer` is derived from `Customer`, it's actually perfectly legitimate for a reference to a `Customer` to be set up to point to either a `Customer` or a `Nevermore60Customer`, or to an instance of any other class that is derived directly or indirectly from `Customer`. Notice that all we've changed here is the declaration of the reference variable. The actual object that gets instantiated with `new` is still a `Nevermore60Customer` object. If for example, you try to call `GetType()` against it, it'll tell you it's a `Nevermore60Customer`.

Being able to point to derived classes with a base reference may look like just a syntactical convenience, but it's actually essential if we want to be able to use derived classes easily—and it's an essential feature of any language that wants to support OOP. We can understand why if we think about how a real cell phone company will want to store the various `Customer`-derived classes. In our example we only have two customers, so it is easy to define separate variables. In the real world, however, we have hundreds of thousands of customers, and we might want to do something like read them from a database into an array, then process them using the array, using code that looks like this:

```
Customer[] customers = new Customer[NCustomers];

   // do something to initialize customers

foreach (Customer nextCustomer in customers)
{
   Console.WriteLine("{0,-20} owes ${1:F2}", nextCustomer.Name,
                                             nextCustomer.Balance);
}
```

If we use an array of `Customer` references, each element can point to any type of customer, no matter what `Customer`-derived class is used to represent that customer. However, if variables could not store references to derived types we'd have to have lots of arrays—an array of `Customers`, an array of `Nevermore60Customers`, and another array for each type of class.

We've now ensured that we can mix different types of classes in one array, but this will now give the compiler a new problem. Suppose we have a snippet of code like this:

```
   Customer aCustomer;

   // Initialize aCustomer to a particular tariff

   aCustomer.RecordCall(TypeOfCall.CallToLandline, 20);
```

What the compiler can see is a `Customer` reference, and we are to call the `RecordCall()` method on it. The trouble is that `aCustomer` might refer to a `Customer` instance, or it might refer to a `Nevermore60Customer`, instance or it might refer to an instance of some other class derived from `Customer`. Each of these classes might have its own implementation of `RecordCall()`. How will the compiler determine which method should be called? There are two answers to this, depending on whether the method in the base class is declared as `virtual` and the derived class method as an `override`:

- If the methods are not declared as `virtual` and `override`, respectively, then the compiler will simply use the type that the reference was declared to be. In this case, since `aCustomer` is of type `Customer`, it will arrange for the `Customer.RecordCall()` method to be called, no matter what `aCustomer` is actually referring to.
- If the methods are declared as virtual and override, respectively, then the compiler will generate code that checks what the `aCustomer` reference is actually pointing to at runtime. It then identifies which class this instance belongs to and calls the appropriate `RecordCall()` override. This determination of which overload should be called will need to be made separately each time the statement is executed. For example, if the `virtual` method call occurs inside a `foreach` loop that executes 100 times, then on each iteration through the loop the reference might be pointing to a different instance and therefore to a different class of object.

In most cases, the second behavior is the one we want. If we have a reference, for example, to a `Nevermore60Customer`, then it's highly unlikely that we'd want to call any override of any method other than the one that applies to `Nevermore60Customer` instances. In fact, you might wonder why you'd ever want the compiler to use the first, non-virtual, approach, since it looks like that means in many cases the "wrong" override will be called up. Why we don't just make `virtual` methods the normal behavior, and say that every method is automatically `virtual`? This is, incidentally, the approach taken by Java, which automatically makes all methods `virtual`. There are three good reasons, however, for not doing this in C#:

- **Performance.** When a virtual method is called, a runtime determination has to be made to identify which override has to be called. For a non-virtual function, this information is available at compile time. (The compiler can identify the relevant override from the type that the reference is declared as!) This means that for a non-virtual function, the compiler can perform optimizations such as inlining code to improve performance. Inlining virtual methods is not possible, which will hurt performance. Another (minor) factor is that the determination of the method itself gives a very small performance penalty. This penalty amounts to no more than an extra address lookup in a table of virtual function addresses (called a vtable) and so is insignificant in most cases but may be important in very tight and frequently executed loops.
- **Design.** It may be the case that when you design a class there are some methods that should not be overridden. This actually happens a lot, especially with methods that should only be used internally within the class by other methods or whose implementations reflect the internal class design. When you design a class, you choose which features of its implementation you make public, protected, or private. It's unlikely that you'll want methods that are primarily concerned with the internal operation of the class to be overrideable, so you typically won't declare these methods as virtual.
- **Versioning.** Virtual methods can cause a particular problem connected with releasing new versions of base classes.

