Programmer to Programmer ™

Visual C# .NET:

A Guide for VB6 Developers

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

Brad Maiani, James Still, Angelo Kastroulis, Marco Bellinaso, Cristian Darie

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

What you need to use this book

This books assumes that you have either:

- **Visual Studio .NET** (Professional or higher), or
- **Visual C# .NET Standard Edition**

These require Windows NT 4.0 with Service Pack 6, Windows 2000, Windows XP Professional, or Windows .NET Server to be installed.

Note that there are some limitations to Visual C# .NET Standard Edition, which mean that not all of the project types used in this book can be easily generated with this edition. In particular, Chapter 8, *Creating Custom Controls*, requires the full edition of Visual Studio .NET.

We're also assuming you have access to either a SQL Server or MSDE database server. If you're using the Standard Edition of Visual C# .NET, you will need MSDE to connect to the database from Server Explorer. MSDE can be installed with the samples shipped with the .NET Framework SDK; this is itself installed with Visual C# .NET.

Chapter 11, *Integrating VB6 and C#*, requires access to a VB6 installation.

Summary of Contents

5

Object-Oriented Programming Part II

In the previous chapter we went through the basics of object-oriented programming with C#. We covered several new concepts. Now is the time to continue our journey into more advanced topics of OOP, and how it is implemented in C#.

More specifically, in this chapter we'll cover:

- Method and property hiding
- Method and property overriding
- Static class members and static constructors
- Interfaces and interface inheritance
- Delegates and events
- Constants and `readonly` fields
- Value types and reference types
- Passing value types by value and by reference
- Passing reference types by value and by reference
- Objects in .NET – `System.Object`
- Boxing and unboxing

All of these really stem from increasing our knowledge of classes and objects, and what we can do with them. Let's start by learning more about base classes and derived classes.

Base Classes and Derived Classes

In Chapter 4 we looked at how a class can be derived from another class. We saw how derived classes can add members – for example new methods and properties – while the members present in the original class continue to work in the same way.

We created a `Person` class, and derived from that `Instructor` and `Student` classes. We saw that we could treat `Instructor` and `Student` objects as if they were `Person` objects using a technique called polymorphism. This meant that we could call the `DescribeYourself` method on either, without worrying about exactly what kind of object we were dealing with.

However, in all our examples we demonstrated how a derived class can add functionality to the base class – but we know that inheritance, by its nature, allows us to *modify* the inherited behavior. In our previous examples, all our objects described themselves the same way – in a real-world application, we are likely to want an `Instructor` to describe itself in a different way from `Student` or from `Person`.

In this section we will see that there are two techniques that can be used to change the inherited behavior – they are called *hiding* and *overriding*. Now that you are curious enough, let's see some code!

Another Example of Polymorphism

Since the concepts of method hiding and overriding are directly related to inheritance and polymorphism, we'll first revisit polymorphism in a simple example, with our familiar `Student` and `Person` classes. Then we'll use this example as a base for the following demonstrations.

Change `Person.cs` to:

```
using System;

namespace PersonExample
{
  class Person
  {
    protected string name;

    public Person(string name)
    {
      this.name = name;
    }

    public void DescribeYourself()
    {
      Console.WriteLine
                    ("Hi there! My name is {0} and I am a nice person.", name);
    }
  }
}
```

Now let's change the `Student` class too. If you used a separate file, like `Student.cs`, modify it like this. Otherwise, just add the `Student` class to the `PersonExample` namespace in `Person.cs`:

```
namespace PersonExample
{
  class Student: Person
  {
    public Student(string name): base(name)
    {
    }
  }
}
```

Note that here we are not adding any new functionality to the `Student` class, and we're not changing any of the existing functionality – we will do that later. For now, `Student` and `Person` behave in exactly the same way. Now modify `MyExecutableClass.cs` to read as follows:

```
using System;

namespace PersonExample
{
  class MyExecutableClass
  {
    static void Main(string[] args)
    {
      Person dave = new Person("Dave");
      dave.DescribeYourself();

      Person claire = new Student("Claire");
      claire.DescribeYourself();

      Student mark = new Student("Mark");
      mark.DescribeYourself();
    }
  }
}
```

Type the code and make sure it works. When executed, the program should display these lines:

```
Hi there! My name is Dave and I am a nice person.
Hi there! My name is Claire and I am a nice person.
Hi there! My name is Mark and I am a nice person.
Press any key to continue
```

There are two things you may have already noticed about this program: first, the code is not very complicated, and second, the output is very boring. It is not particularly interesting to see `Person` and `Student` objects that describe themselves exactly the same way.

Another important thing to note about the code is how we implement polymorphism when creating the objects in `MyExecutableClass`. We create `dave` as `Person`, and ask him to describe himself like this:

```
Person dave = new Person("Dave");
dave.DescribeYourself();
```

Very straightforward. We do the same for Mark, except he's a `Student`:

```
Student mark = new Student("Mark");
mark.DescribeYourself();
```

If with the two boys things are very clear, our girl Claire is a bit more complicated – just like in real life:

```
Person claire = new Student("Claire");
claire.DescribeYourself();
```

Remember that polymorphism allows us to declare objects of a generic type, and use them to access objects of a more specialized type (that is, objects of a derived class). That's what we do now: `claire` is declared as a generic `Person`, but then we assign a `Student` object to it. Although this doesn't appear to be particularly interesting or useful right now, you'll soon understand why we declared and created `claire` like this.

Now let's get back to the main topic – changing behavior in the derived class.

Method and Property Hiding

There are two ways to change the inherited behavior in a derived class: by using hiding or overriding. The best way to understand the implications of each is by example. We'll look at hiding first. Note that both overriding and hiding work as well for properties as they do for methods. Most of the time we only override or hide methods, but keep in mind that the same techniques also apply for properties. In our examples, we'll only use methods.

Hiding and overriding are very similar. They are both ways to declare a method or property in the derived class that also exists in the base class. Of course, the method's or property's signature must be identical in the base class and in the derived class – otherwise it will be considered as adding a new method or property instead of modifying an existing one. Two signatures are considered to be 'identical' if they have the same parameters – the return value does not make a difference.

In our examples, we'll use `DescribeYourself()` to demonstrate these concepts. To demonstrate method hiding, add the following code to the `Student` class. Particularly note the `new` keyword:

```
class Student: Person
{
  public Student(string name): base(name)  {}

  public new void DescribeYourself()
  {
    Console.WriteLine
      ("Hello man, I'm {0}. I'm a cool student.", name);
  }
}
```

Now when we run the program we will see the following:

```
Hi there! My name is Dave and I am a nice person.
Hi there! My name is Claire and I am a nice person.
Hello man, I'm Mark. I'm a cool student.
Press any key to continue
```

Before analyzing the results, let's remind ourselves once again how we defined our three objects:

```
Person dave = new Person("Dave");
Person claire = new Student("Claire");
Student mark = new Student("Mark");
```

`dave`'s behavior is easy to understand: `dave` is a `Person`, so any changes in `Student` doesn't affect it in any way. `mark` is a `Student` object, and will naturally use `Student`'s implementation of `DescribeYourself()`.

With Claire we have a problem. Although she is a `Student`, she describes herself as a `Person`. Let's understand now why this is.

`claire`, which holds a `Person` reference, doesn't see the new implementation of `DescribeYourself()` from `Student`, even if its underlying object is actually `Student`. When we use hiding, the `new` method in the derived class can't be seen using an object that has initially been declared as `Person` – even if we assign a `Student` object to it.

This has implications for methods that use polymorphism. In the previous chapter we added a `Describe()` method to `MyExecutableClass`, which took a `Person` as a parameter. We saw that we could pass a `Student` into this method, and it would still work. But if we passed a `Student` into it now, we would get a `Person` description instead of a `Student` description. Usually this is not the behavior we want – we would want the `Student` description. Fortunately, overriding offers us a way to do this. Let's look at overriding now.

Method and Property Overriding

Method overriding will solve the problem that we saw using hiding in the previous example. However, for this to work we need to modify our `Person` class. Modify the `DescribeYourself()` method of the `Person` class by adding the `virtual` keyword to its definition:

```
public virtual void DescribeYourself()
```

And now, in the `Student` class, replace the `new` keyword with `override`:

```
public override void DescribeYourself()
{
  Console.WriteLine
    ("Hello man, I'm {0}. I'm a cool student.", name);
}
```

Now we will get the following output:

```
Hi there! My name is Dave and I am a nice person.
Hello man, I'm Claire. I'm a cool student.
Hello man, I'm Mark. I'm a cool student.
Press any key to continue
```

Now calling `DescribeYourself()` on `claire` displays the student greeting. We still declared `claire` as a `Person`, and then assigned a `Student` object to it. But this time, `DescribeYourself()` was not hidden in the derived class – instead, in was overridden. When a method is overridden, the compiler will always look for its most derived implementation. In the case of `claire`, the most derived implementation is the one in `Student`. This demonstrates the difference between overriding and hiding.

In order to be able to override a method in a derived class, it *must* be declared as `virtual`, because in C# methods and properties are not `virtual` by default. If we don't explicitly mark a base class's member as `virtual`, we cannot override it in the derived class. If you think that certain members of your class should or could be overridden instead of hidden in derived classes, make sure you mark them as `virtual`.

Keep in mind that just because you declare a method as `virtual` does not mean it will always be overridden in the derived class. By declaring a class member as `virtual`, the derived classes will have a choice to either `override` the member (using `override`), to hide the member (using `new`), or to use the original implementation. If the method is not `virtual`, it cannot be overriden. To test method hiding for virtual methods, modify the `Student` class again by using `new` instead of `override`:

```
public new void DescribeYourself()
{
  Console.WriteLine
    ("Hello man, I'm {0}. I'm a cool student.", name);
}
```

Now, Claire will describe herself again as a person, demonstrating that `DescribeYourself()` was hidden and not overridden.

Because there are three ways we can define the method in the derived class (with `new`, `override`, or neither), and two ways we can have our method in the base class (virtual or non-virtual), you may be a bit confused about the combinations that can be made. The following table should make all options clear.

	Base class's method is `virtual`	**Base class's method is not `virtual`**
Derived class's method is marked with `new` modifier	The method is *hidden* in the derived class	The method is *hidden* in the derived class
Derived class's method is marked with `override` modifier	The method is *overridden* in the derived class	An error is generated by the compiler – you can't override a non-virtual method

	Base class's method is `virtual`	**Base class's method is not `virtual`**
There is no special modifier for derived class's method	The method is hidden by default, but a warning is generated because the `new` modifier is not present	The method is hidden by default, but a warning is generated because the `new` modifier is not present

There are four combinations that lead to method hiding (because this is the default behavior), and only one for method overriding (C# forces us to be explicit when we want to use overriding).

> **Many people confuse overloading with overriding. They usually know the difference, but say the wrong thing without thinking. Remember that overloading refers to the situation when we have different methods with the same name in the same class, and overriding is when we have different methods with the same name *and* with the same signature, in the base class and derived classes. You may find that some publications – particularly web sites – will get this wrong. However, it's usually easy to tell what is meant by the context.**

Even More Polymorphism

In our first example when we demonstrated polymorphism, we used a `Describe()` method in `MyExecutableClass` that had a parameter of type `Person`. That method called `DescribeYourself()` on the object received as a parameter. Let's revisit this method now, and use it to really hammer home our understanding of overriding, hiding, and polymorphism.

Modify `MyExecutableClass.cs` to read as shown:

```
using System;

namespace PersonExample
{
  class MyExecutableClass
  {
    static void Main(string[] args)
    {
      Person dave = new Person("Dave");
      Person claire = new Student("Claire");

      Describe (dave);
      Describe (claire);
    }

    static void Describe(Person person)
    {
      person.DescribeYourself();
    }
  }
}
```

Now, when we use method hiding (with `new`) for `DescribeYourself()`, we obtain this output:

```
Hi there! My name is Dave and I am a nice person.
Hi there! My name is Claire and I am a nice person.
Press any key to continue
```

If we use method overriding (with `override`) instead, the output changes to:

```
Hi there! My name is Dave and I am a nice person.
Hello man, I'm Claire. I'm a cool student.
Press any key to continue
```

Before we leave polymorphism, let's look at how we could modify the `Describe()` method so that it still accepts any `Person` as a parameter, but can do different things depending on whether we pass a `Person`, `Student`, or `Instructor`.

Polymorphism and Type Casting

In the previous example we demonstrated polymorphism using the `Describe()` method:

```
static void Describe(Person person)
{
   person.DescribeYourself();
}
```

We saw that if we send a `Student` as a parameter, `DescribeYourself()` behaved differently, depending whether it was implemented to hide or override the base class method.

In real-world examples, it is very likely that a derived class like `Student` will not only hide or override existing members, but also add new members of its own. Will we be able to call those extra members in `Describe()` method, using a `Person` reference?

This time, the answer is no. If `Student` defined a method named `Learn()`, that wasn't also declared in `Person`, we couldn't do this:

```
Person person = new Student("Joey");
person.Learn();
```

Actually this makes sense – we can't call methods specific to `Student` on `Person` objects! We have to make an explicit *cast* to the `Student` data type before we can call methods specific to that class. To see this in action, add the following method to the `Student` class:

```
public void Learn()
{
  Console.WriteLine
     ("Student wakes up, yawns, scratches head, and tries to look interested.");
}
```

For this example we will also need an `Instructor` class. If you don't have one already from the previous chapter, add a new class – called `Instructor` – to the project and modify it to read like this:

```
using System;

namespace PersonExample
{
  class Instructor : Person
  {
    public Instructor (string name) : base (name) {}

    public override void DescribeYourself()
    {
      Console.WriteLine
        ("Good morning class. My name is {0}, and I am your instructor.", name);
    }

    public void Teach()
    {
      Console.WriteLine("Instructor babbles incomprehensibly.");
    }
  }
}
```

Besides `DescribeYourself()`, which is overridden in both classes (so make sure it is `virtual` in the `Person` class), `Student` now has a method called `Learn`, and `Instructor` has a method called `Teach`. We want to improve the `Describe()` method like this:

- If the object received as a parameter is a `Student`, we want to call `DescribeYourself()` on it, and then `Learn()`
- If the object received as a parameter is an `Instructor`, we want to call `DescribeYourself()`, and then `Teach()`
- If the object received is a simple `Person`, we want to call `DescribeYourself()` and nothing else

`Describe` should look something like this:

```
    static void Describe(Person person)
    {
       person.DescribeYourself();
       // if person is a Student, call Learn()
       // if person is an Instructor, call Teach()
    }
```

How do we implement this functionality? The problem is that our `person` parameter is of type `Person`, so we can't call `Learn` or `Teach` directly on it – the compiler wouldn't recognize these methods.

Let's see the solution first, and then we'll explain how it works. Update `MyExecutableClass` like this:

```
class MyExecutableClass
{
  static void Main(string[] args)
  {
    Person person = new Person("Person");
    Student student = new Student("Student");
    Instructor instructor = new Instructor("Mr. Instructor");

    Describe(person);
    Describe(student);
    Describe(instructor);
  }

  static void Describe(Person person)
  {
    person.DescribeYourself();

    if (person is Student)
    {
      Student s = (Student) person;
      s.Learn();
    }

    if (person is Instructor)
    {
      Instructor i = (Instructor) person;
      i.Teach();
    }
  }
}
```

This is the output:

```
Hi there! My name is Person and I am a nice person.
Hello man, I'm Student. I'm a cool student.
Student wakes up, yawns, scratches head, and tries to look interested.
Good morning class. My name is Mr. Instructor, and I am your instructor.
Instructor babbles incomprehensibly.
Press any key to continue
```

In order to achieve the desired results, we used the `is` operator to test if a certain expression is compatible with a specific type:

```
    if (person is Student)
    {
```

If we find out that `person` is a `Student`, then it's safe to make an explicit cast to `Student`. Once we save `person` using a `Student` reference we are free to call the `Learn()` method, which is specific to `Student`:

```
            Student s = (Student) person;
            s.Learn();
```

Alternatively, we could use the `as` operator to make the cast:

```
            Student s = AnyPerson as Student;
            s.Learn();
```

Making conversions using the `as` operator is more convenient in some cases because if the types are not compatible, it returns `null` instead of raising an exception. In our example, we tested whether the object received as a parameter is compatible with the type we want to cast to using the `if` statement, so here it is just a matter of taste what syntax you prefer to use for casting.

