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

4

Object-Oriented Programming : Part I

Object-Oriented Programming (OOP) is a very important and fundamental topic in the .NET world. Visual Basic 6 supports basic programming using classes and objects, but it lacks support for many of the features that make object-oriented programming so powerful. For example, VB6 does not support implementation inheritance, method overloading and overriding, and constructors.

Unlike VB6, C# moves object orientation to the center of development. OOP is a crucial skill for the successful C# developer. In this chapter and the following one, we will examine the basic skills and concepts needed to write quality code with C#. We will apply many of these ideas to the applications we build in the rest of the book.

In this chapter, the first of two chapters on object-oriented programming, we will cover the fundamental principles of OOP, and how to apply these principles to simple C# programs. More specifically, we will cover these topics:

- Objects – the fundamental concept of OOP
- Objects and Classes
- Fields, Methods, and Properties
- Encapsulation
- Inheritance
- Polymorphism
- Method Overloading
- Constructors

This chapter we will not touch fields specific to C# or even to the .NET Framework. Every concept presented here can be applied to any other object-oriented language, such as Java or Visual Basic .NET – although the syntax will be different.

Objects and Classes

So what does "object-oriented programming" mean? Basically, as the name suggests, it puts objects at the center of the programming model. The **object** is probably the most important concept in the world of OOP – a self-contained entity that *has state and behavior*, just like a real-world object.

In programming, an object's state is described by its fields and properties, and its behavior is defined by its methods and events. An important part of OOP's strength comes from the natural way it allows programmers to conceive and design their applications.

We often use objects in our programs to describe real-world objects – we can have objects of type `Car`, `Customer`, `Document`, or `Person`. Each object has its own state and behavior.

It is very important to have a clear understanding of the difference between a class and an object. A class acts like a blueprint for the object, and an object represents an instance of the class. In the case of objects of type `Car`, for example, `Car` is the class or type. We can create as many `Car` objects as we want, and give them names such as `myCar`, `johnsCar`, or `janesCar`.

So you get the idea – the class (also called the type) defines the behavior attributes that will apply to all objects of that type. All objects of type `Car` will have the same behavior – for example, the ability to change gear. However, each individual `Car` object may be in a different gear at any particular time – each object has its particular state.

Take a look at the following C# code:

```
int age;
string name;

Car myCar;
Car myOtherCar;
```

As you can see, the syntax for creating an object is the same as for creating a simple integer or string variable – first we mention the type, and then we name that particular instance.

Introducing the Person Class

The vast majority of the concepts presented in this chapter are demonstrated using a very simple class named `Person`. There is no single version of the class, as we'll rewrite and modify it all along the chapter. When introducing each new topic, we will only keep a minimal implementation of the class and drop the unimportant details that remained from the previous example. You'll see, we'll have so much fun with this class that you'll miss it once the OOP chapters are over.

As you already know from the previous chapters, you don't need to work with Visual Studio in order to compile a C# program. Notepad and the command-line compiler will do just fine, especially that in this chapter we don't rely much on code that would have been automatically generated for us by the IDE.

However, we will continue to use Visual Studio .NET for the examples, but we will be using a Console Application instead of a Windows Application. Start Visual Studio .NET, go to the File menu, and select New Project.

In the New Project dialog box, select Visual C# Projects as the project type, and then the Console Application template. Console Applications are text–mode programs, and are often very useful as a way of testing classes and objects quickly, because we don't need to worry about setting up windows and components on the screen – we can just write text straight to a Command Prompt window.

The location of the project is `D:\Projects` in my case, but you can choose another location if you like. Type `PersonExample` for the project's name, and click OK.

When creating a new console application, Visual Studio automatically generates a file named `Class1.cs`, which looks like this:

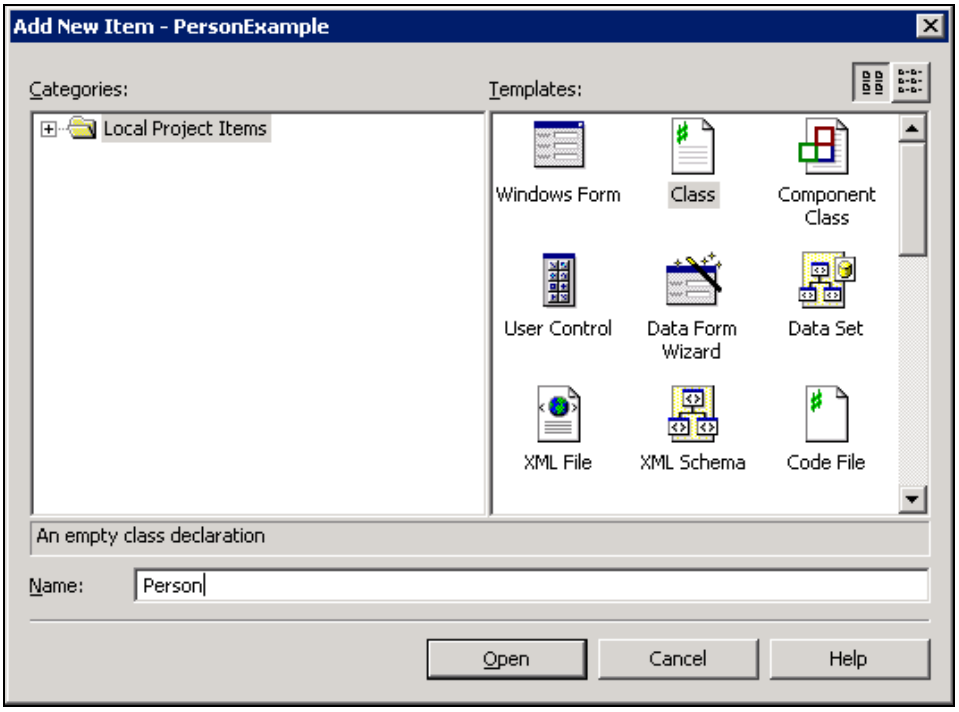
```
using System;

namespace PersonExample
{
  /// <summary>
  /// Summary description for Class1.
  /// </summary>
  class Class1
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
      //
      // TODO: Add code to start application here
      //
    }
  }
}
```

`Class1.cs` contains a class named `Class1`, located in the `PersonExample` namespace. Note that `Class1` has an entry point defined (the `Main` method). This means that our program is ready to be compiled and executed.

However, we want to create our own class called `Person`. This class does not need to be executable in its own right – we will be using it within our executable class.

Adding a Class

To add the `Person` class, select Add New Item from the File menu. A new dialog will appear asking us to choose a template and name for the new file:



Select the Class icon, and in the Name box type Person or Person.cs. Click Open, and a new file named `Person.cs` is added to the project – containing a single class named `Person`:

```
using System;

namespace PersonExample
{
  /// <summary>
  /// Summary description for Person.
  /// </summary>
  public class Person
  {
    public Person()
    {
      //
      // TODO: Add constructor logic here
      //
    }
  }
}
```

Coding the Person Class

This class is different from `Class1` because it has no `Main()` method, but does have a `Person` method. This `Person` method has no return type – not even `void`. This method is the class's constructor. Constructors are a special type of method that are very useful in OO programming. We will look at how to use them later in the chapter.

For now, let's modify the `Person` class by removing the constructor, and adding two public fields:

```
using System;

namespace PersonExample
{
  public class Person
  {
    public int Age;
    public string Name;
  }
}
```

Our `Person` class is very simple and has two fields, called `Age` and `Name`. They are marked with the `public` access modifier, which simply specifies that the two fields can be accessed anywhere that uses objects of this class. We'll take a better look at access modifiers a bit later in this chapter.