The ability of a variable to be used to reference objects of different types, and to automatically call the appropriate version of the method of the object it references, is more formally known as *polymorphism*. However, you should note that in order to make use of polymorphism, the method you are calling must exist on the base class as well as the derived class. For example, suppose we add some other method, such as a property called `HighCostMinutesLeft`, to `Nevermore60Customer` in order to allow users to find out this piece of information. Then the following would be legal code:

```
Nevermore60Customer mrLeggit = new Nevermore60Customer();

   // processing

int minutesLeft = mrLeggit.HighCostMinutesLeft;
```

The following, however, would not be legal code, because the `HighCostMinutesLeft` property doesn't exist in the `Customer` base class:

```
Customer mrLeggit = new Nevermore60Customer();

   // processing

int minutesLeft = mrLeggit.HighCostMinutesLeft;
```

We also ought to mention some other points about virtual members:

- It is not only methods that can be overridden or hidden. You can do the same thing with any other class member that has an implementation, including properties.
- Fields cannot be declared as `virtual` or overridden. However, it is possible to hide a base version of a field by declaring another field of the same name in a derived class. In that case, if you wanted to access the base version from the derived class, you'd need to use the syntax `base.<field_name>`. Actually, you probably wouldn't do that anyway, because you'd have all your fields declared as `private`.
- Static methods and so on cannot be declared as `virtual`, but they can be hidden in the same way that instance methods and other methods can be. It wouldn't make sense to declare a `static` member as `virtual`; `virtual` means that the compiler looks up the instance of a class when it calls that member, but `static` members are not associated with any class instance.
- Just because a method has been declared as `virtual`, that doesn't mean that it has to be overridden. In general, if the compiler encounters a call to a virtual method, it will look for the definition of the method first in the class concerned. If the method isn't defined or overridden in that class, it will call the base class version of the method. If the method isn't derived there, it'll look in the next base class, and so on, so that the method executed will be the one closest in the class hierarchy to the class concerned. (Note that this process occurs at compile time, when the compiler is constructing the vtable for each class. There is no impact at runtime.)

Method Hiding

Even if a method has not been declared as `virtual` in a base class, it is still possible to provide another method with the same signature in a derived class. The *signature* of a method is the set of all information needed to describe how to call that method: its name, number of parameters, and parameter types. However, the new method will not override the method in the base class. Rather, it is said to *hide* the base class method. As we've implied earlier, what this means is that the compiler will always examine the data type of the variable used to reference the instance when deciding which method to call. If a method hides a method in a base class, then you should normally add the keyword `new` to its definition. Not doing so does not constitute an error, but it will cause the compiler to give you a warning.

Realistically, method hiding is not something you'll often want to do deliberately, but we'll demonstrate how it works by adding a new method called `GetFunnyString()` to our `Customer` class and hiding it in `Nevermore60Customer()`. `GetFunnyString()` just displays some information about the class and is defined like this:

```
public class Customer
{

   public string GetFunnyString()
   {
      return "Plain ordinary customer. Kaark!";
   }


...
```

```
public class Nevermore60Customer : Customer
{

   public new string GetFunnyString()
   {
      return "Nevermore60. Nevermore!";
   }


...
```