We also did the same magic for the `Instructor` class, as we've seen in the above code snippet.

Accessing Members of the Base Class

We are free to access any public or protected members of the base class from the body of the derived class – just as we did in the `DescribeYourself()` method in `Student` and `Instructor`, where we accessed the `name` protected member of `Person`.

Although most of the time it's OK to reference base class members just using their names, sometimes we have a member with the same name in the current (derived) class. In this situation, we must be specific about what member we want to access – if we want the one of the base class, we use the `base` keyword; if we want the one in the current class, we use `this` (although if we use neither, `this` is the default).

Let's first look at a short example that demonstrates using `base` and `this` for class fields. Then we'll do a quick upgrade to the `Instructor` class, to demonstrate the same idea for methods. Here's the example with fields:

```
class BaseClass
{
   public int Value = 5;
}

class DerivedClass: BaseClass
{
   public int Value = 10;

   public void WriteValues()
   {
      Console.WriteLine("Value of BaseClass: " + base.Value);
      Console.WriteLine("Value of DerivedClass: " + this.Value);
   }
}
```

If we create an object of `DerivedClass` type and execute its `WriteValues()` method, the output is:

```
Value of BaseClass: 5
Value of DerivedClass: 10
Press any key to continue
```

Note that we didn't have to use `this`, but `this` makes the difference between the `base` value and the `this` value obvious. The program works the same if we use `Value` instead of `this.Value`. The example demonstrates using `base` and `this` for fields, but they work the same way for methods or properties.

For a more interesting example, let's modify the `DescribeYourself()` method of `Instructor` like this:

```
public override void DescribeYourself()
{
   base.DescribeYourself();
   Console.WriteLine
      ("Good morning class. My name is {0}, and I am your instructor.",name);
}
```

In order to test this, we need only two lines in the `Main` method of `MyExecutableClass`. Don't remove the `Describe` method, because we will use it in the following examples:

```
static void Main(string[] args)
{
  Instructor instructor = new Instructor("Mr. Instructor");
  instructor.DescribeYourself();
}
```

When executed, the output will be:

```
Hi there! My name is Mr. Instructor and I am a nice person.
Good morning class. My name is Mr. Instructor, and I am your instructor.
Press any key to continue
```

Abstract Classes, Methods, and Properties

We have played with some very interesting topics so far. We learned about how to use method overriding and hiding in our class hierarchies. We learned how important is to design a class well, even if it takes a lot of time, in order to make our life easier as the project extends.

These concepts for working with base classes and derived classes introduce an interesting idea – base classes that can *only* be used as base classes. We've been working with `Person`, `Student`, and `Instructor`. We've allowed `Person` objects to exist – but in many applications we might not want to allow basic `Person` objects – even though it makes sense to have a `Person` class from which other classes derive. We might want to force every person to be either a student, or an instructor, or some other subtype. In this case, we would make `Person` an *abstract* class. Let's turn our attention to look at abstract classes now – later we will look at abstract methods and properties.

Abstract Classes

Abstract classes are classes that cannot be directly instantiated, but can be inherited from. When talking about inheritance in the previous chapter, we mentioned that derived classes are more specialized versions of the base class. Indeed, we can see that a student is a "more specialized" person, and the same for an instructor. Applying the same idea the reverse way, it's fair to say that the base class serves as a generalized version of its derived classes. You will sometimes find that in your class hierarchies there will be classes that are so general, that it wouldn't make much sense to directly instantiate them. In regard to our `Person` example, say that we want to create an application for a school that must keep track of all its students, teachers, secretaries, managers, and so on. We decide to use `Person` as a base class to create the other classes like `Student`, `Instructor`, `Teacher` or `Secretary`. While it does make sense to create objects of type `Student` or `Instructor`, we decide that nobody should ever *create* objects of type `Person`.

Abstract classes provide us with the means to achieve this behavior. If we declare `Person` as abstract, we will not be able to instantiate it, but we will be free to derive from it, thus ensuring that we'll have a set of classes that are guaranteed to support a set of functionality – the functionality inherited from `Person`. Before going into other details, let's first see how to implement an abstract class. Modify our `Person` class by adding `abstract` to its definition:

```
    abstract class Person
```

Now compile and execute the program, using the same code we used in the previous section, *Accessing Members of the Base Class*. If everything works well, the output should be the same.

An important point to understand about abstract classes is that we aren't allowed to instantiate them, but we can still declare objects of their type. Moreover, we can assign to them created objects of derived classes, thus enabling us to use polymorphism on them. To demonstrate this, let's revisit polymorphism using the following code in the `Main` method of `MyExecutableClass`:

```
static void Main(string[] args)
{
  Person person;

  Instructor instructor = new Instructor("Mr. Bean");
  Student student = new Student("Joey");

  person = instructor;
  person.DescribeYourself();

  person = student;
  person.DescribeYourself();
}
```

Run the program, and if you have the version of `Student` and `Instructor` as presented in the previous examples, you should see the following output:

```
Hi there! My name is Mr. Bean and I am a nice person.
Good morning class. My name is Mr. Bean, and I am your instructor.
Hello man, I'm Joey. I'm a cool student.
Press any key to continue
```

We were allowed to declare a `Person` object:

```
    Person person;
```

If we had tried to instantiate the object with the `new` keyword, we would have received the following compile-time error: Cannot create an instance of the abstract class or interface 'PersonExample.Person'.

Because the `Instructor` and `Student` classes are not abstract, we are allowed to declare and instantiate them, as `instructor` and `student` in our example. Then we are able to assign objects of derived classes to the `Person` object:

```
person = instructor;
person.DescribeYourself();
```

The result of this code snippet depends of course of whether `DescribeYourself()` is overridden or hidden in `Instructor`. Alternatively it might not be defined at all in `Instructor`, in which case the `Person` implementation of the method will be used.

Also, keep in mind that you can still use the `Describe()` method we used in our previous examples. This method takes a `Person` parameter. We used to send `Person`, `Student` and `Instructor` objects to this method. Now that `Person` is abstract, we can only send `Student` and `Instructor` objects, or objects of other classes derived from `Person`. Obviously we can't send `Person` objects, since we can't create `Person` objects.

Abstract Methods and Properties

An abstract class is allowed to incorporate abstract methods and properties. As with overriding and hiding, we will demonstrate the concepts using abstract methods, but the same rules apply for properties too.

Before we learn what we can do with them, let's first see what they look like. An abstract method is a method without an implementation – we provide only the signature, like in the following example:

```
public abstract void DescribeYourself();
```

An abstract property definition looks like this:

```
public abstract int MyProperty
{
   get;
   set;
}
```

Abstract class members can only be part of abstract classes. Since they don't have an implementation, they must be overridden in the derived classes, except for derived classes that are also abstract. In order to understand their use better, let's imagine again the scenario where we create an application for a school.

After the initial design of the School application, we decide that a great number of classes – like `Student`, `Instructor`, or `Secretary` – have many things in common. For example, each of them must have a name, and must know how to describe itself.

You have already learned that inheritance is a great way to implement code reuse. If we have a great deal of code that all our classes share, then it might be good to place that code in the base class (`Person` in our case). Still, there may be times when we don't care too much about code reuse, but from a design point of view we want to ensure that a number of classes (`Student`, `Instructor`, and so on) have a similar behavior, so we can benefit from polymorphism by having a base class from which they all inherit.

By using an abstract base class using abstract members, we can provide a pattern to be followed by other classes. Classes that derive from an abstract class must provide an implementation for every abstract member of that class. In order to understand this better, let's use an example. Modify `Person` like this:

```
abstract class Person
{
  protected string name;

  public Person(string name)
  {
    this.name = name;
  }

  public abstract void DescribeYourself();
}
```

Note that the constructor is still there, and it does it job of storing the name to a private instance field. This functionality is still inherited by derived classes. But the major change is in the way `DescribeYourself()` is defined: notice the `abstract` modifier, and the fact that the method has no implementation.

Now, all non-abstract derived classes will *have to* override `DescribeYourself()`. In a previous exercise, we called the base class's `DescribeYourself()` method from `Instructor`, in a line like this:

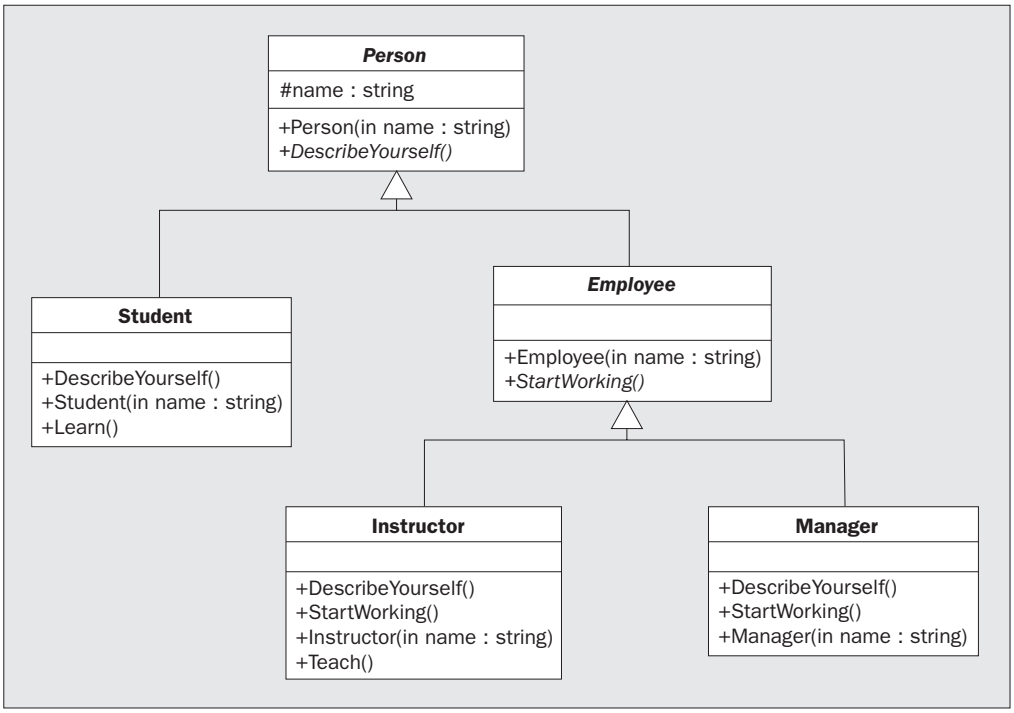
```
base.DescribeYourself();
```

If you still have that line, you need to remove it, because it's not possible to call an abstract method. This is obvious, since an abstract member doesn't have an implementation. Because both `Student` and `Instructor` override `DescribeYourself()` in their current implementation, the program should still run and produce the same results:

```
Good morning class. My name is Mr. Bean, and I am your instructor.
Hello man, I'm Joey. I'm a cool student.
Press any key to continue
```

Abstract in Action

In more complicated examples, you'll often find abstract classes that derive from other abstract classes. An abstract class that derives from another abstract class can override any number of abstract members of the base class, but finally there has to be a non-abstract, derived class, otherwise a hierarchy with only abstract classes wouldn't be too useful. To demonstrate this idea, let's complicate our example a little bit more. Take a look at this diagram, which we will implement after discussing it:



Visio shows abstract classes or abstract class members using italics. `Person`, the base class, contains a constructor, a protected class field (`name`) and an abstract method – `DescribeYourself()`. Because `DescribeYourself()` is abstract, it is overridden in the non-abstract class student `Student`. Even though `Instructor` and `Manager` are not directly derived from `Person`, they must also override `DescribeYourself()` because it is not overridden in any of the intermediary classes (`Employee` in this case).

`Employee` is an abstract class, so it can't be directly instantiated. It contains an abstract method named `StartWorking()`, which is going to be overridden in all its derived classes. Because `Person` has a parameterized constructor and no parameterless constructors, `Employee` must also provide a parameterized constructor that simply passes the argument to `Person`'s constructor – if you don't understand why this is required, please read the section about constructors in the previous chapter again. Also, if you look carefully, each derived class in the hierarchy provides a parameterized constructor, which in turn calls its base class's constructor. You can easily identify constructors in the figure because they are methods with the same name as the class.

Let's see the implementation right now. Because the code is quite long, I've written all classes under the same namespace in the same file.

```
using System;

namespace PersonExample
{
  abstract class Person
  {
    protected string name;

    public Person(string name)
    {
      this.name = name;
    }

    public abstract void DescribeYourself();
  }

  class Student: Person
  {
    public Student(string name): base(name)  {}

    public override void DescribeYourself()
    {
      Console.WriteLine
        ("Hello man, I'm {0}. I'm a cool student.", name);
    }

    public void Learn()
    {
      Console.WriteLine
        ("Student wakes up, yawns, scratches head, and tries " +
         "to look interested.");
    }
  }

  abstract class Employee: Person
  {
    public Employee (string name) : base (name) {}

    public abstract void StartWorking();
  }
```

```
    class Instructor: Employee
    {
      public Instructor (string name) : base (name) {}

      public override void DescribeYourself()
      {
        Console.WriteLine
          ("Good morning class. My name is {0}, and I am your instructor.",name);
      }

      public override void StartWorking()
      {
        Console.WriteLine("Going to my dear students...");
      }

      public void Teach()
      {
        Console.WriteLine("Instructor babbles incomprehensibly.");
      }
    }

    class Manager: Employee
    {
      public Manager(string name): base(name){}

      public override void DescribeYourself()
      {
        Console.WriteLine("I'm {0}. I'm the boss around here.", name);
      }

      public override void StartWorking()
      {
        Console.WriteLine("No, it's my golf day.");
      }
    }
  }
```

For a quick test of these classes, implement the `Main()` method of `MyExecutableClass` as follows:

```
      static void Main(string[] args)
      {
        Person person;
        Employee employee;

        Student student = new Student("Joey");
        Instructor instructor = new Instructor("Mr. Bean");
        Manager manager = new Manager("Mr. Listentomecarefully");

        // apply polymorphism using Person reference
        person = student;
        person.DescribeYourself();
```

```
        person = instructor;
        person.DescribeYourself();

        person = manager;
        person.DescribeYourself();

        // apply polymorphism using Employee reference
        employee = instructor;
        employee.StartWorking();

        employee = manager;
        employee.StartWorking();
      }
```

Execute the code, and expect to obtain this result:

```
Hello man, I'm Joey. I'm a cool student.
Good morning class. My name is Mr. Bean, and I am your instructor.
I'm Mr. Listentomecarefully. I'm the boss around here.
Going to my dear students...
No, it's my golf day.
Press any key to continue
```

As we're at the end of this section, let's summarize the key ideas you should have learned about abstract members in C#:

- Abstract methods and abstract properties aren't allowed to have implementations.
- Abstract classes can't be instantiated. They can only be used as base classes for other classes.
- An abstract class can have both abstract and non-abstract members.
- If a class has at least one abstract member, it must also be declared as abstract – in other words, non-abstract classes aren't allowed to contain abstract members.
- Even if we have a full working class, we can declare it as abstract if we want to make sure nobody will use it directly (we force other programmers to inherit from our class).
- A non-abstract class that derives from an abstract class must override every abstract member inherited from the base class.
- Abstract class members are very similar to virtual class members in that they can be overridden in derived classes. However, abstract properties and method are not allowed to have implementations, and they *must* be *overridden* in the non-abstract derived class – we are not allowed to hide an abstract member.