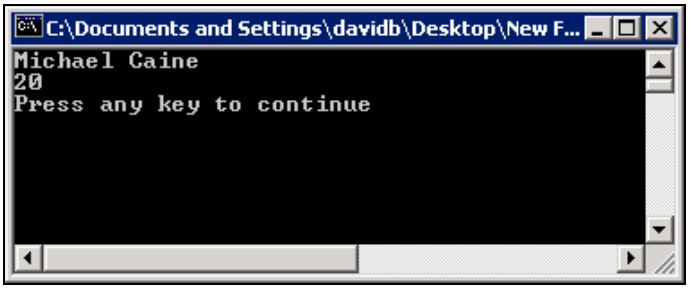
Using the Person Class

In order to use the `Person` class, we need a program that will create `Person` objects. Since `Class1` has a `Main()` method, we can use this. However, `Class1.cs` isn't a good name, so in the Solution Explorer right-click `Class1.cs`, select Rename, and change the name to `MyExecutableClass.cs`

Now double-click this file, and modify the `Main()` method to read as follows. Also change the class name to `MyExecutableClass`, to match the file name:

```
using System;

namespace PersonExample
{
  class MyExecutableClass
  {
    [STAThread]
    static void Main(string[] args)
    {
      Person mike;
      mike = new Person();
      mike.Age = 20;
      mike.Name = "Michael Caine";

      System.Console.WriteLine (mike.Name + "\n" + mike.Age);
    }
  }
}
```

Now we can build and execute the program. If we use *Ctrl-F5* (shortcut for Start Without Debugging), we'll see the following output:



The result is the same if we execute the program with *F5* (shortcut for Start), but the window will close immediately after execution – we won't see the Press any key to continue message. If we wanted to start with debugging, but still wanted to keep the window open at the end, we could add `Console.ReadLine();` to the end of the `Main()` method. This will mean that the program will hold until we press *Enter*.

Let's take a look at how this `Main` method works. First, we declared a new object named `mike`, of type `Person`:

```
Person mike; // declare object of type Person
```

Because `Person` is not a simple type like `integer` or `bool`, we must manually instantiate it. This is the step when the object is actually created in memory:

```
mike = new Person(); // instantiate object
```

Like Visual Basic 6, C# would have allowed us to declare and instantiate the object in a single step:

```
Person Mike = new Person(); // declare and instantiate object
```

We will explain why this is needed with more technical details in the next chapter, when we'll talk about value types and reference types. For now, keep in mind that creating an object involves one more step than creating a simple variable, just as it was in Visual Basic 6. This is the equivalent Visual Basic 6 code:

```
Dim mike As Person      ' declare
Set mike = New Person   ' instantiate
' or
Dim mike As New Person  ' declare and instantiate
```

After we have created our object, we can assign properties to its members using the familiar *object.member* notation:

```
mike.Age = 20;
mike.Name = "Michael Caine";
```

We use exactly the same syntax to read field values from the object:

```
Console.WriteLine (mike.Name + "\n" + mike.Age);
```

Adding Behavior to the Person Class

We learned earlier that an object encapsulates both state (instance data) and behavior. In the `Person` class the `Age` and `Name` fields are instance data, but the class doesn't have any behavior yet. Behavior is implemented with methods, so let's add a simple method to our class.

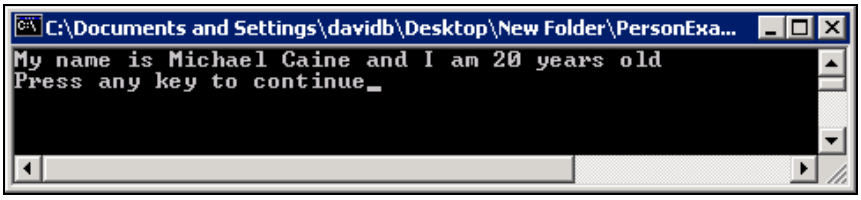
Update the `Person` class like this:

```
namespace PersonExample
{
  public class Person
  {
    public int Age;
    public string Name;

    public void DescribeYourself()
    {
      Console.WriteLine("My name is {0} and I am {1} years old", Name, Age);
    }
  }
}
```

Now, let's update the `Main` method of `MyExecutableClass` so that it calls `DescribeYourself` instead of requesting the name and the age separately:

```
  public class MyExecutableClass
  {
    static void Main(string[] args)
    {
      Person mike;
      mike = new Person();
      mike.Age = 20;
      mike.Name = "Michael Caine";

      mike.DescribeYourself();
    }
  }
```

Press *Ctrl-F5* to execute the program, and see the new output:



By adding some functionality, we have just made our `Person` class smarter – now the caller program, `MyExecutableClass`, can rely on the behavior we implemented in the `Person` class, instead of displaying the name and age separately.

This is a very short example that demonstrates how using OOP can improve maintainability of your programs. If we had many other programs using objects of type `Person`, updating the `Person` class (perhaps to make the description it gives even more polite) would automatically update the functionality of all the programs that used it.

Access Modifiers

If we look at the `Person` class, we can see that all its members are marked with the public access modifier:

```
public int Age;
public string Name;

public void DescribeYourself()
```

This means that other classes that use the `Person` class have full access to these members. In `MyExecutableClass` we stored values to `Age` and `Name` fields, and we were able to call the `DescribeYourself` method:

```
mike.Age = 20;
mike.Name = "Michael";
mike.DescribeYourself();
```

The other access modifier we'll take a more detailed look at now is `private`. A `private` class member is only accessible inside the class in which it is defined; for all the other classes, it is invisible.

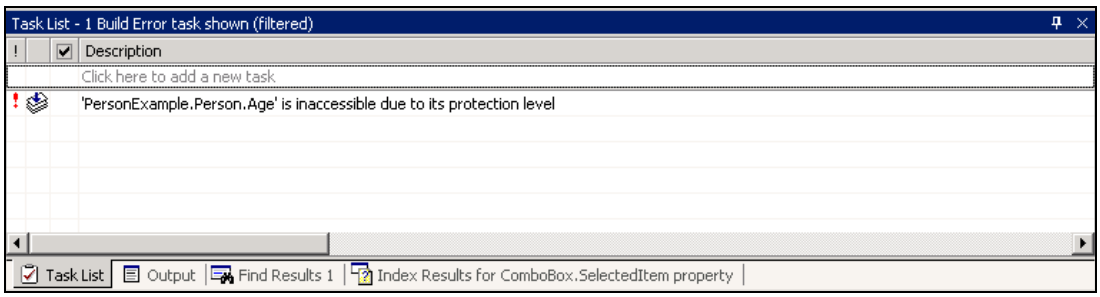
Deciding what access modifier to use for a particular class member can be a very difficult decision to make because it affects not only your class, but also the other classes and programs that use your class. Of special importance are the class's public members, which together form the class's **public interface**. The public interface acts like a contract between your class and the users of your class. If it is designed properly, it should not change over time. If, for example, you mark `DescribeYourself` as `public`, and later decide to make it `private`, all other classes that call this method will have to change accordingly. As an extreme scenario, imagine that in a year Microsoft decided to remove the `WriteLine()` method from the `Console` class – you wouldn't like this very much, would you?

As a simple guideline for designing you class, keep in mind that sometimes it's easier just to make all the members private, and make public only the ones that really need to be public.

Let's go back to our `Person` class now. To demonstrate how private members work, please change the access modifier of `Age` as shown:

```
        private int Age;
```

Now press *Ctrl-Shift-B* to build the program. As expected, the compiler will generate an error regarding this field, because it is no longer accessible from `MyExecutableClass`. We can see the error both in the Output Window (*Ctrl-Alt-O*), and in the Task List (*Ctrl-Alt-K*). Here's a screenshot of the Task List containing the error:



By double-clicking on the error in the Task List window, Visual Studio .NET will automatically select the problematic line, which in our case is the line in `MyExecutableClass` that accesses the `Age` field.