`Nevermore60Customer`'s version of this function will be the one called up, but only if called using a variable that is declared as a reference to `Nevermore60Customer` (or some other class derived from `Nevermore60Customer`). We can demonstrate this with this client code:

```
public static void Main()
{

   Customer cust1;
   Nevermore60Customer cust2;
   cust1 = new Customer();
   Console.WriteLine("Customer referencing Customer: "
      + cust1.GetFunnyString());
   cust1 = new Nevermore60Customer();
   Console.WriteLine("Customer referencing Nevermore60Customer: "
      + cust1.GetFunnyString());
   cust2 = new Nevermore60Customer();
   Console.WriteLine("Nevermore60Customer referencing: "
      + cust2.GetFunnyString());
}
```

This code is downloadable as the `MortimerPhones3Funny` sample. Running the sample gives this result:

```
MortimerPhones3Funny
Customer referencing Customer: Plain ordinary customer. Kaark!
Customer referencing Nevermore60Customer: Plain ordinary customer. Kaark!
Nevermore60Customer referencing: Nevermore60. Nevermore!
```

Abstract Functions and Base Classes

So far, every time we've defined a class we've actually created instances of that class, but that's not always the case. In many situations, you'll define a very generic class from which you intend to derive other, more specialized classes but don't ever intend to actually use. C# provides the keyword `abstract` for this purpose. If a class is declared as `abstract` it is not possible to instantiate it.

For example, suppose we have an abstract class `MyBaseClass`, declared like this:

```
abstract class MyBaseClass
{
...
```

In this case the following statement will not compile:

```
MyBaseClass MyBaseRef = new MyBaseClass();
```

However, it's perfectly legitimate to have `MyBaseClass` references, so long as they only point to derived classes. For example, you can derive a new class from `MyBaseClass`:

```
class MyDerivedClass : MyBaseClass
{
...
```

In this case, the following is perfectly valid code:

```
MyBaseClass myBaseRef;
myBaseRef = new MyDerivedClass();
```

It's also possible to define a method as `abstract`. This means that the method is treated as a `virtual` method, and that you are not actually implementing the method in that class, on the assumption that it will be overridden in all derived classes. If you declare a method as `abstract` you do not need to supply a method body:

```
abstract class MyBaseClass
{
   public abstract int MyAbstractMethod();   // look no body!
   ...
```

If any method in a class is `abstract`, then that implies the class itself should be `abstract`, and the compiler will raise an error if the class is not so declared. Also, any non-abstract class that is derived from this class must override the `abstract` method. These rules prevent you from ever actually instantiating a class that doesn't have implementations of all its methods.

At this stage, you're probably wondering what the use of abstract methods and classes are for. They are extremely useful for two reasons. One is that they often allow a better design of class hierarchy, in which the hierarchy more closely reflects the situation you are trying to model. The other is that the use of abstract classes can shift certain potential bugs from hard-to-locate runtime errors into easy-to-locate compile-time errors. It's a bit hard to see how that works in practice without looking at an example, so let's improve the program architecture of `MortimerPhones` by rearranging the class hierarchy.

Defining an abstract class

We're not redesigning the Mortimer Phones sample just for the fun of it. There's actually a bit of a design flaw in the current class hierarchy. Our class `Customer` represents pay-as-you-go customers as the base class for all the other customer types. We're treating that kind of call plan as if it's a special call plan from which all the others are derived. That's not really an accurate representation of the situation. In reality, the pay-as-you-go call plan is just one of a range of call plans—there's nothing special about it—and a more carefully designed class hierarchy would reflect that. Therefore, in this section, we're going to rework the `MortimerPhones` sample to give it the class hierarchy shown in Figure A-5.