Now lets look at the opposite of `abstract` – `sealed`.

Sealed Classes, Methods, and Properties

The `sealed` modifier, like `abstract`, can be applied both to classes and to class members. We use `sealed` when we want to make sure that a method or property will not be further overridden or hidden in derived classes, or that a class is not inherited from.

We are allowed to mark a class member as sealed only if it is already overriding a member of a base class. When we write a new member in a class and we don't want it to be overridden in derived classes, we simply don't mark it as virtual. But for an overriding member (that is, a member marked with the `override` keyword), the default behavior is that it can be overridden, even if it doesn't have the `virtual` keyword. In fact, it is not allowed to have both `virtual` and `override` keywords on the same member – because the `virtual` would be redundant. So if we want to make sure that an overriding member cannot be further overridden in derived classes, the solution is to use the `sealed` keyword.

Sealed classes cannot be inherited from. Therefore we are not allowed to have `virtual` or `abstract` methods or properties in a `sealed` class.

The purpose of the `sealed` modifier is clear: you'll mark a class as `sealed` when you must be sure nobody inherits from it, and the reasons for this are many – most probably, it's a internal class you're using in your project, and nobody should inherit from it. Also, if you're delivering a set of classes and you want to limit the way they are used (for commercial reasons, or to provide fewer unique entry points to your solution), you'll most probably want to mark many of them as sealed. Many of the classes provided by the .NET are sealed, the most noticeable probably being `String`.

Before ending this section, let's see how to apply the `sealed` keyword. If you want to make a short test, change `Person`'s definition like this:

```
sealed class Person
```

Now, when you try to compile the solution, you'll receive many compiling errors mentioning that one class or another cannot inherit from sealed class 'PersonExample.Person'.

Remember that only overriding methods can be sealed. To test `sealed` on a method, we can modify the `Student` class's `DescribeYourself()` as:

```
public sealed override void DescribeYourself()
```

So far, all of our discussion has assumed that methods, properties, and fields refer to a particular object. However, the world doesn't always work this way. Often we want a particular piece of functionality that does not neatly apply to a particular object. In the next section, we are going to discuss **static** members, and how to use them.

Interfaces and Interface Inheritance

There are two important concepts that we'll learn about in this section: **interfaces** and **interface inheritance**. These are features supported by all modern object-oriented programming languages, and it is important to learn what they are and how we can use them in our applications. Although these two notions are very tightly related, still they are different concepts and we'll learn about both of them separately. After learning the basic concepts, we'll upgrade our `Person` example to use interfaces.

Interfaces

An interface is set of property and method signatures, and serves as a **contract** for classes that implement it. When a class implements an interface, it guarantees that it will implement every signature defined in that interface.

Technically, an interface is very much like an abstract class, so let's first review what we know about abstract classes. In the previous sections we learned that an abstract class cannot be instantiated – it is meant to be inherited from, and serve as a base class for a number of classes. In our examples, we created the `Person` and `Employee` classes. Neither of them could be instantiated, because they were abstract. `Person` had an abstract member named `DescribeYourself()` that had to be overridden in derived classes, but also had a constructor (implementation), which instantiated the `name` protected field. `Employee` only had an abstract method signature called `StartWorking()`.

Let's see now the main similarities and differences between interfaces and abstract classes:

- Neither interfaces nor abstract classes can be instantiated. Still, we can create variables of their types, and then assign to these variables objects of derived classes.
- Both interfaces and abstract classes can contain method and property definitions. In fact, interfaces are allowed to contain *only* definitions, unlike abstract classes which are also allowed to contain method implementations. Additionally, interfaces aren't allowed to contain fields or constants, constructors or destructors (we'll cover these a bit later), and static members.
- Because C# supports only single implementation inheritance, a class cannot derive from more than one (abstract or non-abstract) class. Still, a class can implement any number of interfaces. We'll learn soon what this means. While we're here, note that the terminology is different with interfaces: a class *inherits from* another class, but it *implements* a number of interfaces.

Because the purpose of interfaces is to enable interface inheritance, let's see what it's all about.

Interface Inheritance

From a design standpoint, the power of abstract classes is that by preventing us from using them directly, they help enforce certain rules about how the classes in our class hierarchy can be used. In the latest version of our program, `Person` and `Employee` were abstract classes. We could not implement them, but they defined a set of methods that had to be overridden in all derived classes.

Interfaces provide an alternative way to enforce certain rules for classes through interface inheritance. Note that when we simply say "inheritance", this usually refers to implementation inheritance. This is the only kind of inheritance we have used so far in this chapter, and when I simply say "inheritance", implementation inheritance is what I have in mind.

In order to understand interface inheritance, let's first quickly analyze (once again) implementation inheritance, in a different way from what we have done before. Implementation inheritance achieves two important goals that you rely on when designing your application:

- **Derived classes are guaranteed to support all functionality exposed by the base class's default interface.** The *default interface* consists of all the public members of a class. In other words, if the base class has a public method named `DescribeYourself()`, we can safely call `DescribeYourself()` on any of the derived classes. We are guaranteed that any derived class will expose `DescribeYourself()`, regardless of whether it overrides the method, hides it, or simply relies on the base class's implementation. As a side note, this idea is at the heart of polymorphism.
- **Derived classes inherit all the implementations provided by the base class.** A base class can have abstract members, for which we need to provide an implementation in the derived classes. But if a public member of a base class *has* an implementation (so it's not abstract), we are not required to provide an implementation in the derived class.

It is very important to realize the difference between the two points I highlighted here.

- With implementation inheritance, we inherit not only the interface, but also the implementation.
- With interface inheritance, we only inherit the interface – all the interface is allowed to have is signatures with no implementations. Thus with interface inheritance we can't inherit any implementations.

At the first sight, interface inheritance is not very useful – it requires us to write code for every interface signature in each class that implements the interface. With interface inheritance, it would seem that all the benefits of code reuse provided by implementation inheritance are no longer with us. So you may ask – why do we need interface inheritance?

Well, there might be times when that's exactly what you'll want to do: to force all derived classes to implement all inherited behavior for themselves, as you don't want to provide any "base" implementation. But that's what you could achieve with implementation inheritance, using abstract classes that contained exclusively abstract members.

There is a subtle, but very important difference between inheriting from abstract base classes, and inheriting from interfaces. Remember that implementation inheritance could be verified with the "is a" rule. A person *is a* student, an instructor *is an* employee. This was true because a class could only derive from exactly one other class. With interface inheritance, a class can implement more interfaces, while still being able to derive from a base class – interfaces are "smaller" pieces of "functionality requirements" that we apply to a class in order to make sure the class implements all the interface's members. Inheritance relationship established by using interfaces cannot usually be verified with the "is a" rule; instead, "acts as" is perhaps better suited for this.

Another important difference is the possibility to implement multiple interfaces. With implementation inheritance, you know that you can only derive a class from one base class. This can be limiting sometimes, especially when you want a class to support multiple functionalities – although this sometimes leads to long inheritance chains and complicated design. With interfaces, since they are not inherited from, but implemented by other classes, it's possible for a single class to implement multiple interfaces.

In order to explain the concepts better, it's time to write some code.

Applying the Concepts

We'll start by looking at our last example, which we created in the *Abstract Classes, Methods, and Properties* section. The design goals were to have a set of classes – `Student`, `Instructor`, and `Manager` that had some similar behavior.

For example, all of them needed to describe themselves via a method named `DescribeYourself()`. Also, they all needed to have a name. After analyzing the requirements, we decided to add an abstract class named `Person`, which contained a method named `DescribeYourself()`, and a field called `name`. All the other classes derive from `Person`, thus ensuring that all of them have a name, and can describe themselves.

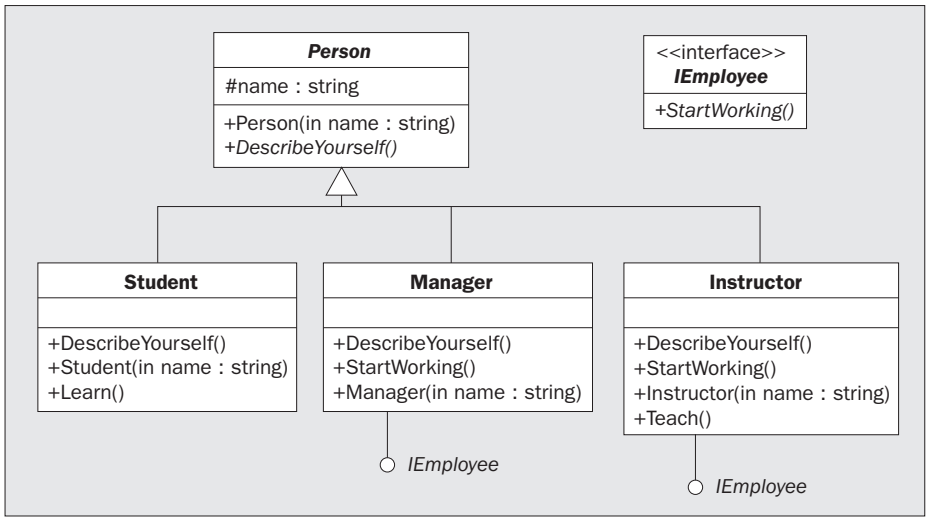
After some further analysis, we realized that `Manager` and `Instructor` had yet another thing in common – they were both employees, so they both needed to have a method named `StartWorking()`. We decided to add another abstract class, `Employee`, derived from `Person`, and we used that class as a base class for both `Instructor` and `Manager`. `Employee` has a `StartWorking()` method, so now we know that `Instructor` and `Manager` both have this method as part of their default interface.

Observe that we needed two abstract classes – `Person` and `Employee` – to ensure the derived classes provide the required functionality. `Person` did have some implemented functionality that we reused in derived classes – more specifically, the constructor. `Employee` only contained the abstract method `StartWorking()`.

In our current example, we decide that we don't need the abstract class `Employee`. We want all the classes – `Student`, `Instructor`, and `Manager` to derive from `Person`. Instead of inheriting from `Employee`, `Instructor` and `Manager` will implement the `IEmployee` interface, which contains the `StartWorking()` method. After we do this, we'll go even further by adding another interface to our example – but let's not spoil the surprise.

Using Interface IEmployee instead of Abstract Class Employee

Here's what we want to achieve. If you want to make a comparison, look a few pages back to the *Abstract in Action* section and see the old design of these classes.



As I warned you, we won't be using `Employee` any more. `Student`, `Manager`, and `Instructor` derive directly from `Person`. `Manager` and `Instructor` implement the `IEmployee` interface. Before proceeding, please make sure you have the code we used in the *Abstract Classes, Methods, and Properties* section.

Since there are more steps that need to be taken in order to upgrade the existing program, I'll mark them so they are easier to follow:

1. No matter how painful this might be, remove the abstract `Employee` class.

2. Add the `IEmployee` interface. For more clarity, I suggest you to add it at the beginning of the file. Interfaces, like base classes, are generally situated before other derived classes.

```
interface IEmployee
{
   void StartWorking();
}
```

3. Modify the `Instructor` class like this:

```
class Instructor: Person, IEmployee
{
   public Instructor (string name) : base (name) {}

   public override void DescribeYourself()
   {
      Console.WriteLine
```

```
            ("Good morning class. My name is {0}, and I am your instructor.",
             name);
      }

      public void StartWorking()
      {
         Console.WriteLine("Going to my dear students...");
      }

      public void Teach()
      {
         Console.WriteLine("Instructor babbles incomprehensibly.");
      }
   }
```

4. Modify `Manager` like this:

```
   class Manager: Person, IEmployee
   {
      public Manager(string name): base(name){}

      public override void DescribeYourself()
      {
         Console.WriteLine("I'm {0}. I'm the boss around here.", name);
      }

      public void StartWorking()
      {
         Console.WriteLine("No, it's my golf day.");
      }
   }
```

5. In the `Main()` method of `MyExecutable` class, declare `employee` as `IEmployee` instead of `Employee`:

```
   IEmployee employee;
```

Now execute the program. The output should be the same as the last time we ran the program.

```
Hello man, I'm Joey. I'm a cool student.
Good morning class. My name is Mr. Bean, and I am your instructor.
I'm Mr. Listentomecarefully. I'm the boss around here.
Going to my dear students...
No, it's my golf day.
Press any key to continue
```

Probably the most important change we made to the code was the addition of `IEmployee`:

```
interface IEmployee
{
   void StartWorking();
}
```

The syntax to declare an interface is simple, using the `interface` keyword. The interface members look like abstract members, because they don't have an implementation. Still, they aren't allowed to be explicitly declared as `abstract` – in fact, we can only specify a return value. They are automatically `public`, but we can't use the `public` (or any other) modifier with them.

Now that we have seen how to declare an interface, let's see how to actually implement it in a class:

```
class Instructor: Person, IEmployee
```

This line specifies that `Instructor` derives from `Person`, and implements `IEmployee`. We don't have to specify a class to derive from before specifying the interfaces: in fact, `Person` and `IEmployee` could both be interfaces, but we know that in our case `Person` is a class. Anyway, if there's a class we inherit from, it must be specified first, before any interfaces.

Then we defined the `StartWorking()` method.

```
public void StartWorking()
```

Notice that compared to its previous version, we removed the `override` keyword from it – this is required, since we aren't overriding any method. Also note that, because we're implementing `IEmployee`, we need to provide an implementation of `StartWorking()` in `Instructor` and `Manager` – if we didn't implement this method, we would break the contract we agree to when we said we would implement the interface. If you remove `StartWorking()` from the `Manager` class, you'll receive an error message specifying that 'PersonExample.Manager' does not implement interface member 'PersonExample.IEmployee.StartWorking()'.

After we successfully implement the interfaces, we can use an interface reference to access its members. These lines are extracted from the `Main()` method, and show how an interface reference can be used to access an implementing class's members through polymorphism.

```
IEmployee employee;

Instructor instructor = new Instructor("Mr. Bean");
Manager manager = new Manager("Mr. Listentomecarefully");

// apply polymorphism using IEmployee reference
employee = instructor;
employee.StartWorking();

employee = manager;
employee.StartWorking();
```

Note that through an `IEmployee` reference we can only call the `StartWorking()` method, plus the four members of `System.Object`, which every object has and we talk about later in this chapter. For now, just think that through an interface reference we can only access the members defined by that interface. The following calls would result in a compile-time error, no matter what the real object behind `employee` is:

```
employee.Teach();
employee.DescribeYourself();
```

Implementing Multiple Interfaces

In order to demonstrate interface inheritance using more than one interface, let's add another design goal to our example. We decide that the ability to describe itself shouldn't be a part of all `Person` objects. We may want to have classes that derive from `Person` that do have this ability, and classes that don't.