> *As well as being used on class members (methods, properties, fields, events, and so on), access modifiers can also be used on whole classes, enums, interfaces, and structs. In this chapter we will focus on access modifiers applied to class members, but it's good to know that we can use them on other elements, too.*

We have learned about the `public` and `private` access modifiers, which are probably the most important. There are several others. Here is the complete list, with a short description of each:

Access modifier	Meaning
`public`	The least restrictive access modifier. Outer classes have full access to a public member.
`protected`	A protected member is accessible from the code within its class, or to classes **derived** from it – we will learn about derived classes later in this chapter.
`internal`	An internal member is accessible from the current assembly (in Visual Studio .NET this normally equals the current project).
`protected internal`	A protected internal member is accessible from the current project, or from the classes derived from its containing class.
`private`	A private member is accessible only to the code in the current class.

Properties

Now that we know about access modifiers, it's time to make our `Person` class a bit smarter by using properties instead of fields. Because Visual Basic 6 also supported properties, this might not be something new for you. Before upgrading the `Person` class, let's quickly see what properties are and what they do.

Properties provide a way for an object to control and validate the values saved to and retrieved from its fields, while enabling the calling classes to use the same natural syntax as with fields.

Because the `Person` class's `Age` and `Name` fields are public, other classes can set them to any value. Take a look at the following code snip:

```
Mike.Age = -5;
Mike.Name = "Michael";
```

We really want some way to validate the values that are saved in our objects. In this case the `Person` class setting `Age` to a negative value or `Name` to a string of empty spaces might not really matter, but in real-world situations it's very frustrating when code breaks because of invalid input data.

What can we do to protect our object? With some programming languages, such as Java, we would use methods instead of properties. `MyExecutableClass` would have looked something like this:

```
        Mike.SetAge (-5);
        Mike.SetName ("  Michael  ");
```

Methods can successfully validate the input value and take some action depending on the results. However, as you can see, the syntax doesn't look very natural. Methods are meant to describe an object's functionality, and setting a value has more to do with the object's state than with its functionality.

Properties allow us to use the same syntax as with fields, while providing the means to write any code needed to validate the values. Properties can also be read-only or write-only, which is yet another level of flexibility.

Adding Properties to the Person Class

Let's update the `Person` class to use properties instead of fields:

```
public class Person
{
   private int age;
   private string name;

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

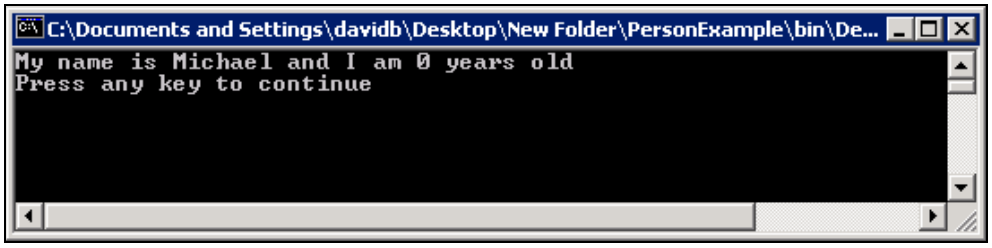
```
        {
            age = (value<0 ? 0 : value);
        }
    }

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            value = value.Trim();
            name = (value=="" ? "Unknown" : value);
        }
    }

    public void DescribeYourself()
    {
        Console.WriteLine
            ("My name is {0} and I am {1} years old", Name, Age);
    }
}
```

Updating the Executable

Now update `MyExecutableClass` to send some invalid data:

```
public class MyExecutableClass
{
  static void Main(string[] args)
  {
    Person mike;
    mike = new Person();
    mike.Age = -5;
    mike.Name = "     Michael    ";
    mike.DescribeYourself();
  }
}
```

Now, when you execute the program (*Ctrl-F5*), you should see the following output:



Let's take a look at how this works. We now use properties instead of fields for the public interface of the class. Note that properties do not hold the actual data. They act as a wrapper for the fields storing the data. The fields are usually private:

```
private int age;
private string name;
```

The official recommendation is to name public class members using Pascal casing (such as `MyProperty`), and private members with camel casing (such as `myProperty`). Remember that unlike Visual Basic 6, C# names are case sensitive – so `age` and `name` are different from `Age` and `Name`.

While the private fields store the object's data, we use the properties' `get` and `set` accessors as an extra layer between the external class that uses our object and the object's internal data.

Let's take another look at the `Age` property:

```
public int Age
{
   get
   {
      return age;
   }
   set
   {
      age = (value<0 ? 0 : value);
   }
}
```

The `set` accessor is used when a program wants to set a value on our object. So when the `Main` method makes this call:

```
         Mike.Age = -5;
```

it is the set accessor of the `Age` property that gets called:

```
            age = (value<0 ? 0 : value);
```

The ternary operator was presented in the last chapter, but here is the equivalent code for the line above:

```
            if (value<0)
               age = 0;
            else
               age = value;
```

`value` is a special keyword that represents the value sent to our property. `value`'s type will always be the same as the property's type. Our `Age` property takes care not to store any values less than zero to the `age` private field.

The `Name` property has a similar logic – it cuts the spaces from the incoming text, so in our example " `Michael` " will be transformed to "`Michael`". If the text is void (or a string of empty spaces), "Unknown" is stored instead.

Using properties is a very effective way to control what goes in or out of our object. It is often a good idea to use properties even for values that are directly stored to the private fields without any validation or calculation. Also, keep in mind that it is not always necessary to have a private value that matches exactly the value returned by the property. We could have, for example, a class named `Rectangle` with an `Area` property defined like this:

```
public int Area
{
   get
   {
      return Height*Width;
   }
   // set is missing - the property is read-only
}
```

In this example, `Area` is a read-only property because it doesn't have the `set` accessor – although `Height` and `Width` could both be read-write properties. Let's take a look at read- and write-only properties now.

Read-only and Write-only Properties

Another reason properties are so useful is that we can make them read-only or write-only. Don't confuse this with `readonly` fields, which we will discuss in the next chapter. A read-only property is a property without a `set` accessor, and a write-only property is a property without a `get` accessor.

Let's make the properties in our class write-only, by removing the `get` accessors:

```
public class Person
{
   private int age;
   private string name;

   public int Age
   {
      set
      {
         age = (value<0 ? 0 : value);
      }
   }

   public string Name
   {
      set
      {
         value = value.Trim();
         name = (value=="" ? "Unknown" : value);
      }
```

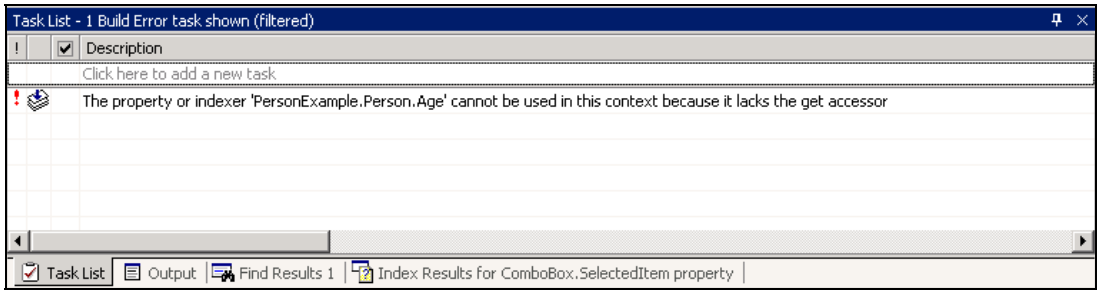
```
        }

        public void DescribeYourself()
        {
            Console.WriteLine
                ("My name is {0} and I am {1} years old", name, age);
        }
    }
```

In this version of the class, we removed the `get` accessor from both properties. We also modified the `DescribeYourself` method to use the private fields `name` and `age` instead of the `Name` and `Age` properties (we cannot read from them anymore). Now the external classes using our object will be able to set values on our properties, but will be unable to read the values.

When the program is executed it will work exactly as the previous time, because in `MyExecutableClass` we don't try to read the values from `Age` and `Name` – we use the `DescribeYourself` method instead. If you're curious to see what happens if we want to set the value of `Age`, add this line to the `Main` method of `MyExecutableClass`:

```
            Console.WriteLine ("Age: " + mike.Age);
```

When compiling the program, we get the expected error:



Principles of Object-Oriented Programming

We now know what objects and classes are, how to add data and behavior to a class, and how to use them in another program. It is now time to explore the fundamental principles of object-oriented programming: encapsulation, inheritance, and polymorphism.

These principles are valid for any object-oriented programming language – they are not specific to C# or the .NET Framework.

Encapsulation

Encapsulation is a concept of object-oriented programming that refers to the clear separation between an object's exposed functionality and its internal structure. In other words – the separation between *what* an object does, and *how* the object does it. Encapsulation is often referred to as the *black box* concept or *data hiding*.

The *what an object does* part is represented by the class's public interfaces. In this chapter we will only focus on the *default* public interface, which is formed by the declaration of the class's public members. The *how it works* part is made up by the class's private members, and method and property implementations.

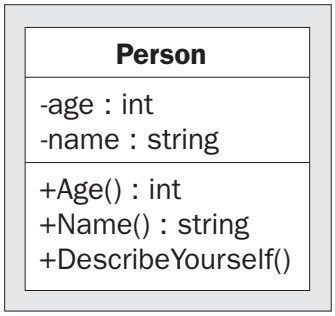
As with the other basic concepts of OOP, encapsulation represents a natural way we think about objects in real life. You know how to use your cell phone – its buttons and screen represent its "public interface" that you use to access functionality; but you probably don't really know how a cell phone works inside. It's the same with objects in programming. Other programmers using your object don't need to know how are the methods implemented, what data types are used internally or what other private methods are called within the method – they only have to know what its parameters are, its return value, and what it is supposed to do.

Even for people who design cell phones, it's still convenient not to think about the way it works inside every single time they make a phone call. The same is true of OO programming. Even if you are the only programmer on a project, encapsulation can make the problem far easier to handle.

Technically speaking, encapsulation is about deciding what access modifiers to use on your class's members. Which members should be public, and which should be private? Let's think about the implications, by looking at our `Person` class for a second:

```
public class Person
{
  private int age;
  private string name;

  public int Age
  {
    // get and set accessors here
  }

  public string Name
  {
    // get and set accessors here
  }

  public void DescribeYourself()
  {
    // method logic here
  }
}
```

We can represent this class using UML. The Enterprise Architect edition of Visual Studio .NET includes Visio, which can reverse engineer classes to produce UML diagrams:



Although UML is not the topic of this book, it is good to know how to read a diagram like this. Notice that right below the class name appear the class's fields (`age` and `name`). In the next box are listed the methods and properties (`Age`, `Name`, `DescribeYourself`). It should not be a great surprise that properties are presented as methods. Although properties are used like fields, technically they are more like methods since they can implement functionality.

Each member has a '+' or '-' on its left – the plus says that it's a public member, and minus is used to mark private members; protected members (not present in this example) are marked with '#'. In this class all fields are private and all methods and properties are public (the upper box has only minuses, and the lower box only pluses). This is not a rule. In fact, most real-world applications will have private methods, which would still be located in the bottom part of the diagram but with a minus on their left.

Let's go back to our theory now. This simple class helps us understand why it is so important to have a clear separation between the default interface (`Age`, `Name`, `DescribeYourself`) and the class's internal structure (`age`, `name`, and the method/property implementations). `Age`, `Name` and `DescribeYourself` can be used and called by outer classes such as client programs. `age` and `name` are internally used by `Person` to store its state. Because this example is very simple, we have only private fields, but we could also have private methods, properties, and so on.

Although we won't discuss its implications with great detail, it is good to know that there is a third, frequently used access modifier – `protected`. Protected access is in between public and private. Protected class members act as private to other classes that use `Person` objects, but are accessible from derived classes. We'll talk about derived classes in the next section.

It is easy to implement encapsulation once you know what you want (use `public`, `protected` or `private` as dictated by the design). But deciding what you want is tricky. Compared to traditional, procedural programming, design in object-oriented systems is much more important. This is mainly because object-oriented programming encourages software reuse. Classes, because they are self-contained units, can be easily reused – you can either reuse a class in many of your programs, or you can even distribute it for others to use. Consequently, improvements or optimizations to a class that is used in many other classes will automatically update the other classes. But if we remove or change the functionality of one of the public members, we can potentially break a number of programs that rely on the "old" functionality. The ease of reuse is a very powerful feature of OOP, and it can be your best friend or worst enemy – it all depends on how you design your classes.

Objects and encapsulation have changed the way software applications are designed. Instead of thinking about what functions and methods to add in a program, it's now very important to decide what classes to create, and what public functionality to include in them. After you have a clear understanding of what every one of your objects should do it's easy to decide what public members to add (including their signature consisting of the return type and input parameters). After that, it's easier to make the next step and create an implementation that provides functionality for every public signature.

Inheritance

Inheritance is a very important feature of OOP that enables a class to inherit all the properties and behavior of another class, while permitting the class to modify the inherited behavior or add its own. The class that inherits is called a *derived class*, while the class being inherited from is said to be the *base class*.

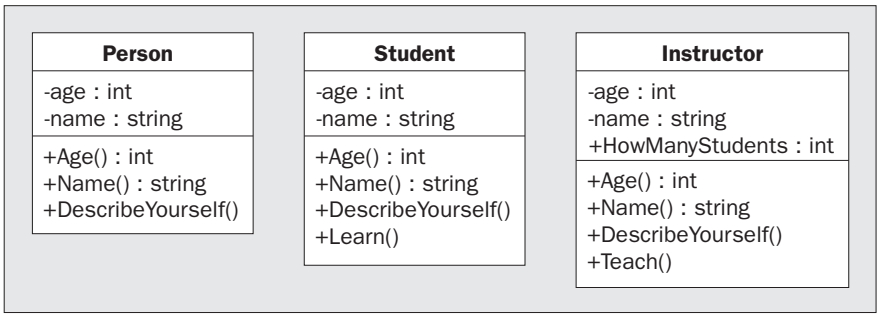
With C# and .NET, every class you write directly derives from exactly one other class – C# only supports *single inheritance*. You are not allowed to derive a class from more than one other class (as it is possible to do with C++), and if you don't supply a base class then `System.Object` will automatically be considered as the base class. Don't worry too much about these details right now – we will cover them in more depth in the next chapter.

Visual Basic 6 supported inheritance via the `Implements` keyword. That kind of inheritance is referred to as **interface inheritance**. It is also supported by .NET and C#, and it is covered in the next chapter. The kind of inheritance we are discussing right now is **implementation inheritance**. Unless specified otherwise, in this book (and in most material about OOP) the term 'inheritance' refers to implementation inheritance.

When you have a large number of classes, it is very likely that some of them are related to others in some way. How you design and implement these relations in code is very important, as it affects all further development using your classes.

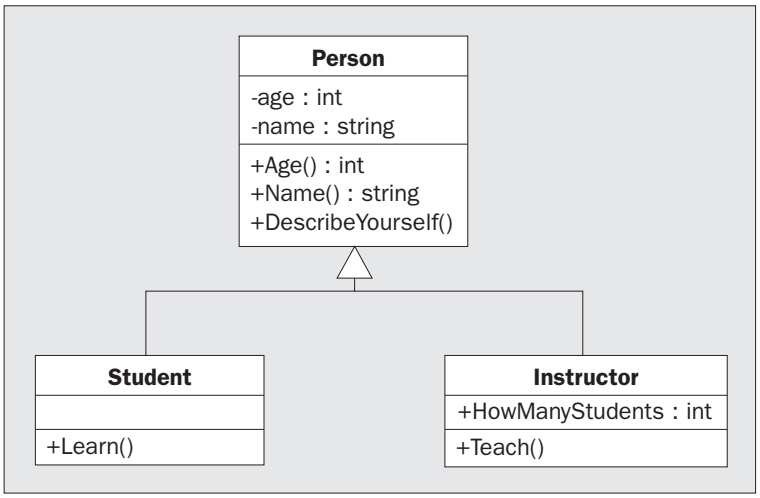
In order to demonstrate inheritance, let us imagine that you have to implement two classes named `Student` and `Instructor`. The imaginary specification requires that both `Student` and `Instructor` have `Age`, `Name`, and the possibility to describe themselves using the `DescribeYourself` method – just like our `Person` does. Apart from this common functionality, we want a `Student` to have specific behavior through a `Learn` method, and we also want to make `Instructor` able to `Teach`, and to have a public field named `HowManyStudents`.

You can simply build these classes as independent, standalone classes. In many cases, depending on the situation, this will be the right decision to make – to write your classes from scratch. Take a look at the UML diagram of the three "unrelated" classes:



Although this diagram can be relatively easy to implement with code, you can see that there would be a lot of redundant code. Even if `Age`, `Name`, and `DescribeYourself` have identical implementations, without inheritance we still need to write code for each of them three times. Additionally, if we find a better implementation for one of the members, we have to change it in all three places.

Inheritance can help us in this situation, as it allows us to reuse all `Persons`'s functionality in `Student` and `Instructor`. Both in programming and in the real world, an object that inherits from another will have the properties and behavior of the base object, while being able to have some properties of its own or change some of the behavior inherited from the initial object. If we create `Student` and `Instructor` by deriving them from `Person`, the UML diagram of the new scenario looks like this:



As you can see from the diagram, the derived classes only implement the additional functionality, which was not part of the base class (`Person`). The already existing functionality is inherited. `Student` and `Instructor` can also modify part of the inherited functionality, and we will cover this in the next chapter when presenting method and property overriding and hiding.

When to Use Inheritance

As a simple rule to keep in mind, inheritance can be applied when a class *is a* kind of base class. For example, it sounds fair to say that a `Student` *is a* `Person`, and that an `Instructor` *is a* `Person`. If two classes pass the *is a* test, generally this means that you can use inheritance with them.

However, just because you can use inheritance, this doesn't mean that you necessarily should. Also, if the classes don't pass the *is a* test, then most probably you shouldn't use inheritance, even if at the first sight it seems to be a good idea. Keep this in mind, as it is an important rule that is sometimes forgotten even by experienced programmers.

Inheritance is often highly overemphasized and many programmers who are new to OOP think that it should be used everywhere. This results in awkward and overcomplicated design. Inheritance allows for software reuse by adding very tight coupling between classes. If not properly designed, this might result in an inflexible and inextensible system.

Inheritance vs. Composition

Note that there is no single way to successfully implement a hierarchy of classes. While we will not cover advanced details about object-oriented design in this book, there are a few general principles that are well worth remembering.

Although inheritance is indeed very powerful and useful, keep in mind that there are alternatives. The most important of these is composition. Inheritance and composition are both object-oriented techniques that allow for code reuse. Now that we know what they have in common, let's quickly analyze the differences:

Inheritance is used to create a *specialized* version of a class that already exists. `Student` and `Instructor` are special kinds of `Person`. With inheritance, all public members of the base class automatically become part of the derived class's default interface. Inheritance offers the highest degree of reusability (all the base class's members are reused), but also implicates very tight coupling between classes. You must take very good care when designing the classes, because after a class is inherited from, it is very hard to modify it. Inheritance is verified with the *is a* rule.

Composition refers to using one or more objects as fields of another class. The new class is said to be composed of other objects. A typical example is the `Car` class, which among its (public or private) fields would have objects of type `Engine` and `Door`. Composition is verified using the *has a* rule. In our `Person` scenario we do not use composition. We could have used it if, for example, we wanted an instructor to keep a record of all their students. In that situation, we could have a private class field as collection of students.

Implementing Inheritance

OK, enough with the theory – let's write some code! The UML we are trying to implement can be seen in the diagram on the facing page.

We need to add two new classes to our project. Use File | Add New Item to add a new Class called `Student`. Then do the same to create a class called `Instructor`. This will add the files `Instructor.cs` and `Student.cs`.

`Person` didn't change from the last example – here is a reminder of how it should look:

```
public class Person
{
  private int age;
  private string name;

  public int Age
  {
    get
    {
      return age;
    }
    set
    {
      age = (value<0 ? 0 : value);
```

```
    }
  }

  public string Name
  {
    get
    {
      return name;
    }
    set
    {
      value = value.Trim();
      name = (value=="" ? "Unknown" : value);
    }
  }

  public void DescribeYourself()
  {
    Console.WriteLine
      ("My name is {0} and I am {1} years old", Name, Age);
  }
}
```

Now edit the `Student` and `Instructor` classes so that the class body reads as follows (leave the `using` directive and `namespace` declarations in place):

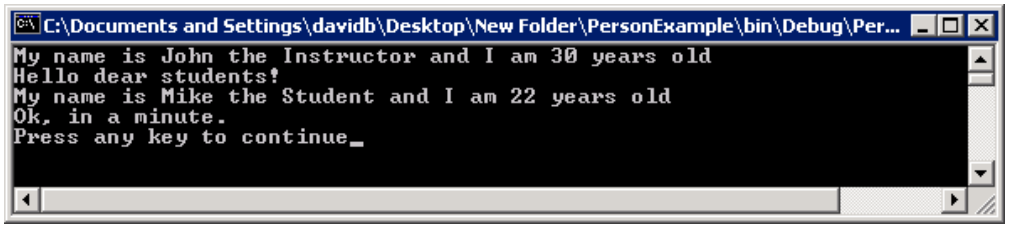
```
public class Student: Person
{
  public void Learn()
  {
    Console.WriteLine ("Ok, in a minute.");
  }
}
public class Instructor : Person
{
  public int HowManyStudents;

  public void Teach()
  {
    Console.WriteLine ("Hello dear students!");
  }
}
```

In order to test our fresh new classes, change `MyExecutableClass.cs` to read like this:

```
using System;

namespace PersonExample
{
  class MyExecutableClass
  {
    static void Main(string[] args)
    {
```

```
        Instructor john = new Instructor();
        john.Age = 30;
        john.Name = "John the Instructor";

        Student mike = new Student();
        mike.Age = 22;
        mike.Name = "Mike the Student";

        john.DescribeYourself();
        john.Teach();

        mike.DescribeYourself();
        mike.Learn();
      }
    }
  }
```

Press *Ctrl-F5* and you should see the following output:



We can see that both the `Instructor` and the `Student` can describe themselves in exactly the same way that a `Person` can. However, an `Instructor` can teach – but a `Student` can't. `Student` can learn – but an `Instructor` can't. Let's have a look at how this all works.

The first important thing to notice is the class definitions. As you can see, the C# syntax used to specify that a class derives from another is pretty simple. Here it is in the `Student` class:

```
class Student: Person
{
  public void Learn()
  {
    Console.WriteLine ("Ok, in a minute.");
  }
}
```

The first line specifies that `Student` inherits from `Person`. We then implement the `Learn` method, which is specific to `Student`. When we create a `Student` object, we can access all the public members of `Person`, plus the new `Learn` method:

```
Student Mike = new Student();
Mike.Age = 22;
Mike.Name = "Mike the Student";
Mike.DescribeYourself();
Mike.Learn();
```

Yes, that's pretty impressive. As in real life, a student is a person too – and has all of the behavior and characteristics that any other person has.

When referencing an object's member, the member is first looked for in the object's class. If it is not found there, then the member is searched for up in the class hierarchy – first in the base class, then in the base class's parent, and so on. If the member is not found anywhere, the compiler issues an error message.

If we had a big class hierarchy starting from `Person`, doing an optimization in the `Person` base class would have automatically upgraded all inherited classes. Also, you can see that we have created more advanced classes – `Student` and `Instructor` – with only a few lines of code. Inheritance can significantly reduce the number of lines of code we write.

This example shows how powerful inheritance can be in a real, commercial application. While OOP offers us many great advantages over traditional, procedural programming, it also can become our worst enemy if we don't take enough care when designing the application. Inheritance allows us to reuse some basic functionality in a whole hierarchy of classes, but if we don't design the base carefully enough, the whole structure won't last for too long. Compared to non-OOP architectures, we need to take much more care when designing an OOP class hierarchy. A well-designed base class will help enormously when further adding or updating classes in the project.

Polymorphism

Polymorphism is the ability to make use of objects of distinct classes by only knowing a base class from which they derive. Technically, polymorphism is about using a base class reference to access objects of derived classes.

If we have an object that is of type `Person` or a type derived from `Person`, then we know for sure that it exposes all `Person`'s public signatures. In other words, we know that we will always be able to call `DescribeYourself` on a `Person` object, on a `Instructor` object, or on any other object that has `Person` as a parent in its class hierarchy.

Let's take a look at an example demonstrating polymorphism. Using the code from the previous section, modify the `MyExecutableClass` like this:

```
class MyExecutableClass
{
  static void Main(string[] args)
  {
    Student mike = new Student();
    mike.Age = 22;
    mike.Name = "Mike the Student";

    Instructor john = new Instructor();
    john.Age = 30;
    john.Name = "John the Instructor";

    DescribePerson (mike);
    DescribePerson (john);
  }
```

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

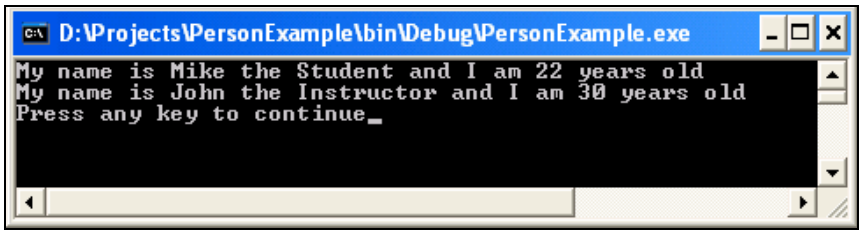
What we're doing here is very simple. We create two objects, one of type `Student` and one of type `Instructor`, and we send these objects as parameters to a method named `DescribePerson`:

```
  Student mike = new Student();
  //...

  Instructor john = new Instructor();
  //...

  DescribePerson (mike);
  DescribePerson (john);
```

The very interesting idea that you must be aware of is that `DescribePerson` is capable of dealing with both types of objects – `Student` and `Instructor`. But how is that possible? To reveal the answer, let's analyze the `DescribePerson` method:

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

The solution to our question is revealed by `DescribePerson` method's definition: `DescribePerson` is defined to accept a parameter of type `Person`, and give it the name `person`. According to the definition of polymorphism, we can use a base class reference to reference an object of a derived class. That's exactly what we are doing here. Although the `person` parameter is of type `Person`, we can also send objects of any type derived from `Person`. This is the output of the program:



Polymorphism is a very powerful feature. We can define a generic method, like `DescribePerson` in our example, without knowing exactly what kind of `Person` we will receive – we only need to know that what we receive *is a* `Person`, and that guarantees us that the received object supports all functionality exposed by `Person`.

Keep in mind that, with all the flexibility provided by polymorphism, the `person` parameter is still a `Person` object. This means that we cannot have a `DescribePerson` like this one:

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

This would not compile, because `Learn` is defined in `Student` and not in `Person`, and `Person` does not know anything about `Learn`.

There are more interesting areas we will cover about polymorphism, but for now it is important for you to understand the main concept. We'll deal with some subtle tricks in the following chapter.

Method Overloading

Method overloading is a very useful feature of modern programming languages, which enables us to have several methods with the same name, but with different types of parameters. Visual Basic 6 did not support method overloading.

Let's start with a simple example. For the examples to come, we'll use public fields `Age` and `Name` instead of properties, so that the code is shorter and easier to follow. In real life, though, it's generally better to use properties.

Modify the `Person` class to read like this:

```
public class Person
{
  public int Age;
  public string Name;

  public void DescribeYourself()
  {
    Console.WriteLine("Hello dude, my name is {0}.", Name);
  }

  public void DescribeYourself(string somebody)
  {
    Console.WriteLine("Hello {0}, my name is {1}.", somebody, Name);
  }
}
```

Use this implementation of `MyExecutableClass` to test the new `Person`:

```
class MyExecutableClass
{
  static void Main(string[] args)
  {
    Person johnny = new Person ();
    johnny.Name = "Johnny Bravo";
```

```
        johnny.DescribeYourself("Simon");
        johnny.DescribeYourself();
      }
    }
```

The output of the program is:

```
Hello Simon, my name is Johnny Bravo.
Hello dude, my name is Johnny Bravo.
Press any key to continue
```

Our code worked as expected. The compiler was able to determine which version of `DescribeYourself` to call each time we called it.

The basic requirement for having overloaded methods is that they should have different parameters. In our example, one method has no parameters, and the other has one input parameter. This makes our methods totally different from the compiler's point of view – the compiler will always know which version of `DescribeYourself` we are calling, by examining the number of parameters we send to it.

C# also allows us to have overloaded methods with the same number of parameters. If this is the case, the parameter types must be different. We can also have different return types for our overloaded methods, but simply having a different return type is not sufficient (we still need to have different parameter types or different numbers of parameters).

Let's quickly create another very simple example. Add these two methods to `Person`:

```
public void SomeMethod(int param)
{
  Console.WriteLine("INT");
}

public void SomeMethod(long param)
{
  Console.WriteLine("LONG");
}
```

Which method will be called if we send number 5 to `SomeMethod`? Let's test this by making a call in the `Main` method of `MyExecutableClass`:

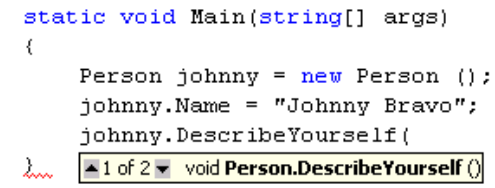
```
        johnny.SomeMethod(5);
```

We'll see INT as the output, demonstrating that `int` is the default data type for integer values. If we wanted to use the overload that takes a `long` parameter, we needed to make an explicit cast:

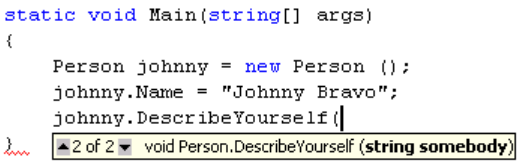
```
        johnny.SomeMethod((long)5);
```

Now the output shows LONG.

There is no theoretical limit on the number of overloads a method may have, but most times you won't need more than two or three. Of course this is not a general rule, and the .NET Framework contains classes with many overloads for the same method name.
The last word about method overloading is about the support for overloaded methods provided by the IntelliSense technology. Here's a screenshot from Visual Studio .NET:



The screenshot shows the first of the two overloads. If you press the 'down' arrow key, IntelliSense will move on to the next overload:



Simulating Optional Parameters in C#

One of the useful features of Visual Basic 6 is that it allows us to have optional method parameters. C# does not support this feature, but we can use method overloading to provide the same behavior.

Here is a simple implementation of the `Person` class in Visual Basic 6:

```
'Person class
Public Name As String

Public Sub DescribeYourself(Optional somebody As String = "dude")
    MsgBox "Hello " & somebody & ", my name is " & Name
End Sub
```

Here the `DescribeYourself` method accepts the optional `CallerName` parameter. If the caller program supplies a parameter, then the supplied parameter is used.

We could test our `Person` with this code:

```
Private Sub Form_Load()
    Dim mike As New Person
    mike.Name = "Michael"
    mike.DescribeYourself
End Sub
```

The output will be a text box containing the text "Hello , my name is Michael". If we change the call like this:

```
mike.DescribeYourself "Christian"
```

The output is "Hello Christian, my name is Michael".

Let's switch back to C#. As I said before, there are no optional parameters in C#. Instead, we use overloaded methods. Here is the implementation of `DescribeYourself`, the C# way:

```
public void DescribeYourself()
{
  DescribeYourself("dude");
}

public void DescribeYourself(string somebody)
{
  Console.WriteLine ("Hello {0}, my name is {1}", somebody, Name);
}
```

If called without parameters, the parameterless method will call the overload with a parameter, supplying the default value. If we directly call the overload that accepts a parameter, the supplied string is used instead. If you execute the program again, you'll see that the results are identical. The difference now is that all the logic is in the parameterized method. This way of coding may be helpful if you want to keep all the logic in one place.

Constructors

Constructors are a special kind of methods used to initialize the object. The constructor is always the first method that gets called when the object is created using the `new` keyword. Constructors can initialize the object before other methods are called on it. Probably most classes you will write will have at least one constructor defined (yes, we can have more constructors for a class, as we'll see shortly), since most classes need to have some data initialized at creation time.

The only two ways to call a constructor are to create the object (so the constructor is called automatically), or to call the constructor from another constructor (we will see how to do this very soon). Note that in this section we are talking about *instance constructors* – there is another type of constructors named static constructors, which will be presented in the next chapter.

Visual Basic 6 did not support constructors in their strictest meaning, but there was a function named `Class_Initialize` that served a similar purpose. As you will see, constructors are much more powerful than Visual Basic's `Class_Initialize` function.

In C#, the constructor is defined as a method that has the same name as the class, and has no return type (not even `void`). If you don't supply a constructor for your class, the C# compiler will automatically generate a default constructor.

In the following examples, we'll experiment by adding various types of constructors to the `Person` class. Add the following constructor to `Person`:

```
class Person
{
  public int Age;
  public string Name;

  public Person()
  {
    Console.WriteLine("Constructor called.");
    Name = "Somebody";
    Age = 21;
  }

  public void DescribeYourself()
  {
    DescribeYourself("dude");
  }

  public void DescribeYourself(string somebody)
  {
    Console.WriteLine ("Hello {0}, my name is {1}", somebody, Name);
  }
}
```

Modify the `Main` method of `MyExecutableClass` like this:

```
class MyExecutableClass
{
  static void Main(string[] args)
  {
    Person johnny = new Person ();
    johnny.Name = "Johnny Bravo";

    Person somebody = new Person ();

    johnny.DescribeYourself("Simon");
    somebody.DescribeYourself();
  }
}
```

The output is the following:

```
Constructor called.
Constructor called.
Hello Simon, my name is Johnny Bravo
Hello dude, my name is Somebody
Press any key to continue
```

Of course, for this particular example the constructor isn't very useful. Moreover, we could achieve (almost) the same functionality by using default values for instance fields in the `Person` class like this:

```
public int Age = 21;
public string Name = "Somebody";
```

When Exactly Is the Constructor Called?

We learned earlier that an object is created in two steps: first we declare the type of the object, and then we allocate memory for it.

In order to make it very clear when the constructor is called, use the `Person` class we have just updated by adding a constructor, and modify `MyExecutableClass` like this:

```
class MyExecutableClass
{
  static void Main(string[] args)
  {
    Console.WriteLine("Declaring Person ...");
    Person john;

    Console.WriteLine("Creating a new Person in memory ...");
    john = new Person();

    john.DescribeYourself();
  }
}
```

The output answers our question:

```
Declaring Person ...
Creating a new Person in memory ...
Constructor called.
Hello dude, my name is Somebody
Press any key to continue
```

The constructor is called to create a new instance of a `Person`. Actually, this is normal, and it is the only possible sequence: we cannot execute a method on an object that hasn't been created yet. The output of our last program makes sense.

Parameterized Constructors

Parameterized constructors are like normal constructors, in that they are called when we create the object, but they also receive parameters. As we'll see, they are very useful in a programmer's everyday life.

Note that if we supply a constructor with parameters, the default (parameterless) constructor is not automatically generated by the C# compiler any more. This means that if we supply only constructors that take parameters, we must supply those parameters when creating the object.

For now, let's update `Person`'s constructor like this:

```
class Person
{
  public int age;
  public string name;

  public Person(int age, string name)
  {
    this.age = age;
    this.name = name;
    Console.WriteLine("I'm {0}. I feel like a new object!", name);
  }

  public void DescribeYourself()
  {
    DescribeYourself("dude");
  }

  public void DescribeYourself(string somebody)
  {
    Console.WriteLine ("Hello {0}, my name is {1}", somebody, name);
  }
}
```

We chose to set a `Person`'s initial values in the constructor, so we made the instance fields private – public `Age` and `Name` became private `age` and `name`. Be sure to also update `DescribeYourself` to use `name` instead of `Name`.

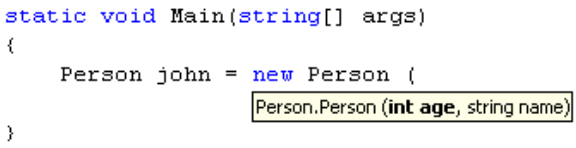
What our new constructor does is simple – it takes two parameters, and saves their values to the private instance fields. It also writes a 'this is a constructor' type of message.

If you still have the `Student` and `Instructor` classes in your working files, comment them out for now, because they won't compile after adding parameters to the `Person`'s constructor. The reason is that the default constructor of the derived classes automatically calls the base class's parameterless (or default) constructor. Because we don't have a parameterless constructor in the base class right now, and we also don't have a default constructor, `Student` and `Instructor` cannot be created. There are basically two solutions to this problem: one is to add a parameterless constructor to the base class; the other is to add a constructor in the derived class that doesn't call the base class's parameterless constructor. We'll see how to implement both solutions in the following sections.

Update `MyExecutableClass` like this:

```
class MyExecutableClass
{
   static void Main(string[] args)
   {
      Person johnny = new Person (25, "Johnny Bravo");
      johnny.DescribeYourself();
   }
}
```

Observe the syntax used to create an object with a parameterized constructor. We must pass the two parameters when creating the object. Also note that Visual Studio's IntelliSense technology works great with constructor's parameters, too:



The program's output is as expected:

```
I'm Johnny Bravo. I feel like a new object!
Hello dude, my name is Johnny Bravo
Press any key to continue
```

After working with this last example, we can see two reasons why parameterized constructors are very useful:

- They requires less lines of code to initialize an object – `MyExecutableClass` is now much shorter than its previous versions.
- They limit the ways an object can be initialized. With the last `Person` class, we can't create a `Person` object if we don't supply a name and an age for it, which makes more sense than having default values for these fields.

If you're curious what happens if we try to create a `Person` without supplying the constructor's parameters, add a line like this in the `Main` method:

```
      Person john = new Person ();
```

The compiler will generate the following error: No overload for method 'Person' takes '0' arguments. This demonstrates that, indeed, no default constructor is generated for the `Person` class after we add a constructor to our class.

Overloaded Constructors

As it is possible to have overloaded methods, C# also allows us to have overloaded constructors. Again, this very useful feature was missing, unfortunately, in Visual Basic 6. The rules are the same as with method overloading – all constructors must have different parameters.

Let's modify the `Person` class like this:

```
class Person
{
  public int age;
  public string name;

  public Person()
  {
    this.age = 99;
    this.name = "Somebody";
    Console.WriteLine("I'm {0}. I don't like my default name.", name);
  }

  public Person(int age, string name)
  {
    this.age = age;
    this.name = name;
    Console.WriteLine("I'm {0}. I'm a great person!", name);
  }

  public void DescribeYourself()
  {
    DescribeYourself("dude");
  }

  public void DescribeYourself(string somebody)
  {
    Console.WriteLine ("Hello {0}, my name is {1}", somebody, name);
  }
}
```

We have two constructors now. The parameterless constructor is not very useful, but we're using it just to demonstrate the theory. Having these constructors allows us to create objects like this:

```
class MyExecutableClass
{
  static void Main(string[] args)
  {
    Person somebody = new Person ();
    Person mike = new Person (25, "Michael");
  }
}
```

The result is as expected:

```
I'm Somebody. I don't like my default name.
I'm Michael. I'm a great person!
Press any key to continue
```

Calling a Constructor from Another Constructor

We know that we cannot explicitly call a constructor from another method of the class. But what about calling a constructor from another constructor? Fortunately, C# provides us with the means to do this. Calling constructors allows achieving the "default parameter" feature that we have already demonstrated for methods. Unlike methods, constructors can call one another only through a special syntax that offers less flexibility.

Modify the `Person` class's parameterless constructors like this:

```
public Person(): this(99, "Somebody")
{
  Console.WriteLine
    ("I was initialized in the other constructor.");
}

public Person(int age, string name)
{
  this.age = age;
  this.name = name;
  Console.WriteLine("I'm {0}. I'm a great person!", name);
}
```

Execute the program, using the version of `MyExecuteClass` from the previous section. The output is:

```
I'm Somebody. I'm a great person!
I was initialized in the other constructor.
I'm Michael. I'm a great person!
Press any key to continue
```

What happens when using the parameterless constructor is that it automatically calls the parameterized constructor by providing the default values – and this is the first thing the constructor does:

```
public Person(): this(99, "Somebody")
```

After the referenced constructor is executed, the rest of the code in the calling constructor is called too:

```
{
  Console.WriteLine
    ("I was initialized in the other constructor.");
}
```

The way one constructor calls another constructor is less flexible than with methods because we cannot control the exact time the other constructor gets called. In the case of constructors, the referenced constructor is always called before any other code in the calling constructor is executed. The output of this program demonstrates the flow of code execution.

Sequence of Executing Constructors with Derived Classes

When we create an object with the type of a derived class, the derived class's constructor first calls the base class's constructor before executing itself. Why? Well, if you think about it, it makes sense – the derived class is very likely to "use" functionality of the base class, even in the constructor, so the base class must be initialized before the derived class.

In order to test this, we need a class that derives from `Person`. Modify `Student` by adding a parameterless constructor:

```
class Student: Person
{
  public Student()
  {
    Console.WriteLine ("I'm a student? No way!");
  }
  public void Learn()
  {
    Console.WriteLine ("Ok, in a minute.");
  }
}
```

The `Person` class is the one used in the previous example. It has the following constructors:

```
public Person(): this(99, "Somebody")
{
  System.Console.WriteLine
    ("Parameterless constructor called");
}

public Person(int age, string name)
{
  this.age = age;
  this.name = name;
  Console.WriteLine("My name is {0}. I'm a great person!", name);
}
```

Cool, we have three constructors right now! Let's initialize a new `Student`, and see how all three get called:

```
class MyExecutableClass
{
   static void Main(string[] args)
   {
      Student student = new Student();
   }
}
```

Execute the program – you should see the following output:

```
My name is Somebody. I'm a great person!
Parameterless constructor called
I'm a student? No way!
Press any key to continue
```

After analyzing the code and the results, it should be very clear what happens: when we create a new student, the `Person` constructors are called before the `Student` constructor. Since we did not explicitly mention which of `Person`'s constructors should be called, the parameterless constructor is called by default. The parameterless constructor, in its turn, calls the parameterized constructor before it itself is executed. This explains why the `Person` parameterized constructor is the first that is actually executed, followed by `Person`'s parameterless constructor, followed by `Student` constructor.

Choosing which Constructor of the Base Class to Call

In the last experiment of this chapter, we play again with `Student` and `Person`. In the previous test, we saw that `Student`'s constructor automatically calls by default the `Person` class's *parameterless* constructor. There are many times when we want to override this behavior – one of the scenarios being when `Person` actually does not have a parameterless constructor. In the code of the previous example, if you remove `Person`'s parameterless constructor, you will receive a compile-time error: No overload for method 'Person' takes '0' arguments.

For this example, do just that – remove the parameterless constructor from the `Person` class. We should only have the parameterized constructor:

```
public Person(int age, string name)
{
  this.age = age;
  this.name = name;
  Console.WriteLine("Me - {0}. Am I cool or what?!", name);
}
```

Modify `Student`'s constructor like this:

```
public Student(int age, string name): base(age, name)
{
  Console.WriteLine ("Show me to the classroom");
}
```

Now that we don't have a parameterless constructor in `Student`, we have to supply the required values in `MyExecutableClass`:

```
class MyExecutableClass
{
  static void Main(string[] args)
  {
    Student johnny = new Student(25, "Johnny Bravo");
  }
}
```

Execute the program. You should receive the following output:

```
Me - Johnny Bravo. Am I cool or what?!
Show me to the classroom
Press any key to continue
```

Let's quickly see why this works. Remember from the previous examples that when we create a `Student` object, the constructor of `Person` is also called. If not otherwise specified, the parameterless constructor is called. In this example, after we removed the `Person` parameterless constructor, we had to instruct the `Student` constructor what constructor of the base class to call.

As you have probably seen, the syntax to do this is not very complicated:

```
public Student(int age, string name): base(age, name)
```

`Student`'s constructor calls the base class's constructor that takes two parameters. As expected, `Person`'s constructor is the first one that is called – this is demonstrated by the program's output.

Summary

Here we are. It's been quite a long, complicated chapter – but hopefully an interesting one. We've covered a lot in this chapter – the world of objects can be so amazing! The subjects we've looked at include:

- Objects and classes. We saw that classes define an object, and objects are instances of a class. We saw how we can use software objects to model real-world objects in our code.
- How to add fields, methods, and properties to our classes – and how to use them from instantiated objects. We saw that these items are called members, and that we can control access to them using access modifiers such as `private` and `public`.
- We saw that classes can inherit from other classes, and that this is appropriate where one class *is a* specialized type of another class. We also saw that we can use composition for *has a* relationships between objects.
- We looked briefly at polymorphism – how one method can process objects from several classes, provided they derive from a known base class.
- We saw how to overload methods, and how this can enable us to simulate optional parameters.
- Constructors and how multiple constructors within a class hierarchy can work together.

These techniques are central to C# programming, and it is important to have a good understanding of them. In the next chapter we will be looking at some more advanced techniques, and these will seem confusing without a thorough understanding of the material here.

In this chapter we learned how to add functionality in derived classes, but we haven't seen yet how to modify the inherited behavior. We learned that objects are a part of any OO programming language, but we didn't see what special features .NET and C# have for us. These will be just few of the topics we will cover in the following chapter. Hang on – the journey has only just begun.