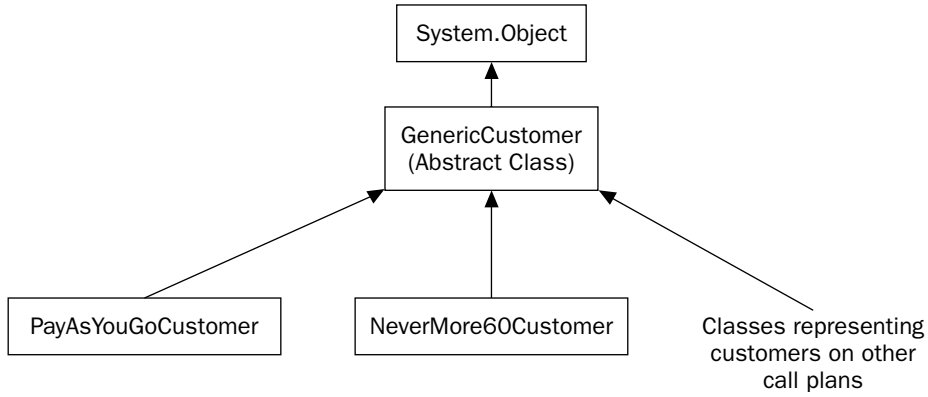

Figure A-5

Our old `Customer` class is gone. Our new abstract base class is `GenericCustomer`. `GenericCustomer` implements all the stuff that is common to all types of customers, such as methods and properties that have the same implementation for all customers and therefore are not virtual. This includes retrieving the balance or the customer's name, or recording a payment.

However, `GenericCustomer` does not provide any implementation of the `RecordCall()` method, which works out the cost of a given call and adds it to the customer's account. The implementation of this method is different for each call plan, so we require that every derived class supplies its own version of this method. Therefore, `GenericCustomer`'s `RecordCall()` method will be declared as `abstract`.

Having done that, we need to add a class that represents the pay-as-you-go customers. The `PayAsYouGoCustomer` class does this job, supplying the override to `RecordCall()` that with our previous hierarchy was defined in the base `Customer` class.

You may wonder whether it is really worth the effort in redesigning the sample class hierarchy in this way. After all, the old hierarchy worked perfectly well, didn't it? The reason for regarding the new hierarchy as a better designed architecture is simple: it removes a possible subtle source of bugs.

In a real application, `RecordCall()` probably wouldn't be the only virtual method that needed to be implemented separately for each call plan. What happens if later on someone adds a new derived class, representing a new call plan, but forgets to add the overrides of some of these methods? Well, with the old class hierarchy, the compiler would have automatically substituted the corresponding method in the base class. With that hierarchy, the base class represented pay-as-you-go customers, so we would have ended up with subtle runtime bugs involving the wrong versions of methods being called. With our new hierarchy, however, that won't happen. Instead, we'll get a compile-time error, with the compiler complaining that the relevant abstract methods haven't been overridden in the new class.

Anyway, on to the new code, and as you might have guessed by now, this is the `MortimerPhones4` sample. With the new hierarchy, the code for `GenericCustomer` looks like this. Most of the code is the same

as for our old `Customer` class; in the following code we've highlighted the few lines that are different. Note the `abstract` declaration for the `RecordCall()` method:

```
public abstract class GenericCustomer

{
   ...
   public void RecordPayment(decimal amountPaid)
   {
      balance -= amountPaid;
   }

   public abstract void RecordCall(TypeOfCall callType, uint nMinutes);

}
```

Now for the implementation of the pay-as-you-go customers. Again, notice that most of the code is taken directly from the former, obsolete `Customer` class. The only real difference is that `RecordCall()` is now an override rather than a virtual method:

```
public class PayAsYouGoCustomer : GenericCustomer

{

   public override void RecordCall(TypeOfCall callType, uint nMinutes)

   {
      // same implementation as for Customer
   }
}
```

We won't display the full code for `Nevermore60Customer` here as the `RecordCall()` override in this class is long and completely identical to the earlier version of the example. The only change we need to make to this class is to derive it from `GenericCustomer` instead of from the `Customer` class, which no longer exists:

```
public class Nevermore60Customer : GenericCustomer

{
   private uint highCostMinutesUsed;
   public override void RecordCall(TypeOfCall callType, uint nMinutes)
   {
      // same implementation as for old Nevermore60Customer
   }
   ...
```