In the new scenario, the only functionality provided by `Person` is its ability to store the person's name when constructing the object (through its constructor). `Person` will look like this:

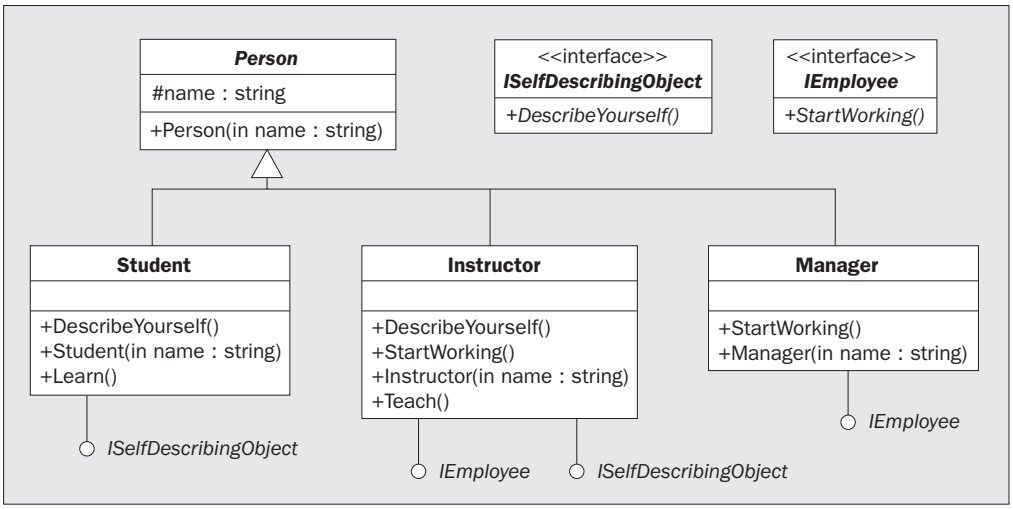
```
abstract class Person
{
   protected string name;

   public Person(string name)
   {
      this.name = name;
   }
}
```

We decide that it would be very useful to have an interface, named `ISelfDescribingObject`, that would contain the `DescribeYourself()` method. We would implement this interface on classes that should have the ability to describe themselves.

At this point, I think you may be wondering about one thing: Why do we need an interface? Why don't we simply implement `DescribeYourself()` in the classes we want, without the trouble of adding an interface? For now, please just follow the steps to implement this idea. We'll see in the next section how this can be useful in real-world situations.

This is what we want to achieve:



Note that with this new version, we deliberately didn't implement `ISelfDescribingObject` in `Manager`. For our experiments, we'll need a class that doesn't implement this interface. In order to implement this design, follow the steps:

1. Remove the definition for `DescribeYourself()` from the `Person` class. The class should now look like:

```
abstract class Person
{
   protected string name;

   public Person(string name)
   {
      this.name = name;
   }
}
```

2. Now let's add the `ISelfDescribingObject` interface:

```
interface ISelfDescribingObject
{
   void DescribeYourself();
}
```

3. Update the `Instructor` class to use `ISelfDescribingObject`. First, we need to modify its declaration, by adding yet another interface:

```
class Instructor: Person, IEmployee, ISelfDescribingObject
```

Then remove the `override` keyword from the `DescribeYourself()` declaration:

```
public void DescribeYourself()
```

Note that the rest of the code doesn't need to change. Since we have a `DescribeYourself()` method, we are honoring the `ISelfDescribingObject` contract.

4. We update now the `Student` class to use `ISelfDescribingObject`:

```
class Student: Person, ISelfDescribingObject
```

We need to remove the `override` keyword from `DescribeYourself()` declaration:

```
public void DescribeYourself()
```

5. We don't want to provide the `Manager` class with the ability to describe itself. Just remove or comment out the `DescribeYourself()` method.

6. The final step is to modify the `Main()` method of `MyExecutableClass`, because the old version calls `DescribeYourself()` on `Person` objects, which is no longer possible. Modify `Main()` to:

```
static void Main(string[] args)
{
   IEmployee employee;
   ISelfDescribingObject sdo;

   Student student = new Student("Joey");
   Instructor instructor = new Instructor("Mr. Bean");
   Manager manager = new Manager("Mr. Listentomecarefully");

   sdo = instructor;
   sdo.DescribeYourself();

   sdo = student;
   sdo.DescribeYourself();

   employee = instructor;
   employee.StartWorking();

   employee = manager;
   employee.StartWorking();
}
```

Now execute the program, and see the following:

```
Good morning class. My name is Mr. Bean, and I am your instructor.
Hello man, I'm Joey. I'm a cool student.
Going to my dear students...
No, it's my golf day.
```

If you understood the previous example, you should have no problem understanding this one too. Probably the most interesting part is the new `Instructor`'s definition, which now derives from `Person`, but also implements `ISelfDescribingObject` and `IEmployee`.

Interface Recap

Before we move on, let's review some of the key points about interfaces:

- An interface is set of property and method signatures, and serves as a **contract** for classes that implement it.
- When a class implements an interface, it guarantees that it will implement every signature defined in that interface.
- With interface inheritance, we only inherit the interface – the interface is only allowed to have signatures with no implementations. Thus with interface inheritance we can't inherit any implementations.
- It's possible for a single class to implement multiple interfaces.

Static Class Members

The ability of having static members in a class is a new concept to Visual Basic 6 developers. However, we've already used static members in this chapter – without realizing it.

All of the classes we've created so far have only included instance members. An instance member is a member that belongs to a particular instance of a class – in other words, to a particular object. For example, take a look at the following code snippet:

```
Student student = new Student();
student.Name = "Peter";
student.DescribeYourself();
```

Here, we access `Name` and `DescribeYourself()` on an object named `student`. `Name` and `DescribeYourself()` are instance members of the `Student` class. `Name` is an instance field (or perhaps property), and `DescribeYourself()` is an instance method.

Static class members (fields, methods, and properties) are members that don't belong to a particular class instance, but rather to the class as a whole. Before implementing static members in an example, let's first look at a very popular static member that we used many times:

```
Console.WriteLine ("very interesting text here");
```

The `WriteLine` method of the `Console` class is one of our dearest friends, at least in this chapter. However, we never instantiate a `Console` object, and then call the `WriteLine` method on the object:

```
Console c = new Console(); // this never happens
c.WriteLine(); // this never happens
```

Instead, we call `WriteLine` directly on the `Console` class. This is a hint for us that `WriteLine` is a static method of `Console`. The `Console` class was created as a utility class that shouldn't be instantiated, and provides us with some general-use members. There are a few more things to note about the `Console` class, in regard to the previous code snippet:

- `Console` cannot be instantiated because it has a `private` constructor. Having private constructors in a class is an effective way to prevent other classes or programs from creating instances of our class.
- `Console` cannot be inherited from because it is `sealed`. Utility classes are a good example of classes that are in their 'final' version and we don't want anybody to derive from them.
- Even if we could create a `Console` instance, `WriteLine()` could not be called on that instance. Static members can't be accessed using a class's instance.

For these reasons, an attempt to use the two lines of code presented above would fail dramatically.

> *Note that in C# the `static` keyword has a different meaning from Visual Basic 6. In Visual Basic, we marked a local variable that we wanted to maintain value between method calls as `static`. In C# it is used to mark a class member that isn't meant to be used on a particular instance of the class, but on the class as a whole.*

Microsoft .NET provides us with more utility classes like `Console` that are composed of static members, another popular one being `System.Math` (which is also sealed and not creatable). We are also allowed to have classes that contain both static members and instance members. Take a look at this short example, which shows a few static members in action:

```
class OnlyStaticMembers
{
   static void Main()
   {
      Console.WriteLine("Max(5,10): " + Math.Max(5,10));
      Console.WriteLine("Sqrt(25): " + Math.Sqrt(25));
      Console.WriteLine("Int32.MaxValue: " + Int32.MaxValue);
      Console.WriteLine("String.Concat: " + String.Concat("Hello ","World!"));
   }
}
```

This should produce the following output:

```
Max(5,10): 10
Sqrt(25): 5
Int32.MaxValue: 2147483647
String.Concat: Hello World!
Press any key to continue
```

Now that we have met a few popular static fields, let's play a bit with our dear `Person` and create our own static members.

Adding Static Members to the Person Class

Before doing any changes to the code, let's imagine a scenario when we would need to have static data in the `Person` class. We can't make any of the members already presented static, because it wouldn't make much sense; we had the `Name` property that does belong to a particular `Person`; the same for the `DescribeYourself()` method that allowed a specific `Person` to describe itself.

For our example, let's say we wanted to make the `Person` class be aware of how many `Person` objects have been created so far. Let's first see the code, then we'll analyze how it works. Modify your existing `Person` class to:

```
class Person
{
   protected string name;
```

```
      private static int howManyPersons = 0;

      public Person(string name)
      {
         this.name = name;
         howManyPersons++;
      }

      public static void ShowHowManyPersons()
      {
         Console.WriteLine("{0} persons created so far.", howManyPersons);
      }

      public virtual void DescribeYourself()
      {
         Console.WriteLine
            ("Hello! My name is {0} and I am a nice person.", name);
      }
   }
```

The important lines are highlighted. Note that `DescribeYourself()` is not abstract any more (as it was in our last example). For the purposes of this example, it's better to have an implementation for `DescribeYourself()`. In this example we'll also use the `Student` class. The `Student` class should already look like this, if it doesn't then please change it:

```
class Student: Person
{
  public Student(string name): base(name)  {}

  public sealed override void DescribeYourself()
  {
    Console.WriteLine
      ("Hello man, I'm {0}. I'm a cool student.", name);
  }

  public void Learn()
  {
    Console.WriteLine
      ("Student wakes up, yawns, scratches head, and tries to look interested.");
  }
}
```

The final step is to write the code for `MyExecutableClass`:

```
  class MyExecutableClass
  {
    static void Main(string[] args)
    {
      Person firstPerson = new Person("Eddie");
      firstPerson.DescribeYourself();
```

```
        Person.ShowHowManyPersons();

        Person secondPerson = new Person("Mike");
        secondPerson.DescribeYourself();

        Person.ShowHowManyPersons();

        Student firstStudent = new Student("Alice");
        firstStudent.DescribeYourself();

        Person.ShowHowManyPersons();

        Student secondStudent = new Student("Chris");
        secondStudent.DescribeYourself();

        Student.ShowHowManyPersons();
      }
    }
```

OK! Run the program, and expect to see:

```
Hello! My name is Eddie and I am a nice person.
1 persons created so far.
Hello! My name is Mike and I am a nice person.
2 persons created so far.
Hello man, I'm Alice. I'm a cool student.
3 persons created so far.
Hello man, I'm Chris. I'm a cool student.
4 persons created so far.
Press any key to continue
```

What's going on here? The whole magic is being done in the `Person` class, so we'll analyze it first. At the heart of the counting mechanism is a static field named `howManyPersons`. Static class members are declared using the `static` modifier:

```
      private static int howManyPersons = 0;
```

The initial value is zero, and we increase it in `Person`'s instance constructor – meaning that it will be increased every time we create a new instance of `Person`:

```
      public Person(string name)
      {
         this.name = name;
         howManyPersons++;
      }
```

howManyPersons is a field private, so we won't need to deal with it directly from outside. Instead, we have a static method that writes the number of Person objects created to the console:

```
public static void ShowHowManyPersons()
{
   Console.WriteLine("{0} persons created so far.", howManyPersons);
}
```

Looking at this method, you can see that it looks just like a non-static method. But because it is static, we can't call it like we call an instance method – instead we call it using the class's name. We can also call ShowHowManyPersons() on Student, since Student inherits from Person:

```
static void Main(string[] args)
{
  Person firstPerson = new Person("Eddie");
  firstPerson.DescribeYourself();

  Person.ShowHowManyPersons();
  // ...

  Student secondStudent = new Student("Chris");
  secondStudent.DescribeYourself();

  Student.ShowHowManyPersons();
  // ...
}
```

Accessing Static Members from Instance Methods

We are allowed to access static members from instance class members. The reverse is not true. To demonstrate the first statement, remove the static modifier from the ShowHowManyPersons() definition:

```
public void ShowHowManyPersons()
{
  Console.WriteLine  ("{0} persons created so far.", howManyPersons);
}
```

This is equivalent to:

```
public void ShowHowManyPersons()
{
  Console.WriteLine  ("{0} persons created so far.", Person.howManyPersons);
}
```

The method is not static any more, but it can still access howManyPersons, which is a static field. This is because there's no reason why a static field wouldn't be accessible to the class's instance members.

Now that `ShowHowManyPersons()` is an instance method, we must access it using a class instance. Change all calls to `Person.ShowHowManyPersons()` and `Student.ShowHowManyPersons()` to `firstPerson.ShowHowManyPersons()`. Now when we rerun the program, you can see the output is the same, but we used an instance method instead of a static method to get the number of persons created.

Analyzing the two options we have for `ShowHowManyPersons()`, it is pretty obvious that it's better to have it as a static method. First of all, this method logically is a method concerning the whole class, and not a particular object. Second, we must create a `Person` object in order to be able to find out how many objects were created, which is not very good – it's much better to be able to find out information concerning the class even without creating an object from it.

Now that we have analyzed our options, make `ShowHowManyOptions()` static again, and fix the `Main()` method to call the method on `Person` instead of `firstPerson`.

We have seen that it's possible to access a static member from an instance member. The reverse, though, is not possible. Think about it for a second: does it make sense to add the following line to the `ShowHowManyPersons()` static method?

```
public static void ShowHowManyPersons()
{
  Console.WriteLine  ("{0} persons created so far.", Person.howManyPersons);
  Console.WriteLine("My name is " + name);
}
```

If `ShowHowManyPersons()` is a static method, it will be called using `Person.ShowHowManyPersons()`. What value should `name` return? We might not even have a single `Person` created. From this we deduce that it's not allowed (and it wouldn't even make sense) to access instance fields from a static method. The error generated by the compiler is: An object reference is required for the nonstatic field, method, or property 'Person.name'

The rule is: *we can access a static field from an instance property, method or constructor.* The reverse is not true – *we cannot access an instance class member from a static member.*

The last thing to note about the current example is that the counting also works when we create `Student` objects, even if the counting mechanism is implemented in `Person`:

```
        Student student = new Student("Chris");
        student.DescribeYourself();

        Person.ShowHowManyPersons();
```

We learned earlier in the previous chapter that when we create a `Student` object, `Person`'s constructor is called first. The program demonstrates that value of `howManyPersons` increases even when we create `Student` objects, demonstrating that the `Person` constructor does get called.

Static Constructors

C# possess a feature called **static constructors**. Static constructors are used to initialize any static members the class might have. We're not allowed to access any instance members from the static constructor, so we can only play with class's static data.

A static constructor belongs to the class as a whole, so it gets called only once in an application, not every time an instance of the class is created. It's hard to be sure exactly when the static constructor gets called, but we know it will be before any static members are used, and before any instances of the class are created. Most of the time the static constructor is called immediately before the class is used for the first time.

Static constructors are not allowed to have a return type, parameters, or access modifiers. None of these would make sense in the context of a static constructor. A class can have a static constructor in addition to its instance constructors. There is no conflict between a parameterless instance constructor and a static constructor.

For a simple example, we will continue to work on the scenario we imagined in the *Adding Static Members to the Person Class* section. This time we want to improve our `Person` class so that it knows the maximum number of persons that can be created overall. This number would be loaded from a database, and stored in a static class field named `maxPersons`. If the maximum number of `Persons` is exceeded, a warning message should be displayed on the screen.

If this seems scary, don't worry. We won't really use a database in this chapter – although we will learn how to later in the book. For this example, we'll assume that the maximum number of persons read from the database is 2. Modify the `Person` class like this:

```
class Person
{
   protected string name;
   private static int howManyPersons;
   private static int maxPersons;

   public Person(string name)
   {
      this.name = name;
      if (++howManyPersons>maxPersons)
         Console.WriteLine("Warning! Maximum number of persons exceeded!");
   }

   static Person()
   {
      howManyPersons = 0;
      maxPersons = 2;
   }

   public static void ShowHowManyPersons()
   {
      Console.WriteLine
         ("{0} persons created so far.", Person.howManyPersons);
```

```
    }

    public virtual void DescribeYourself()
    {
       Console.WriteLine
       ("Hello! My name is {0} and I am a nice person.", name);
    }
}
```

When you run the program, you should see:

```
Hello! My name is Eddie and I am a nice person.
1 persons created so far.
Hello! My name is Mike and I am a nice person.
2 persons created so far.
Warning! Maximum number of persons exceeded!
Hello man, I'm Alice. I'm a cool student.
3 persons created so far.
Warning! Maximum number of persons exceeded!
Hello man, I'm Chris. I'm a cool student.
4 persons created so far.
Press any key to continue
```

Since we've learned a lot about static fields, the way that this works should be obvious. Just notice the syntax used to declare the static constructors, and observe that the static fields really get initialized. After the first two persons are created, the instance constructor of `Person` displays the warning message for every additional person created.

Destructors

The destructor is a special kind of method that gets called when an object is removed from memory. This usually happens when the object goes out of scope, and the destructor's role is to allow the object to free any resources it has been using, before being finally deleted from memory. In Visual Basic there was a method called `Class_Finalize` that served a similar purpose, although C# destructors work in a very different way.

The most important thing to know about C# destructors is that their execution is not deterministic: after the object goes out of scope, it is only marked for destruction. The Garbage Collector, which is a sophisticated memory management mechanism implemented by the .NET runtime, every so often frees the memory consumed by unused objects. The object's destructor is invoked by the Garbage Collector right before the object is actually removed from memory.

If your object holds important resources, such as database connections to a server, it is important for the object to start the resource-releasing sequence as soon as it goes out of scope. This kind of behavior is implemented using a method named `Dispose()`. This is an advanced topic which will not be covered in this book. Don't worry – there are still many interesting things to learn.

Let's implement a destructor in the `Person` class to provide some functionality that has long been needed. In our previous examples we used a `static` field named `howManyPersons` that increased every time we created a new person. However, the design didn't consider the possibility that some objects might actually be destroyed. `howManyPersons` would never decrease. This would reduce the chance of the warning message appearing, because out-of-scope objects would no longer count towards the total.

Please update the `Person` class from the previous example by adding a destructor like this:

```
class Person
{
  protected string name;
  private static int howManyPersons;
  private static int maxPersons;

  public Person(string name)
  {
    this.name = name;
    if (++howManyPersons>maxPersons)
      Console.WriteLine("Warning! Maximum number of persons exceeded!");
  }

  static Person()
  {
    howManyPersons = 0;
    maxPersons = 2;
  }

  ~Person()
  {
    howManyPersons--;
    Console.WriteLine
      ("{0} going offline. {1} active persons remaining.", name, howManyPersons);
  }

  public static void ShowHowManyPersons()
  {
    Console.WriteLine("{0} persons created so far.", Person.howManyPersons);
  }

  public virtual void DescribeYourself()
  {
    Console.WriteLine
      ("Hello! My name is {0} and I am a nice person.", name);
  }
}
```

Now execute the program again. Expect output similar to this:

```
Hello! My name is Eddie and I am a nice person.
1 persons created so far.
Hello! My name is Mike and I am a nice person.
2 persons created so far.
Warning! Maximum number of persons exceeded!
Hello man, I'm Alice. I'm a cool student.
3 persons created so far.
Warning! Maximum number of persons exceeded!
Hello man, I'm Chris. I'm a cool student.
4 persons created so far.
Chris going offline. 3 active persons remaining.
Eddie going offline. 2 active persons remaining.
Alice going offline. 1 active persons remaining.
Mike going offline. 0 active persons remaining.
Press any key to continue
```

Even if destructors are usually called at mysterious times chosen by the Garbage Collector, they are always executed before the program completes execution. This explains why all destructors are called when our program terminates executing. The order in which the objects are destroyed cannot be always predetermined, so you might see your objects 'disappear' in a different order.

Delegates and Events

Although the notion of an event is well known to Visual Basic programmers, the concept of delegates is probably shrouded in complete mystery, so let's talk about these in this introduction. We'll then analyze delegates and events with technical details and examples in the next two subsections.

A delegate is a special kind of .NET data type, and delegates are particularly important because they are at the heart of event handling mechanism – and as a Visual Basic programmer, you know that without events and event handling we wouldn't be able to create useful Windows applications.

With both Visual Basic 6 and Visual C#, in order to generate code for a `Button` event handler, it is enough to double-click on the button. Still, what Visual Studio .NET does behind scenes is greatly different from what went on in Visual Studio 6. In Visual Basic 6, all the code you could see in your file was something like:

```
Private Sub Command1_Click()
    MsgBox "You just pressed this button"
End Sub
```

That would be it – no more, no less – you could have a fully working program, even with some functionality, with just three lines of code. Indeed, Visual Basic 6 does a very good job of hiding what happens behind the scenes. This is very good because it makes application programming easy – but it places many constraints on programmers who want more flexibility.

Visual Studio .NET does the same great job of automatically generating code, still the code is more involved now and you can no longer have a fully working button with only three lines of code. This section aims to teach you what you need to know in order to use delegates and events in your programs.

> *Because Visual C# does a good job at hiding most details of handling events and delegates, it is not required to understand them in order to create Windows Form Applications. But as with any other concept presented in this chapter, learning delegates will make you a stronger programmer, because it allows you to understand* ***why*** `button1_Click()` *is called when you click on that button.*

Delegates

A bit earlier I said that delegates represent a special kind of data type. Why are they so special, you might ask? Well, while `int` variables store numbers, and objects store class instances, delegates store *references to methods.*

Just as we can send an integer, a string, or an object as a method parameter, delegates allow us to send *methods* as parameters. Sounds exciting? It really is!

Delegates are used when a class must call a method on another class, but doesn't know at compile-time what that method is. Since a delegate instance references a particular method, it can be sent as a parameter to another method; the method that receives the delegate as a parameter can use it to call the original method the delegate is pointing to.

Although delegates are references to methods, thus acting like methods, technically they are a kind of classes – in fact, all delegates derive from a class named `System.Delegate`. There is yet another kind of delegate, called a multicast delegate, that derives from `System.MulticastDelegate` which in turn derives from `System.Delegate`. Multicast delegates are a special kind of delegates that can hold references for more than one method – we'll see an example soon.

Also, since a delegate is a data type, it means that it can be instantiated, just like a class. Unlike a classes, we don't have terminology to differentiate between a delegate and a delegate instance. The delegate *type* specifies the kind of methods its instances will reference. For example, we can have a delegate that is defined to work for methods that take no parameters and have no output values. We can then create any number of delegate instances, every instance referencing (probably different, but not necessarily) methods that comply with the delegate's definition (no parameters, no return values).

Since this is your first exposure to delegates, all this theory might be a bit confusing at first. Let's start writing some code, and everything should become clearer.

Upgrading Person to use Delegates

In its previous version from the *Destructors* section, the `Person` class writes various warning messages to the screen when something happens to it. For example, it sends a warning message when the number of instances created exceeds the maximum allowed number. It also sends a message when it is destroyed in the destructor – we'll consider this message to be a warning message.

In our case, writing to the screen is acceptable, but in a real-world scenario we would want to tell the class managing the `Person` object (`MyExecutableClass` in this example) that something is happening to its objects. `MyExecutableClass` should receive some sort of signal from its `Person` objects, and decide what to do depending on the nature of the received signal.

This is the current design goal we're trying to achieve at this section: a `Person` shouldn't display the warning messages by itself; instead it should send the warning message as a notice to the class that created it, by calling a method of that class and supplying the text of the message as a parameter. We don't know ahead of time what class is creating `Person` objects and needs to be notified, and also we don't know what method is to be called for notification.

As I mentioned earlier, delegates are useful when a method wants to call another method (probably of another class), but doesn't know what method – we specify the method to be called using a delegate.

We'll use delegates to meet the design requirements. We'll add a method to `MyExecutableClass` named `ProcessWarningMessage()` that will be called by `Person` objects when they want to send a warning message. When creating `Person` objects, we'll pass a delegate instance that references the `ProcessWarningMessage()` method to their constructor. They will use this delegate instance to call the method and pass the warning message to it.

Now that we know what we want to do, let's start modifying the current program. Before proceeding, make sure you have the `PersonExample` solution working as presented in the previous section. Since there are more modifications needed, let's go through upgrading the solution step by step.

1. First let's add `ProcessWarningMessage()` to `MyExecutableClass`. As mentioned earlier, this is the method that will be called by `Person` objects when they want to warn us about something.

```
static void ProcessWarningMessage(string warning)
{
   // very complicated code here
   Console.WriteLine("Warning: " + warning);
}
```

Note that in a real-world application, the signal sent by the `Person` would contain a lot of mysterious codes and data that would be understood by `MyExecutableClass`; for our current example, we'll just send out the warning message's text.

In this example the method to be called is a static method, but be aware that delegates also work with instance methods.

2. We will need to instruct `Person` to pass the warning message to this method instead of displaying it on the screen. For this, we need to declare a delegate. Add `WarningEventHandler` to the `PersonExample` namespace, not inside any class. It must be visible from all classes that use it, so it's easier to place it in the same namespace with `Person` and `MyExecutableClass` (otherwise, you would need to import its namespace or use its fully qualified name in those classes).

```
delegate void WarningEventHandler(string message);
```

This defines a delegate type to be used for methods that have no return values, and receive a `string` as a parameter. Since `ProcessWarningMessage` exactly fits this description, we'll be able to create an instance of the `WarningEventHandler` delegate and make it reference `ProcessWarningMessage()`.

Note that the delegate type `WarningEventHandler` does not reference any method by itself; it is just a type, a "blueprint", just like a class. In order to make it reference actual methods, we must create instances of this delegate.

3. Now that we have a delegate to use for transmitting the warning messages, let's modify `Person` to use it. There are three sub-steps that we must follow here:

- First, we need to add in `Person` a private field of the type of the delegate. We'll use this delegate to call `ProcessWarningMessage()` of `MyExecutableClass`:

```
class Person
{
   protected string name;
   private static int howManyPersons;
   private static int maxPersons;
   private WarningEventHandler warningMethodToCall;
```

Let me highlight once again that both `WarningEventHandler` and `warningMethodToCall` are referred to as delegates – even if `WarningEventHandler` is the type, and `warningMethodToCall` is an instance of that type. With classes, we have a separate name for their instances – objects. For delegates, we use the same name to refer to the type or to its instances. Keep this in mind when reading technical documentation about delegates, including this section of the chapter.

- Our second sub-step is to modify `Person`'s constructor to take a `WarningEventHandler` instance as a parameter, and store it to a `warningMethodToCall` field. Also in the constructor, we make use of `warningMethodToCall` for the first time – if the maximum number of persons exceeds the established value, then we need to send a warning:

```
public Person(string name, WarningEventHandler method)
{
   warningMethodToCall = method;
   this.name = name;
   if (++howManyPersons>maxPersons)
      warningMethodToCall("Maximum number of persons exceeded!");
}
```

- Our third and final sub-step is to instruct the destructor of `Person` to use the delegate instead of writing the destruction message to screen:

```
~Person()
{
   howManyPersons--;
   warningMethodToCall(
      String.Format("{0} going offline. {1} active persons remaining.",
                    name, howManyPersons));
}
```

4. We need to modify `Main()` in `MyExecutableClass` now. We create an instance of the delegate, and instruct that instance to act as a reference to the `ProcessWarningMessage()` method. Remember that a delegate instance can act as a reference to any method that has the same signature as delegate's definition. The delegate instance that we create is sent as a parameter to `Person`'s constructor:

```
static void Main(string[] args)
{
   WarningEventHandler warning = new WarningEventHandler(ProcessWarningMessage);

   Person firstPerson = new Person("Eddie", warning);
   firstPerson.DescribeYourself();

   Person secondPerson = new Person("Mike", warning);
   secondPerson.DescribeYourself();

   Person thirdPerson = new Person("Doug", warning);
   thirdPerson.DescribeYourself();
}
```

At this point it is interesting to see the syntax used to create a new delegate instance:

```
   WarningEventHandler warning = new WarningEventHandler(ProcessWarningMessage);
```

It looks similar to the syntax for creating objects. The method to be referenced is sent as a parameter. Note that in this case we're using a static method, so no class instance is required. If `ProcessWarningMessage()` was an instance method, we would need to create a class instance, and attach the method of that instance to the delegate.

OK, that's it. We have done quite a number of changes, so here is the whole program, implemented as a single file:

```
using System;

namespace PersonExample
{
   delegate void WarningEventHandler(string message);

   class MyExecutableClass
   {
      static void Main(string[] args)
      {
         WarningEventHandler warning = new
            WarningEventHandler(ProcessWarningMessage);

         Person firstPerson = new Person("Eddie", warning);
         firstPerson.DescribeYourself();

         Person secondPerson = new Person("Mike", warning);
         secondPerson.DescribeYourself();
```

```
            Person thirdPerson = new Person("Doug", warning);
            thirdPerson.DescribeYourself();
        }

        static void ProcessWarningMessage(string warning)
        {
            // very complicated code here
            Console.WriteLine("Warning: " + warning);
        }
    }

    class Person
    {
        protected string name;
        private static int howManyPersons;
        private static int maxPersons;

        private WarningEventHandler warningMethodToCall;

        public Person(string name, WarningEventHandler method)
        {
            warningMethodToCall = method;
            this.name = name;
            if (++howManyPersons>maxPersons)
                warningMethodToCall("Maximum number of persons exceeded!");
        }

        static Person()
        {
            howManyPersons = 0;
            maxPersons = 2;
        }

        ~Person()
        {
            howManyPersons--;
            warningMethodToCall
                (String.Format("{0} going offline. {1} active persons remaining.",
                name, howManyPersons));
        }

        public static void ShowHowManyPersons()
        {
            Console.WriteLine("{0} persons created so far.", Person.howManyPersons);
        }

        public virtual void DescribeYourself()
        {
            Console.WriteLine
                ("Hello! My name is {0} and I am a nice person.", name);
        }
    }
}
```

If everything looks OK, press *Ctrl-F5* to execute the program. You should see this output:

```
Hello! My name is Eddie and I am a nice person.
Hello! My name is Mike and I am a nice person.
Warning: Maximum number of persons exceeded!
Hello! My name is Doug and I am a nice person.
Warning: Doug going offline. 2 active persons remaining.
Warning: Eddie going offline. 1 active persons remaining.
Warning: Mike going offline. 0 active persons remaining.
Press any key to continue
```

Multicast Delegates

A multicast delegate is a delegate that holds a reference not to one, but multiple methods. We won't go into too much detail here, and using a multicast delegate doesn't help too much in our `Person` example anyway. In order to use a multicast delegate, modify `MyExecutableClass` to:

```
class MyExecutableClass
{
   static void Main(string[] args)
   {
      WarningDelegate delegate1 = new WarningDelegate(FirstMethod);
      WarningDelegate delegate2 = new WarningDelegate(SecondMethod);
      WarningDelegate multicast = delegate1 + delegate2;

      Person firstPerson = new Person("Eddie", multicast);
      firstPerson.DescribeYourself();

      Person secondPerson = new Person("Mike", multicast);
      secondPerson.DescribeYourself();

      Person thirdPerson = new Person("Doug", multicast);
      thirdPerson.DescribeYourself();
   }

   static void FirstMethod(string warning)
   {
      Console.WriteLine("Method1: " + warning);
   }

   static void SecondMethod(string warning)
   {
      Console.WriteLine("Method2: " + warning);
   }
}
```

The output is:

```
Hello! My name is Eddie and I am a nice person.
Hello! My name is Mike and I am a nice person.
Method1: Maximum number of persons exceeded!
Method2: Maximum number of persons exceeded!
Hello! My name is Doug and I am a nice person.
Method1: Doug going offline. 2 active persons remaining.
Method2: Doug going offline. 2 active persons remaining.
Method1: Eddie going offline. 1 active persons remaining.
Method2: Eddie going offline. 1 active persons remaining.
Method1: Mike going offline. 0 active persons remaining.
Method2: Mike going offline. 0 active persons remaining.
Press any key to continue
```

As a note, instead of:

```
WarningDelegate delegate1 = new WarningDelegate(FirstMethod);
WarningDelegate delegate2 = new WarningDelegate(SecondMethod);
WarningDelegate multicast = delegate1 + delegate2;
```

we could have used the shortcut syntax:

```
WarningDelegate multicast = new WarningDelegate(FirstMethod);
multicast += new WarningDelegate(SecondMethod);
```

Events

Events play a very important role in programming for the Windows platform because they are the main means by which an application interacts with the user, but they are not limited to this area. They can be used in situations where an executing object needs to send signals to a number of other objects that may be listening.