To finish, we'll add some new client code to demonstrate the operation of the new class hierarchy. This time we've actually used an array to store the various customers, so this code shows how an array of ref-

erences to the abstract base class can be used to reference instances of the various derived classes, with the appropriate overrides of the methods being called:

```
        public static void Main()
        {

            GenericCustomer arabel = new Nevermore60Customer();
            arabel.Name = "Arabel Jones";
            GenericCustomer mrJones = new PayAsYouGoCustomer();
            mrJones.Name = "Ben Jones";
            GenericCustomer [] customers = new GenericCustomer[2];
            customers[0] = arabel;
            customers[0].RecordCall(TypeOfCall.CallToLandline, 20);
            customers[0].RecordCall(TypeOfCall.CallToCellPhone, 5);
            customers[1] = mrJones;
            customers[1].RecordCall(TypeOfCall.CallToLandline, 10);
            foreach (GenericCustomer nextCustomer in customers)
            {
               Console.WriteLine("{0,-20} owes ${1:F2}", nextCustomer.Name,
                                                         nextCustomer.Balance);
            }

        }
```

Running this code produces the correct results for the amounts owed:

```
MortimerPhones4
Arabel Jones         owes $2.90
Ben Jones         owes $0.20
```

Sealed Classes and Methods

In many ways you can think of a sealed class or method as the opposite of an abstract class or method. Whereas declaring something as abstract means that it must be overridden or inherited from, declaring it as `sealed` means that it cannot be. Not all object-oriented languages support this concept, but it can be useful. In C# the syntax looks like this:

```
sealed class FinalClass
{
...
```

C# also supports declaring an individual override method as `sealed`, preventing any further overrides of it.

The most likely situation when you'll mark a class or method as `sealed` will be if it is very much internal to the operation of the library, class, or other classes that you are writing, so you are fairly sure that any attempt to override some of its functionality causes problems. You might also mark a class or method as `sealed` for commercial reasons, in order to prevent a third party from extending your

classes in a manner that is contrary to the licensing agreements. In general, however, you should be careful about marking a class or member as `sealed`, since by doing so you are severely restricting how it can be used. Even if you don't think it would be useful to inherit from a class or override a particular member of it, it's still possible that at some point in the future someone will encounter a situation you hadn't anticipated in which it is useful to do so.

Interfaces

Earlier in this appendix, we indicated that there are two types of inheritance: implementation inheritance and interface inheritance. So far we've discussed implementation inheritance; in this section we are going to look more closely at interface inheritance.

In general, an interface is a contract that says that a class must implement certain features (usually methods and properties), but which doesn't specify any implementations of those methods and properties. Therefore you don't instantiate an interface; instead a class can declare that it *implements* one or more interfaces. In C#, as in most languages that support interfaces, this essentially means that the class inherits from the interface.

To get an idea of how an interface looks in programming terms, we'll show the syntax for the definition of an interface that is defined in the .NET base classes, `IEnumerator`, from the `System.Collections` namespace. `IEnumerator` looks like this:

```
interface IEnumerator
{
   // Properties
   object Current {get; }

   // Methods
   bool MoveNext();
   void Reset();
}
```

As you can see, the `IEnumerator` interface has two methods and one property. This interface is important in implementing collections and is designed to encapsulate the functionality of moving through the items in a collection. `MoveNext()` moves to the next item, `Reset()` returns to the first item, while `Current` retrieves a reference to the current item.

Beyond the lack of method implementations, the main point to note is the lack of any modifiers on the members. Interface members are always public and cannot be declared as virtual or static.

So why have interfaces? Up to now we've treated classes as having certain members and not concerned ourselves about grouping any members together—our classes have simply contained a list of various miscellaneous methods, fields, properties, and so on. There are often situations in which we need to know that the class implements certain features in order to be able to use a class in a certain way. An example is provided by the `foreach` loop in C#. In principle, it is possible to use `foreach` to iterate through a class instance, provided that that class is able to act as if it is a collection. How can the .NET runtime tell whether a class instance represents a collection? It queries the instance to find out whether it implements the `System.Collections.IEnumerable` interface. If it does, then the runtime uses the methods on this interface to iterate through the members of the collection. If it doesn't, then `foreach` will raise an exception.