Let's briefly see how events work by looking at the typical `Button` example. When a user clicks on a `Button`, that button **raises an event** saying that it was clicked on. If the program that includes the `Button` has **subscribed** to its `Click` event (meaning that it is listening for this event), it responds to the event by executing its associated **event handler method**.

Any class can generate many different kinds of events. Programs that use that class are able to subscribe to a number of those events; when a program subscribes to an event, it will be notified when the class generates that event. The class notifies its listeners that it's raising the event by calling a method on them (sounds familiar?), and this is implemented by using delegates.

After you have seen how to use delegates and how we can allow a class to call a method of another class through a delegate, you might be wondering why events are useful, after all – the same functionality can be achieved by only using delegates. Why complicate things further by introducing yet another concept?

Indeed, events add a bit of additional complexity, especially when coding the class that generates events, but they are more appropriate to use in situations where we don't know much about who's going to subscribe. They provide an infrastructure in which the class that generates an event does not need to know anything about what classes are listening to these events.

Any number of programs can subscribe to *or unsubscribe* from an object's event at any time. The subscription/unsubscription mechanism is standard, and doesn't need any complicated coding at either side. When the object wants to generate an event, it doesn't need to keep track of who subscribed to this event – the event handling system will take care of these details for us.

Events also have some requirements that allow them to be used in a standardized way. As you'll see, every event handler method is required to have two parameters – one being the object that generates the event (very useful, if we use the same method to handle events from different sources), and the second being an object derived from `System.EventArgs`.

OK, let's see some events in action. We'll modify the program we have from the *Delegates* section earlier, and we'll update it to use events.

Upgrading Person to use Events

We'll need to update the `Person` class to generate events, and then we'll also modify `MyExecutableClass` to make use of the events generated by `Person`. The event we'll generate in `Person` is called `Warning`, and it will contain the warning text message. In `MyExecutableClass` we'll capture the events generated by `Person` objects, and we'll write the text message of the event to the screen.

In the previous example with delegates, we sent the warning message directly as a string variable. With events, it's not this simple – the data we want to pass along with the event is required to be encapsulated in a class that derives from `System.EventArgs`. In `MyExecutableClass`, we'll receive this object and we'll extract the string warning message from it.

> *Note that most of the time you'll only* ***consume events*** *generated by other classes, mostly the predefined controls. Still, you'll be a much stronger programmer after you understand the whole mechanism of raising and handling events.*

While writing code I'll also point out the naming conventions for events recommended by Microsoft. Before proceeding, let me highlight once again that you'll see the real usefulness of events when we get to use the `Person` class.

Now let's start implementing the `Warning` event:

1. Let's first add the class that will wrap the text message. As I told you earlier, this is a requirement for events. This class could contain many methods and properties, but in our case it is very simple because we only want to store a simple `string`. The naming recommendation for this class is to add *EventArgs* as a suffix on the event's name, so let me present the `WarningEventArgs` class:

```
class WarningEventArgs : EventArgs
{
   private string message;

   public string Message
   {
      get { return message; }
   }

   public WarningEventArgs(string warning)
   {
      this.message = warning;
   }
}
```

2. Now we need to modify the delegate definition. In the previous program, it looked like this:

```
delegate void WarningEventHandler(string message); // part of previous code
```

This delegate was used to hold a reference to the `ProcessWarningMessage()` method of `MyExecutableClass`, which took a string as a parameter. I mentioned that with events, we must follow a strict definition for the event handler method, so we have to modify the delegate to:

```
    delegate void WarningEventHandler (object sender, WarningEventArgs e);
```

The delegate now has two parameters, and no return value. This is a requirement to use this delegate for the generation of events. The first parameter is the object that raised the event and the second must be an object derived from `System.EventArgs`. The suggested naming convention for the event handler is to append *EventHandler* to the event's name.

Observe that the second parameter is a `WarningEventArgs` object. This is no great surprise, because I mentioned earlier we can't send the warning message directly, but a `WarningEventArgs` object that contains the warning message. But what about the first parameter? Since events are very flexible in nature and work for various, unrelated classes, it's important to know which object has raised the event. As you'll see, in `MyExecutableClass` we'll create several `Person` objects that will use a single event handler method.

When a method is used as an event handler for more than one object, we'll use the `sender` argument to find out what object generated the event. We'll see a bit later how to do that.

3. Now, let's add the `Warning` event declaration to the `Person` class:

```
protected string name;
private static int howManyPersons;
private static int maxPersons;
public event WarningEventHandler Warning;
```

This is a replacement for the following line from the code in the *Delegates* section:

```
private WarningEventHandler warningMethodToCall; // part of previous code
```

In the *Delegates* section code, we used `warningMethodToCall`, which was an instance of the `WarningEventHandler` delegate. Now, instead of using a delegate instance, we're raising events. `Warning` can also be seen as an instance of `WarningEventHandler`, and it is used in a similar way, but now it is an event and is defined using the `event` keyword.

1. Instead of raising the event directly using `Warning`, it is recommended to add an additional method in your code, which in turn raises the `Warning` event. This method is typically named with *On* as a prefix to the event's name:

```
protected virtual void OnWarning(WarningEventArgs e)
{
   if (Warning != null) { Warning (this, e); }
}
```

The important thing to note here is that we check to see if `Warning` is not `null` – this is a check to see if anybody is actually listening for the event with a suitable event handler. If there were no methods associated with the `WarningEventHandler` delegate, a run-time error would be generated, and so the check to see if `Warning` is not `null` will avoid this.

As you can see, the recommendation to implement this method is quite useful. It's easier to call `OnWarning()` instead of calling `Warning()` directly and testing every time to see if there is somebody actually listening to the event.

2. We need to perform some housekeeping in `Person`, and update it to use `OnWarning()`. The `Person` instance constructor should become:

```
public Person(string name)
{
   howManyPersons++;
   this.name = name;
}
```

As you can see, there is no sign of the delegate instance as a parameter now. Events can be associated at any point of time, not just in the constructor or in a separate method. We'll see how to listen to events a bit later, when coding `MyExecutableClass`.

In the previous version, we sent a warning message in the `Person` constructor. We aren't doing the same here, because a `Person` event can be assigned an event handler in `MyExecutableClass` only after `Person` is created. So if we tried to raise the event here in the constructor, it wouldn't make much difference, because nobody's listening yet.

Feel free to remove `static int maxPersons` and the static constructor from `Person`, as we won't use them any more. Also, feel free to remove `ShowHowManyPersons()` which will not be used any more.

3. Modify the `Person` destructor as follows. We want it to raise an event and warn us when the object is being garbage-collected:

```
~Person()
{
   howManyPersons--;
   string str = String.Format("{0} going offline. {1} active persons remaining.",
                              name, howManyPersons);
   OnWarning (new WarningEventArgs(str));
}
```

4. Right now, the only place in `Person` that generates events is the destructor. It isn't much fun to send warning messages only from the destructor, and using our current `Person` there wouldn't be many places where it would make sense to raise a warning event. Add the following method to `Person`:

```
public void PleaseGenerateEvent()
{
   OnWarning (new WarningEventArgs(name + " generated this event!"));
}
```

When we want `Person` to raise an event, we can ask, politely, the `PleaseGenerateEvent()` method. Now we have two places in `Person` that raise the `Warning` event – the destructor and our brand-new coded method.

5. Now that we have a well behaved `Person` that generates events, let's modify `MyExecutableClass` to catch these events.

```
using System;

namespace PersonExample
{
   class MyExecutableClass
   {
      static void Main(string[] args)
      {
         Person eddie = new Person("Eddie");
         eddie.Warning += new WarningEventHandler (Warn);
         eddie.PleaseGenerateEvent();

         Person mike = new Person("Mike");
         mike.Warning += new WarningEventHandler (Warn);

         Person julia = new Person("Julia");
         julia.Warning += new WarningEventHandler (GirlsWarn);
         julia.PleaseGenerateEvent();
      }
```

```
        static void Warn(object sender, WarningEventArgs warn)
        {
           Console.WriteLine("Warning: " + warn.Message);
        }

        static void GirlsWarn(object sender, WarningEventArgs warn)
        {
           Console.WriteLine("A Girl warning: " + warn.Message);
        }
     }
  }
```

OK, that's it. Execute the program, and expect to see this output:

```
Warning: Eddie generated this event!
A Girl warning: Julia generated this event!
A Girl warning: Julia going offline. 2 active persons remaining.
Warning: Eddie going offline. 1 active persons remaining.
Warning: Mike going offline. 0 active persons remaining.
Press any key to continue
```

The most important thing to note about this code is how a program can enlist to receive events generated by an object:

```
           eddie.Warning += new WarningEventHandler (Warn);
```

This line instructs that whenever `eddie` generates a `Warning` event, the `Warn()` method of `MyExecutableClass` gets called. Note that for two objects we use one event handler method (`Warn()`), and for our third object we used another event handler (`GirlsWarn()`).

Now where have you seen this syntax before? That's right, it's the same syntax that you'll find in the code automatically generated by Visual Studio .NET when, for example, you double-click on a `Button` in the designer and the event-handler is added:

```
  this.button1.Click += new System.EventHandler(this.button1_Click);
```

Hopefully, this section should have helped you begin to understand what Visual Studio .NET is creating for you behind the scenes.

I mentioned earlier that if the same method is used as an event handler for multiple objects, then we can use the `sender` parameter to discover who the original sender of the event is. For example, we could change `Warn()` to:

```
        static void Warn(object sender, WarningEventArgs warn)
        {
           if (sender is Person)
           {
              Console.WriteLine("Message from:");
              Person p = sender as Person;
```

```
            p.DescribeYourself();
            Console.WriteLine("Warning: " + warn.Message);
         }
         else
         {
            Console.WriteLine("Message from unknown type of sender");
            Console.WriteLine("Warning: " + warn.Message);
         }
      }
```

The method now checks if `sender` is a `Person` object (we saw the `is` keyword in Chapter 3 and earlier in this chapter), and if so, retrieves the `Person` object using the `as` keyword, and then calls an appropriate method on this object. The `sender` is converted into a `Person` object with `sender as Person` – we've seen this technique earlier in the chapter as well.

In Chapter 3 we actually made extensive use of this technique in SweepCSharp – remember how we cast `sender` to a `Button` to extract information about the `Button` that was clicked, and also how we used the `System.Windows.Forms.MouseEventArgs` parameter of our `MouseDown` event handler, `OnMouseDown()`, to determine which mouse button was clicked? These examples demonstrated the kind of information that can be carried in the two parameters of an event handler.

Constants and Read-Only Fields

Almost every programming language allows programmers to declare constants. Constants are not new to Visual Basic 6 developers, but `readonly` fields are a new feature provided by C#.

Constants are used in situations when the value is known when we write the program, while `readonly` fields can be used in situations where the initial value needs to be calculated first. We'll take a brief look at each one in the following two sections.

Constants

Constants are fields marked with the `const` keyword, and represent a value that cannot be changed at run-time. *Their value must be known at compile-time.* The C# compiler replaces occurrences of each constant in your code with the literal value, which means that we can only use basic data types (such as strings or numbers) as constants. A constant cannot be set based on the value of a variable, or the result of a method:

```
public int value = 13;
public const int MinAge = value; // not allowed
```

… but it can be based on another constant, or simple calculations, because these can be evaluated at compile-time:

```
public const int MinAge = 15;
public const int MaxAge = MinAge + 100; // allowed
```

The compiler will now replace occurrences of `MinAge` with `15`, and occurrences of `MaxAge` with `115`. The compiled code does not use constants – they are all replaced by literals.

As far as the language is concerned, constants are `static`. We don't need to say that they are – but if we want to use constants from another class, we need to refer to them as members of the class – not a particular instance.

Let's see an example that highlights the most important facts about constants. Note that in this example we are starting over with our `Person` class. For the rest of the chapter, we also won't need any derived classes, so we'll not use `Student` and `Instructor` any more. If you want, you can create a new project now, in order not to ruin all the work you have done using persons, students, instructors, and managers.

Here's the new code – to save space, it's presented as if it were in one file. In fact, you can save time by typing it all into one file – C# files can contain several classes – or if you prefer, you can create them as separate files using Add New Item in Visual Studio .NET:

```
using System;

class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person myPerson = new Person();
      myPerson.Age = 50;
      myPerson.Age = 150;

      Console.WriteLine ("Minimum age: " + Person.MinAge);
   }
}

class Person
{
   public const int MinAge = 15;
   public const int MaxAge = MinAge + 100; // allowed

   private int age;

   public int Age
   {
      get
      {
         return age;
      }
      set
      {
         if (value >= MinAge && value <= MaxAge)
         {
            age = value;
            Console.WriteLine ("Age set to: " + value);
         }
         else
            Console.WriteLine ("Invalid value: " + value);
      }
   }
}
```

The output should be:

```
Age set to: 50
Invalid value: 150
Minimum age: 15
Press any key to continue
```

The `MinAge` and `MaxAge` constants are public. We can see in the `Main` method that constants are, indeed, static:

```
        Console.WriteLine ("Minimum age: " + Person.MinAge);
```

If you try to reference `MinAge` or `MaxAge` using a `Person` instance, you'll receive a compiler error:

```
        Console.WriteLine ("Minimum age: " + myPerson.MinAge); // error!
```

Because constants are static, we can't use the `this` keyword to reference them within their class. This is not correct:

```
            if (value >= this.MinAge && value <= this.MaxAge) // error!
```

but this works OK:

```
            if (value >= Person.MinAge && value <= Person.MaxAge) // OK!
```

Constants are great because they make code easier to read and easier to modify. Every programmer has bad memories of times when they haven't used constants properly. However, often we want an unchangeable value that we can only figure out at run-time (for example, if it's based on user input). In this case, we can use `readonly` fields – let's look at them now.

Read-Only Fields

We've seen that with constants, we need to know their exact value at write time. Sometimes, though, we want to be able to set the value of a field based on the result of a method, or something else not known when we're writing the program, and then prevent it from changing again. To achieve this we can use `readonly` fields.

A `readonly` field provides more flexibility than a constant because it allows you to set its initial value in the constructor of the class that contains it. In fact, we can change its value as many times as we like, but only in the constructor. Another degree of a `readonly` field's flexibility over constants is that we are allowed to have `readonly` fields of complex types (such as `Person`), while constants are limited to simple types.

Note that if we were to create a `readonly` field of a complex object, such as `Person`, the *members* are not read-only – we are allowed to edit the object. However, we are not allowed to assign a new object to the field – this is what read-only usually means in the context of objects.

`readonly` fields are not static by default, but we can specify `static readonly` fields. If the `readonly` field is static, its initial value can only be set in a static constructor. If the `readonly` field is an instance field, its initial value can only be set in an instance constructor. Alternatively, for both kinds of `readonly` fields, their values can be set when they are defined:

```
private readonly int MinAge = 50;
```

To demonstrate using `readonly` fields, let's modify the previous example to use instance `readonly` fields:

```
using System;

class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person person = new Person(18, 65);
      person.Age = 22;
      person.Age = 15;
      person.Age = 70;
   }
}

class Person
{
   private readonly int MinAge;
   private readonly int MaxAge;

   private int age;

   public Person (int MinAge, int MaxAge)
   {
      Console.WriteLine ("Instance constructor of Person:");
      Console.WriteLine ("Setting MinAge to: " + MinAge);
      this.MinAge = MinAge;
      Console.WriteLine ("Setting MaxAge to: " + MaxAge);
      this.MaxAge = MaxAge;
   }

   public int Age
   {
      get
      {
         return age;
      }
      set
```

```
        {
           const int x = 10;
           if (value >= this.MinAge && value <= this.MaxAge)
           {
              age = value;
              Console.WriteLine ("Age set to: " + value);
           }
           else
              Console.WriteLine ("Invalid value: " + value);
        }
     }
  }
```

The output is:

```
Instance constructor of Person:
Setting MinAge to: 18
Setting MaxAge to: 65
Age set to: 22
Invalid value: 15
Invalid value: 70
Press any key to continue
```

If you try to modify `readonly` fields somewhere else in the program than in the constructor, the compiler will throw this error: A readonly field cannot be assigned to (except in a constructor or a variable initializer).

Value Types and Reference Types

In C# and .NET there are two categories of data types we deal with: value types and reference types. The basic difference between these kinds of types is the way they are stored in memory. Value types are stored directly, in a place named *the stack*, while for reference types the stack only holds a reference to the actual data – the value itself is in a place named *the (managed) heap*.

All primitive data types, such as `int`, `char`, `float`, and `bool`, are value types. Working with value types is always very fast, especially because CPUs have built-in features for working directly with these types. We talked at length about dealing with value types in Chapter 3.

Classes represent the most important kind of reference type. Typically, when we talk about reference types, it's safe for you to think about classes. The other common kind of reference type is the array – we looked at arrays in Chapter 3. Although it is often considered to be a 'basic' data type, the string is also a reference type.

When we declare a variable with a reference type, we are simply assigning space in the stack to act as a reference to an object of a given type:

```
Person myPerson;
```

Then when we construct it, we are creating a new object in the heap and assigning a reference to it back to our variable:

```
myPerson = new Person();
```

In .NET, the heap is called a managed heap because programmers don't have to explicitly destroy (free memory used by) objects they created. With lower-level programming languages such as C++, after finishing working with an object, we must take care of releasing the memory that was occupied. The object itself must know how to release all the resources and free all the memory it occupies. In .NET the Garbage Collector takes care of freeing the memory when we no longer use an object that was previously stored on the heap (a typical example is an object that was created in a method, and went out of scope after the method finished executing). We already met the Garbage Collector a few pages ago, in the *Destructors* section.

In the following sections we'll see several code samples that help us better understand the nature of value types and reference types. We won't investigate the low-level differences between storing data on the stack versus storing on the heap, but we'll see enough to provide a solid understanding of the implications of using value types and reference types.

Copying Values and References

When we copy a variable of a value type to another variable, we'll have two copies of the same value in memory:

```
int firstValue = 10;
int secondValue = firstValue;
```

When we declare two variables like this, we'll have the value '10' stored in two different places in memory (on the stack). On the other hand, when we copy an object to another object, only the reference is copied. We will have two separate copies of the reference on the stack, but both pointing to the same block of memory on the heap:

```
Person firstObject = new Person(10);
Person secondObject = firstObject;
```

The objects we create are likely to have many value types among their members. These value types are also stored on the heap, as part of the object they belong to. However, most of the principles about value types continue to apply, even if the value type happens to be stored on the heap.

We can also copy an object reference over the top of another object that has already been created in memory. In this case, the object constructed with `Person(50)` is lost and its memory will be freed by the Garbage Collector. We end up again by having two separate variables referencing the same object in memory:

```
Person firstObject = new Person(10);
Person secondObject = new Person(50);
secondObject = firstObject;
```

Let's start playing with value types and reference types using a simple example. We use two variables of type `int` and two objects of type `Person`, and we apply the same actions on them. However, in the end the results are not the same:

```
using System;

class MyExecutableClass
{
   static void Main(string[] args)
   {
      int firstValue = 10;
      Person firstObject = new Person(10);

      int secondValue = firstValue;
      Person secondObject = firstObject;

      secondValue = 20;
      secondObject.Age = 20;

      Console.WriteLine("firstValue = " + firstValue);
      Console.WriteLine("secondValue = " + secondValue);

      Console.WriteLine("firstObject.Age = " + firstObject.Age);
      Console.WriteLine("secondObject.Age = " + secondObject.Age);
   }
}

class Person
{
   public int Age;

   public Person(int Age)
   {
      this.Age = Age;
   }
}
```

Here's the output:

```
firstValue = 10
secondValue = 20
firstObject.Age = 20
secondObject.Age = 20
Press any key to continue
```

The key to understanding this program's output is to analyze what happens in these two lines:

```
      int secondValue = firstValue;
      Person secondObject = firstObject;
```

Here we declare a new `int` variable named `secondValue`, which will initially have the same value as `firstValue`. Still they represent separate values on the stack. When we declare the second object, we are assigned a place on the stack for it. We populate that place with a reference to the object named `firstObject`, so in the end we'll have two references, `firstObject` and `secondObject` that will both reference the same object on the heap. From now on, any change made to the members of `firstObject` will also affect `secondObject`, and vice-versa.

Passing Parameters by Value and by Reference

Both value types and reference types can be passed as method parameters either by value, or by reference. Visual Basic 6 developers are familiar with the concepts of passing by value and by reference – Visual Basic 6 even has the `ByVal` and `ByRef` keywords that can be used on method parameters to specify how they should be passed.

> **In C# all method parameters, regardless of whether they are value types or reference types, are passed by value by default. Visual Basic 6 passes parameters by reference by default.**

But passing by value for value types has a different meaning from passing by value for reference types. For more clarification, let's examine each scenario, one at a time.

Passing a Value Type by Value

When a variable is passed as a method parameter by value, a copy of the data is transferred to that method. Any modification to the data in the method will not have any effect on the original value.

Take a look at this short Visual Basic program:

```
Private Sub Form_Load()
    Dim value As Integer
    value = 50
    DoSomething value
    MsgBox value
End Sub

Public Sub DoSomething(parameter As Integer)
    parameter = 100
End Sub
```

When executed, this program would show a message box containing the value 100. This demonstrates that changing the value in `DoSomething` method has effect on the original variable defined in `Form_Load`. This means that the value was sent by *reference*. Now change the definition of `DoSomething` like this:

```
Public Sub DoSomething(ByVal parameter As Integer)
```

The message would now contain the value of 50. This time the parameter was sent by value, meaning that a copy of the value was sent to the subroutine, not a reference to the original variable.

By default, Visual Basic 6 passes parameters by reference and C# passes by value. To demonstrate this difference, let's translate the code to C#:

```
using System;

class MyExecutableClass
{
   static void Main(string[] args)
   {
      int value = 50;
      DoSomething (value);
      Console.WriteLine (value);
   }

   static void DoSomething(int parameter)
   {
      parameter = 100;
   }
}
```

The program's output (50) confirms that the variable is sent by value, not by reference. We didn't need to add any special keyword – we relied on the default behavior of C# passing a value type by value.

Passing a Value Type by Reference

Passing a value type by reference is, in other words, *passing the reference of the value (variable)*. If we pass the reference, any change on the value using the reference is persisted no matter where in the program we are making the change (could be in another method, property, or even another object). Any change we make to the parameter inside the method will affect the value after the method's return.

In C#, passing by reference is accomplished by using the `ref` keyword. The `ref` keyword must be added to the method definition, like this:

```
   static void DoSomething(ref int parameter)
```

We must use the `ref` keyword when passing the parameter as well:

```
      DoSomething (ref value);
```

Here's the complete code that demonstrates using `ref` parameters for value types:

```
using System;

class MyExecutableClass
{
```

```
    static void Main(string[] args)
    {
       int value = 50;
       DoSomething (ref value);
       Console.WriteLine (value);
    }

    static void DoSomething(ref int parameter)
    {
       parameter = 100;
    }
}
```

The output this time is 100, as expected.

Passing a Reference Type by Value

After we learned what it means to pass a value type by reference or by value, it may be pretty confusing to think about passing a reference type by value, or worse yet – by reference.

Generally, passing by value means passing the value that is stored on the stack. For reference types, we know that the value that is stored on the stack represents a reference to the actual location of the object in memory. So, if we pass a reference type by value, it means that we pass the object's reference (its stack value). Any modification done using that reference, will finally still modify the same object stored on the heap.

So, passing reference types by value is nothing like passing value types by value – it is more like passing value types by reference. In the code example, we use `Person` as the reference type:

```
using System;

class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person person = new Person(50);
      DoSomething (person);
      Console.WriteLine (person.Age);
   }

   static void DoSomething(Person somePerson)
   {
      somePerson.Age = 100;
   }
}

class Person
{
   public int Age;
```

```
    public Person(int Age)
    {
       this.Age = Age;
    }
}
```

The output shows 100, exactly what we expect. However, now try changing the `DoSomething()` method to this:

```
    static void DoSomething(Person somePerson)
    {
      somePerson = new Person (100);
    }
```

When you rerun the program, the result is again 50. But why? Didn't we send the reference to the method?

The answer is: yes, we did send the reference. This is great as long as we don't change the reference. Remember that reference types have two elements to them – the reference, and the object. Now, when we called the `DoSomething()` method we created a *copy* of the reference. It still pointed to the same object, so changes to the object did affect the main program. But changes to the reference do not – when the method finishes, the copy of the reference simply disappears.

When we use the `somePerson.Age` property to change the age, we are changing the object itself. But in the second attempt we create a *new* object, and change the reference to point to it – this change to the reference will be lost.

So what should be done? You guessed it: the solution is to send the reference type by reference. By doing that, if we change the reference stored by `somePerson`, there is another "parent" reference that is automatically updated. Sounds confusing? Let's look at the idea in more detail.

Passing a Reference Type by Reference

This is a nice one. We learned that when passing a reference type by value, we pass a reference to the object's location in memory. Now, when passing a reference type by reference, we pass a reference to the reference.

As we've seen, passing reference types by value does not work in all situations – particularly when we need to change the reference to point to a new object.

Here is an example where passing by reference is useful:

```
class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person person = new Person(50);
      DoSomething (ref person);
      Console.WriteLine (person.Age);
```

```
        }

        static void DoSomething(ref Person somePerson)
        {
            somePerson = new Person(100);
        }
    }
```

This time, the program's output will be 100, which is what we want. Let's take a look at what's going on here. The `person` variable is really a reference to a `Person` object that is on the heap. When we call `DoSomething()`, the compiler creates a reference to the person reference (rather than a reference to the `Person` object that it points to). Within the `DoSomething()` method, then, `somePerson` is a reference to the `Person` reference, which in turn references the object on the heap. However, `DoSomething()` *knows* that the value was passed by reference – and therefore any changes made to `somePerson` are really made to `person`. The practical upshot of this is that `somePerson` behaves as if it were the `person` reference, not a copy of it.

So far we've concentrated on designing classes and creating objects. We've seen how objects are reference types, and primitive data types are value types. However, C# enables us to create our own value types called structs, as we will see now.

Structs

A struct is a user-defined value data type very similar to the class – it can contain constructors, fields, methods, and properties. Structs are declared using the `struct` keyword instead of `class`.

Let's examine the most important differences between structs and classes:

- The struct is a value type, while classes are reference types.
- Inheritance doesn't work with structs: a struct cannot derive from a class or another struct; a class cannot derive from a struct. All structs are implicitly derived from `System.ValueType` (which in turn is derived from `System.Object` – we'll see more details about this a bit later in this chapter).
- Structs always contain a parameterless, default constructor. You are allowed to add more overloads, but you can't add a parameterless constructor.
- You are not allowed to provide default values for structs' fields.
- You don't have to create a new struct using the `new` keyword (because the struct is a value type). If you don't use `new`, the default, parameterless constructor is called when creating the struct. However, you'll need to use `new` keyword if you want to supply parameters to a struct's constructor, just like you do for classes.

Although structs are very powerful, they are mainly designed to act as containers for data rather than as full-featured objects. Because structs are value types, they are stored on the stack. We didn't discuss in depth the difference between the stack (where value types are stored) and the managed heap (where reference types are stored), but keep in mind that storing on the stack can be very fast if you're working with small amounts of data. MSDN says that data structures smaller than 16 bytes may be handled more efficiently as structs rather than as classes.

To demonstrate using a struct, we'll again use the example program from the *Passing a Reference Type by Value* section. This is the program:

```
using System;

class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person person = new Person(50);
      DoSomething (person);
      Console.WriteLine (person.Age);
   }

   static void DoSomething(Person SomePerson)
   {
      SomePerson.Age = 100;
   }
}

class Person
{
   public int Age;

   public Person(int Age)
   {
      this.Age = Age;
   }
}
```

Remember that running this program resulted with the value 100 printed on the screen. Now instead of a class use a struct:

```
struct Person
```

That's the only change you need to do! Rerun the program and admire the new output that shows 50. This simple but interesting experiment shows us that structs are indeed value types, not reference types.

It's also very simple to test all the other properties of the struct. For example, try to add a parameterless constructor in the `Person` struct:

```
   public Person()
   {
      Age = 0;
   }
```

The error generated by the compiler is: Structs cannot contain explicit parameterless constructors.

Another restriction about structs is that they are not allowed to have initialized fields. So change the `Person` struct like this:

```
      public int Age = 0;
```

The error message is: 'Person.Age': cannot have instance field initializers in structs.

Of course, trying to initialize `Age` to zero would be completely useless, as it is automatically initialized to zero anyway. But as you can see, you must use a parameterized constructor in order to initialize the fields to some other values than their default ones. As a final experiment, you can even make `MyExecutableClass` into a struct:

```
   struct MyExecutableClass
```

Execute the program, and you'll see that it works just as before!

Enums

Enumerations are user-defined value data types used to store a set of named constants that are all of the same data type. Enumerations were present in Visual Basic 6, and they are also present in C#. Also, like in Visual Basic 6, we use the `enum` keyword to declare an enumeration.

The `enum` element's data type can be any of the predefined integral data types (signed or unsigned, 8-, 16-, 32- or 64-bit integer data types). The default enumeration's data type is `int`, and the default value for the first element is 0.

Take a look at this example, where we have upgraded the `Person` struct (you can make it a class if you want, it doesn't matter) to use enums:

```
using System;

public enum Age {From13to18, From18to25, From25to40, From41to65};
public enum Sex {Female, Male};

class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person person = new Person(Age.From41to65, Sex.Female);

      Console.WriteLine("Age (string): " + person.Age);
      Console.WriteLine("Age (int)   : " + (int)person.Age);

      Console.WriteLine("Sex (string): " + person.Sex);
      Console.WriteLine("Sex (int)   : " + (int)person.Sex);
   }
}
```

```
struct Person
{
   public Age Age;
   public Sex Sex;

   public Person(Age age, Sex sex)
   {
      this.Age = age;
      this.Sex = sex;
   }
}
```

The output shows that an `enum` element knows both its string representation and its internal `int` value:

```
Age (string): From41to65
Age (int)   : 3
Sex (string): Female
Sex (int)   : 0
Press any key to continue
```

We don't have to use an `enum` element's internal value in our program – although the values are internally stored as numeric values, we typically only use their string representation:

```
        Person person = new Person(Age.From41to65, Sex.Female);
```

The program's output also shows that the default value for the first element of the enum is zero, and for each following element it is increased by one. We can manually set the value of an element like this:

```
public enum Age {From13to18 = 10, From18to25, From25to40, From41to65};
```

In this case, the output will be:

```
Age (string): From41to65
Age (int)   : 13
Sex (string): Female
Sex (int)   : 0
Press any key to continue
```

We can choose the data type used for each `enum` value like this:

```
public enum Sex: byte {Female, Male};
```

Objects in the .NET Framework

If this is the first book where you read about object-oriented programming, by now you have probably started dreaming about objects – don't worry, the effect of first exposure to objects doesn't usually last for more than a week. Even though this is yet another section about objects, it won't be a boring one – moreover, we'll cover some important concepts that every serious .NET programmer must know.