You might wonder why in this case we don't just see if the class implements the required methods and properties. The answer is that that wouldn't be a very reliable way of checking. For example, you can probably think of all sorts of different reasons why a class might happen to implement a method called `MoveNext()` or `Reset()`, which don't have anything to do with collections. If the class declares that it implements the interfaces needed for collections, then you know that it really is a collection.

A second reason for using interfaces is for interoperability with COM. Before the advent of .NET, COM, and its later versions DCOM and COM+, provided the main way that applications could communicate with each other on the Windows platform, and the particular object model that COM used was heavily dependent on interfaces. Indeed, it was through COM that the concept of an interface first became commonly known. We should stress, however, that C# interfaces are not the same as COM interfaces. COM interfaces have very strict requirements, such as that they must use GUIDs as identifiers, which are not necessarily present in C# interfaces. However, using attributes (a C# feature that we cover in the beginning of the book), it is possible to dress up a C# interface so it acts like a COM interface, and hence provide compatibility with COM. We discuss COM interoperability in Chapter 28.

For more details on interfaces, see Chapter 4.

Construction and Disposal

For this final section of the appendix, we are going to leave inheritance behind, and look at another topic that is important in OOP programming: creation and disposal of objects—or to use the usual terminology, construction and destruction of objects. Say you have this code:

```
{
   int x;
   // more code
}
```

You will be aware that when `x` is created (comes into scope), memory gets allocated for it, and that when it goes out of scope, that memory is reclaimed by the system. If you are familiar with C#, you'll also be aware that `x` is initialized with the value zero when the variable comes into scope. For integers, the language defines what initializations happen automatically when an `int` gets created. But wouldn't it be nice if we could do the same for our own classes? Well, most modern OOP languages support the ability to do this—and C# is no exception. This support happens through something called a *constructor*. A constructor is a special method called automatically whenever an object of a given class is created. You don't have to write a constructor for a class, but if you want some custom initialization to take place automatically, you should place the relevant code in the constructor.

Similarly, OOP languages, including C#, support something called a *destructor*. A destructor is a method called automatically whenever an object is destroyed (the variable goes out of scope). Reclaiming memory aside, destructors are particularly useful for classes that represent a connection to a database, or an open file, or those that have methods to read from and write to the database/file. In that case, the destructor can be used to make sure that you don't leave any database connections or file handles hanging open when the object goes out of scope.

That said, the facilities offered by the .NET Framework and the garbage collector mean that destructors are not only used a lot less often in C# than they are in pre-.NET languages, but also that the syntax for

defining them is more complex (indeed, destructors are almost the only thing that is more complex to code in C# than in C++!). For that reason we won't look any more closely at destructors in this appendix. How to write destructors in C# is covered in Chapter 5. In this appendix we will concentrate on constructors, to give you an idea of how the concept works.

Visual Basic developers will note that there are some similarities between constructors and the `Initialize()` *and* `Form_Load()` *methods of VB class modules. Constructors, however, are far more flexible and powerful.*

Creating Constructors

When you see a constructor definition in C#, it looks much like a method definition, but the difference is that you don't usually call a constructor explicitly. It's like a method that is always called on your behalf whenever an instance of a class is created. In addition, because you never call the method explicitly, there is no way you can get access to any return value, which means that constructors never return anything. You can identify a constructor in a class definition because it always has the same name as the class itself. For example, if you have a class named `MyClass`, a skeleton constructor will be defined as follows:

```
public class MyClass
{
   public MyClass()
   {
   }
   ...
```

This constructor so far does nothing, because you haven't added any code to it. Let's add an integer field `MyField` to the class and initialize it to 10:

```
public class MyClass
{
   public MyClass()
   {

      myField = 10;

   }

   private int myField;

   ...
```

Notice that no return type is specified, not even `void`. The compiler recognizes the constructor from the fact that it has the same name as the containing class. You should note that one implication of this is that it is not possible to write a method that has the same name as the class it belongs to, because if you do the compiler will interpret it as a constructor.