So far we have covered various concepts that apply in one way or the other to almost any truly object-oriented language. Every language has its particularities – for example, in contrast to C#, C++ supports multiple inheritance and has predictive object destruction; class members in Java are virtual by default, unlike C#'s members that must be explicitly declared as virtual. Despite some differences like these, the general concepts apply the same way in each of these languages.

You have probably already heard that 'everything is an object'. Please allow me to be original now: seriously, *everything is an object*. This has two meanings: first of all, in C# every program consists of a class. Even an executable program is implemented as a class that has a `Main()` method. In all stages of developing an application, from design to implementation, decisions must be made in regard to the way we design and relate objects and classes to each other. Yes, objects are everywhere.

.NET extends this to yet another level, giving the phrase 'everything is an object' an additional meaning: in the world of .NET, every class ultimately derives from a base class named `System.Object`, so 'everything is an object' becomes 'everything is an `Object`'.

C# and .NET support single inheritance, meaning that every object directly derives from exactly one other object. If we write a class and don't explicitly derive it from another class, it will automatically inherit from `System.Object`.

If you're not convinced yet, start Visual Studio .NET documentation or Microsoft .NET SDK documentation, and go to `System.Windows.Forms.Form` class (or, simply, *Form class*). `Form` is a class we've already used, as it is the base class used in .NET by Windows Forms. Unlike in Visual Basic 6 where Forms are managed by the IDE, in .NET a form is simply a class that derives from `Form`. Take a look at `System.Windows.Forms`.

Anyway, once you've looked it up click on Form Overview. You'll be presented a page containing this hierarchy of classes:

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.Control
            System.Windows.Forms.ScrollableControl
               System.Windows.Forms.ContainerControl
                  System.Windows.Forms.Form
                     System.ComponentModel.Design.CollectionEditor.CollectionForm
                     System.Web.UI.Design.WebControls.CalendarAutoFormatDialog
                     System.Windows.Forms.Design.ComponentEditorForm
                     System.Windows.Forms.PrintPreviewDialog
```

Don't worry, it's not important to know what every class in this list does. Actually, there are quite a few classes that you'll use directly and often in your programs. Most of them are used in quite rare situations. Anyway, the point is to observe that at the base of the whole hierarchy is `System.Object`.

Another important class you should know about is `System.ValueType`. Value types, including hand-made structs and predefined types like `int` and `char`, implicitly derive from a class named `System.ValueType`, which in turn derive from `System.Object`. We know that in C# `int` is an alias for `System.Int32`. MSDN reveals the inheritance hierarchy for this object:

```
System.Object
   System.ValueType
      System.Int32
```

`System.ValueType` overrides some of `Object`'s functionality, in order to make it more appropriate for value types. This is required since `Object` by itself is mainly designed to serve as a base class for reference types.

`object` is C#'s alias for `System.Object`. Since `Object` is so important that every other class derives from it, either directly or indirectly, it deserves a closer look.

System.Object Members

Analyzing the `System.Object` members is very important, because all its public members automatically become part of each class's public interface. Since every class derives from `Object`, we can access each object's public member on any class.

The designers of .NET decided that there is a minimal functionality that must exist in every entity that we call an object. Understanding this basic functionality that exists in each object in the .NET world helps us understand that vision, and program in the .NET spirit.

In the next sections, we'll cover the `System.Object` public members:

- `Equals()`
- `ReferenceEquals()`
- `GetHashCode()`
- `GetType()`
- `ToString()`

Let's see what each of them has in store for us.

Equals()

`Equals()` returns a Boolean value that specifies if two object instances are equal. The method comes in two flavors: a static version, and an instance version. Let's see an example that uses both:

```
class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person first = new Person(5);
      Person second = new Person(5);
      Person third = first;

      Console.Write("first = second: ");
      Console.WriteLine (first.Equals(second));

      Console.Write("first = third: ");
      Console.WriteLine (Object.Equals(first,third));
   }
}

class Person
{
   public int Age;

   public Person(int age)
   {
      this.Age = age;
   }
}
```

The output is:

```
first = second: False
first = third: True
Press any key to continue
```

In this example we have used both the instance version of `Equals()`:

```
      Console.WriteLine (first.Equals(second));
```

… and the static overload:

```
      Console.WriteLine (Object.Equals(first,third));
```

Keep in mind that for value types the value is compared, while for reference types the reference is compared. The instance version of `Equals()` is declared as virtual, so we can override it in our own classes. You may want to do this in your classes if you want to compare internal values rather than the instance – to make them behave just like it does for value types.

In our example, both `first` and `second` objects are initialized with the same value:

```
        Person first = new Person(5);
        Person second = new Person(5);
```

Still, they are different objects, located in different memory locations. Because `Person` is a reference type, `Equals()` evaluates the equality of object's reference, not their internal values, so the result is that they are note equal:

```
first = second: False
```

However, if we "transform" our `Person` class to a value type, by making it a `struct`, the result changes:

```
first = second: True
first = third: True
Press any key to continue
```

This demonstrates that for value types, `Equals()` compares their value rather than their particular instance, or in other words their location in memory.

We can override `Equals()` in our classes to make them behave like value types (at least as far as the `Equals()` method is concerned). Let's override `Equals()` in our `Person` class. This is perfectly possible, since `Person` derives from `System.Object`, and the instance implementation of `Equals()` in `System.Object` is `virtual`:

```
public virtual bool Equals(object);
```

Since we have the complete method's definition, it is easy to override it. Modify `Person` like this:

```
class Person
{
  public int Age;

  public override bool Equals(object other)
  {
    if (other == null) return false;
    if (GetType() == other.GetType())
    {
      Person otherPerson = (Person) other;
      return (otherPerson.Age == this.Age);
    }
    return false;
  }

  public Person(int age)
  {
    this.Age = age;
  }
}
```

We have overridden the `Equals()` method to compare the `Age` value. If the `Person` objects being compared have the same `Age`, the method returns `true`. Otherwise, it returns `false`. Rerun the program, and you'll see indeed that the output changes, showing that the objects are equal now:

```
first = second: True
first = third: True
Press any key to continue
```

Note that when you compile the program, you receive the following warning: 'Person' overrides Object.Equals(object o) but does not override Object.GetHashCode(). This warning appears because when you override `Object.Equals()`, sometimes you should also override `Object.GetHashCode()`, so that when `Object.Equals()` returns `true`, `Object.GetHashCode()` should also return `true`. You can read more details in the *GetHashCode()* section.

ReferenceEquals()

`ReferenceEquals()` is also used to test for equality, but it always compares the reference – if we supply objects or variables located at different places in memory, they will not be seen as equal. For reference types, their location on the heap is compared; for value types, their location on the stack is compared. The consequence is that two value type objects (like two `structs` or two `ints`) will always be different, while two reference type objects will be equal if they both point to the same location on the heap.

`ReferenceEquals()` is supplied only as a static method of `Object`. The most important difference between `Equals()` and `ReferenceEquals()` you'll notice in this example will be about value types, which are not reported as being equal – when using `Equals()`, value types with the same value are always equal.

```
using System;

class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person first = new Person(5);
      Person second = new Person(5);
      Person third = first;

      Console.Write("first = second: ");
      Console.WriteLine (Object.ReferenceEquals(first,second));

      Console.Write("first = third: ");
      Console.WriteLine (Object.ReferenceEquals(first,third));
   }
}

class Person
{
   public int Age;

   public Person(int Age)
   {
      this.Age = Age;
   }
}
```

When `Person` is a class, the result is:

```
first = second: False
first = third: True
Press any key to continue
```

When `Person` is a struct, the result is:

```
first = second: False
first = third: False
Press any key to continue
```

GetHashCode()

`GetHashCode()` returns the hash value of a specific object. The hash code of an object is an integer value that ideally would uniquely identify that object. We will not cover the details of hash codes in this book – for now, the important thing to keep in mind is that if two objects are equal, then their hash codes will be identical. The reverse is not necessarily true – two objects with the same hash code are not necessarily identical, but the probability of them being identical is very high.

For this reason, when we override `Equals()` in our class, we should also override `GetHashCode()` in order to make sure the rules are still followed. For a demonstration, use this class:

```
class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person first = new Person(5);
      Person second = new Person(5);
      Person third = first;

      Console.Write("Hash Code for first: ");
      Console.WriteLine (first.GetHashCode());

      Console.Write("Hash Code for second: ");
      Console.WriteLine (second.GetHashCode());

      Console.Write("Hash Code for third: ");
      Console.WriteLine (third.GetHashCode());
   }
}
```

If `Person` is a class, the results will be:

```
Hash Code for first: 3
Hash Code for second: 5
Hash Code for third: 3
Press any key to continue
```

If `Person` is a struct, the output is:

```
Hash Code for first: 5
Hash Code for second: 5
Hash Code for third: 5
Press any key to continue
```

GetType()

`GetType()` returns an object of `System.Type` type, that contains information about our object. We met `GetType()` in Chapter 3, where we obtained an object's type as a string representation. Keep in mind that `System.Type` provides us with very powerful techniques to programmatically get detailed data about an object's type – in .NET this is called *reflection*. As a hint, think about Microsoft's IntelliSense technology: Visual Studio is capable of providing us with hints about a specific object by getting its type information. However, this is considered an advanced topic, and we won't be covering reflection in this book.

This is how the type of an object can be found out:

```
        Person first = new Person(5);
        Console.WriteLine("Type of first: " + first.GetType());
```

The output is: Type of first: Person.

ToString()

This is probably the "simplest" member of the `Object` data type, and you'll find it very useful in your programs. `ToString()` is used to return the text representation of a class. `Object`'s implementation of `ToString()` returns the class name on which the method is called. For example, this piece of code will write Person on the screen:

```
      static void Main(string[] args)
      {
         Person obj = new Person(5);
         Console.WriteLine(obj.ToString());
      }
```

It is particularly useful to override `ToString()` because there are certain situations when it is automatically called. Take a look a this piece of code:

```
      static void Main(string[] args)
      {
         Person obj = new Person(5);
         Console.WriteLine("obj is " + obj);
      }
```

Because of C#'s type safety, the second line should never compile, because we try to add a string with a `Person`. But in fact, the program compiles and works very well, and the output is obj is Person. When we try to add a string with a non-string type, the non-string type's `ToString()` method is automatically called, and this explains the output of the previous code snip. This is another way of obtaining the same result:

```
static void Main(string[] args)
{
   Person obj = new Person(5);
   string strA = "obj is ";
   string strB = strA + obj;
   Console.WriteLine(strB);
}
```

For basic data types like `int`, `ToString()` returns the string representation of the number. Take a look a this short example:

```
static void Main(string[] args)
{
   int a = 5;
   string b = " + ";
   int c = 5;
   string d = " = 10";

   Console.WriteLine(a+b+c+d);
}
```

Thanks to the `ToString()` method, it is possible to "add" an integer with a string – this behavior might seem normal to you, the Visual Basic 6 developer, but as you learned in Chapter 3, in reality C# is very strict about type casts.

Boxing and Unboxing

With the process called *boxing*, a value type can be transformed to a reference type. Unboxing is the reverse operation.

Boxing

We know by now that all C# types derive from `System.Object`. While it's pretty obvious how this works for classes, it might be a mystery for you how it's possible to call a method on a basic data type. Try to execute the following line of code, and you'll see that the output is `True`:

```
Console.WriteLine(true.ToString());
```

So, the `ToString()` method is overloaded in `System.Boolean`, and it returns "`True`" for true values and "`False`" otherwise.

We know that reference types are stored on the heap, while value types are stored on the stack (unless they are fields of a reference type). We know that the performance of value types is very good, especially for basic data types that have direct support in the CPU. Basic data types that are stored on the stack occupy the exact size in bytes of the data type. For example, an `int` in C# occupies 32 bits (4 bytes), while a short takes exactly 16 bits (2 bytes).

So how can we call methods on a variable? The answer lies in a technique called boxing. The previous example where we typed the output of `true.ToString()` is a typical example of boxing. Let's see another example:

```
static void Main(string[] args)
{
   int i = 5;
   object o = i;
   Console.WriteLine(o.ToString());
}
```

So we store the variable `i` in an object of the reference type `object`. Then, using the object reference, we're able to call any of its members, like the `ToString()` method. These are in fact the steps that happen in the background in the following example, which has exactly the same effect:

```
static void Main(string[] args)
{
   Console.WriteLine(5.ToString());
}
```

It is important to understand that in the process of boxing, the object does not hold a reference to the original variable. In fact, it couldn't, because the original variable is stored on the stack, while the object is stored on the heap. So the original variable and the object used to box it are two independent entities. Here's an example that demonstrates this theory:

```
static void Main(string[] args)
{
   int i = 5;      // original value of i is 5
   object o = i;   // original value in o is 5
   i = 10;      // does not affect the value in o
   o = 20;      // does not affect the value in i

   Console.WriteLine("i = " + i);    // i = 10
   Console.WriteLine("o = " + o);   // o = 20
}
```

Unboxing

Unboxing refers to casting from a reference type to a value type. This is a typical example for unboxing:

```
   int i = 5;
   object o = i;   // boxing
   int j = (int) i;   // unboxing
```

If the boxed value cannot be stored in the destination variable, an `InvalidCastException` is generated:

```
static void Main(string[] args)
{
   try
   {
      float i = 5;
      object o = i;      // boxing
      int j = (int) o;   // unboxing
      Console.WriteLine("j = " + j);
   }
   catch (Exception e)
   {
      Console.WriteLine(e.ToString());
   }
}
```

When you execute the code above, the text of the error is typed to the screen:
System.InvalidCastException: Specified cast is not valid.

Boxing to ValueType

It is possible to cast any value type (or any other type) to `Object`, because all type derives from `Object`. If you remember the section about polymorphism well, you know that we can cast any object to any base class.

Since all value types derive from `System.ValueType`, it's also possible to "box" your values to `System.ValueType` objects, not necessarily to `System.Object` objects. Although I'm not sure that you'll actually need to do this in your programs, it's good to know that this is possible. Here's a short example:

```
static void Main(string[] args)
{
   int i = 5;
   ValueType vt = i;
   Console.WriteLine(vt);
}
```

Summary

Well, that's the end of our two-part journey into object-oriented programming. We have built on the basics of object-oriented programming that we learned in the previous chapter, and looked at more advanced topics of object-oriented programming, and at how these features are implemented in C#.

We've covered a massive amount of ground in this chapter, and we've seen our simple `Person` object inherit, polymorph, hide, override and generally be manipulated in all sorts of interesting ways – ways that would have required extensive amounts of code and sizeable alteration to existing code without the power of object-oriented programming.

You should now feel comfortable with object-oriented concepts and terms – we've not just covered definitions and theory in this chapter, we've cemented the concepts with real code examples.

The topics we looked at in this chapter included:

- Method and property hiding, and overriding, and the difference between these concepts
- Static class members and static constructors
- Abstract classes, interfaces, and interface inheritance
- Destructors
- Delegates and events
- Constants and `readonly` fields
- Value types and reference types
- Passing value and reference types either by value or by reference
- Objects in .NET – `System.Object`
- Boxing and unboxing

That brings us to the end of our voyage into the C# language and object-oriented programming, for now. As we move on to the next chapter, and begin to construct a more sizeable Windows application to which we will add further features over the following three chapters, we will continue to learn more about the C# language, and see the concepts that we have learned about in these two chapters in action.