From the previous example, you might wonder if we've actually achieved anything new. After all, in terms of C# syntax, we could have written:

```
public class MyClass

   private int myField = 10;
```

This achieves the same effect—specifying how to initialize each object without explicitly indicating a constructor. Indeed, we have already done something like this in all our `Authenticator` samples, in which we specified that the password field should automatically be initialized to an empty string. The answer is that here we are trying to introduce the concept of a constructor. The above code is really just C# shorthand for specifying construction code implicitly—a shorthand that is specific to C#. Behind this shorthand there is still a constructor at work. Besides, by writing a constructor explicitly, it means we can write code to compute initial values at runtime—the shorthand requires values to be known at compile time, as constants.

It's not necessary to provide a constructor for your class—we haven't supplied one for any of our examples so far. In general, if you don't explicitly supply any constructor, the compiler will just make up a default one for you behind the scenes. It'll be a very basic constructor that just initializes all the member fields to their normal default values (empty string for strings, zero for numeric data types, and `false` for `bool`s).

> *Initializing to default values is something that happens in C# because C# initializes all members of a class. If you are coding in a different language, this behavior might differ. For example, by default C++ never initializes anything unless you explicitly indicate that's what you want. So in C++, if you don't supply a constructor to a class, then its members won't get initialized to anything (unless they have constructors instead).*

Passing parameters to constructors

Let's go back to our `Authenticator` class. Say we wanted to modify the class so that we can specify the initial password when we first instantiate the class. It is possible to do this by supplying a constructor that takes parameters. In this regard, a constructor behaves like a method in that we can define whatever parameters we want for it, and this is where constructors really score over Visual Basic's `Initialize` or `Form_Load`.

For the `Authenticator`, we'd probably add a constructor that takes an initial password as a parameter:

```
public class Authenticator
{

   public Authenticator(string initialPassword)
   {
      password = initialPassword;
   }

   private string password = "";
   private static uint minPasswordLength = 6;
   ...
```

The advantage of using such a constructor is that it means an `Authenticator` object is guaranteed to be initialized the instant it is created. It is, therefore, not possible for other code to access the object before it has been initialized, as would be possible if we initialized it by calling a method after instantiating an object.

Now, to instantiate the object we would use a line of code similar to the following:

```
Authenticator NewUser = new Authenticator("MyPassword45");
```

Here we have created an instance with the password `MyPassword45`. You should note that the following line will not compile any more:

```
Authenticator NewUser2 = new Authenticator();
```

This is because we do not supply any parameters to the constructor, and the constructor requires one parameter. However, if we wanted to, we could simply create an overload for the constructor that didn't take any parameter arguments and simply set a default password in this constructor overload (this would not be a very secure approach though!).

More uses of constructors

Although the only thing we've done with constructors is to initialize the values of fields, a constructor does act as a normal method so you can place any instructions you want in it; for example, you might perform some calculations to work out the initial values of the fields. If your class encapsulates access to a file or database, the constructor might attempt to open the file. The only thing that you cannot do in a constructor is return any value (such as indicating status) to the calling code.

Another novel use is to use a constructor to count how many instances of a class have been created while the program is running. If we wanted to do that for the `Authenticator` class, we could create a static field, `nInstancesCreated`, and amend the code for the constructor as follows:

```
public class Authenticator
{

   private static uint nInstancesCreated = 0;


   public Authenticator(string initialPassword)
   {

      ++nInstancesCreated;

      Password = initialPassword;
   }

   private string password = 10;
   private static uint minPasswordLength = 6;
   ...
```

This example might not have many practical applications, but it demonstrates the kind of flexibility you have by being able to specify your own constructors. Counting instances is something you're unlikely to want to do in release builds of code, but it's something that you might want to do for debugging purposes.

Summary

The aim of this appendix has been to introduce you to the basic concepts of object-oriented design in C#, including:

- Classes, objects, and instances
- Fields, methods, and properties
- Overloading
- Inheritance and class hierarchies
- Polymorphism
- Interfaces

Object-oriented programming methodology is strongly reflected in the design of the C# language, and of Intermediate Language too which becomes apparent when you start using the .NET base classes. Microsoft has done this because with our current understanding of programming techniques, it simply is the most appropriate way of coding any large library or application.