1. OOP Overview

The second part of the C# book is dedicated to the object-oriented features that were added to the original C language by Bjarne Stroustrup. To understand this part and the following chapters, you must have a solid understanding of all the chapters discussed in Part 1 of this book. You have to gain mastery of all the fundamental concepts and control structures of the C language, like conditions, loops, functions, arrays, strings, structures, and pointers, and all the details explained in Part 1.

So, what is OOP? First of all, OOP is not a programming language; it is a style of programming that is applied to many programming languages. This means that C# is not the only object-oriented language; in fact, the first object-oriented language was a language called Simula. What Bjarne did was apply the OOP features to the original C language and call the new version C with classes; then the name was changed to C#.

In the old style of programming that we learned in part one of this book, called procedural programming or structured programming, the program consisted of several functions (procedures) that performed a specific task and the main function that controlled the program execution. In the OOP style of programming, we have classes and objects.

A class is a general term like the term bird, and a specific dove is an object created from the bird class; a sparrow is another object that we can create from the bird class; we cannot create a cat object from the bird class because it simply doesn't make sense; we should make a class for cats to create cat objects; so, objects belonging to a class share common attributes, also called data members, and functionalities, also called member functions or methods.

We create objects from classes; objects have attributes (data) and methods:

Object = Attributes + Methods

Let me give you another example: the football player can be considered a class, and the individual players are objects created from that class. Football players all share common attributes; all of them have speed, length, skills, position, shot strength, etc., but each player has his own values, and all of them have functionalities; they all run, pass, shoot, some of them attack and some of them defend, etc. Take a look at the following figure:

Anything in the world can be an object. An object is a specific item that belongs to a class; it is also called an instance of a class.

So, why do we need to program in the OOP style? The invention of this style of programming came to solve issues and problems that appeared in the old style of programming, especially with large programs. Programmers needed a way to protect their data from the rest of the program; they needed a cleaner way to organize the code in classes that hold objects with common attributes and functionalities. This isolation of data, also known as data hiding, also known as encapsulation, provided a layer of protection to the data. A class is like a capsule that contains data and methods.

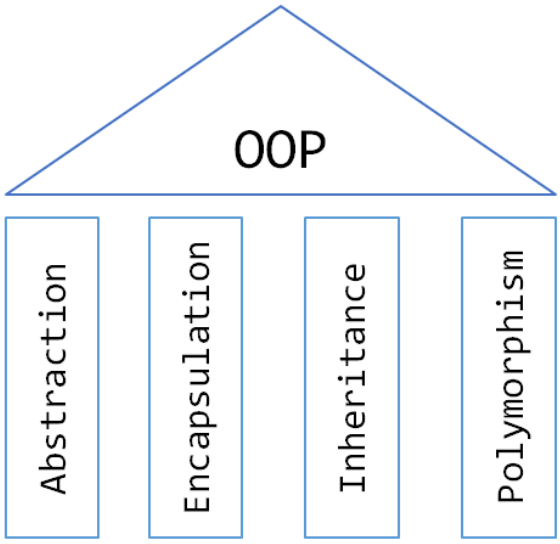
Classes also provide data abstraction, which is the process of paying attention to important properties while ignoring the details of implementation. When you, as a programmer, don't need to know the details of a specific class because you just want to benefit from its functionality, all the implementation details are hidden inside the class.

Also, this grouping of code made error handling easier since the code of each class was isolated from the other classes in the program.

Another powerful feature of OOP is inheritance, which simply means inheriting the code that was previously written rather than writing it all over again. This will save development time and make C# programmers more productive. Another feature is polymorphism, which means a specific function can take many forms. The modules will change the way they operate depending on the context, as we will see later. And many other exciting features are added to the programming languages to make programming more productive, safer, and easier to handle errors.

OOP supports all the procedural programming features and also provides:

1. Data abstraction
2. Encapsulation (data hiding)
3. Inheritance (code reusability)
4. Operators overloading
5. Polymorphism
6. Exceptions handling



Defining a Class

The class definition depends on our understanding of the problem we want to solve. We have to think about what attributes or data we need to put in the class and then what methods are needed to process that data. The syntax of defining a class in C#:

```
class ClassName
{
    // private members
    private int data1;
    private void Method1()
    {
    }

    // public members
    public int data2;
    public void Method2()
    {
    }

    // protected members
    protected int data3;
    protected void Method3()
    {
    }
}
```

The class definition will contain attributes or data and methods, with three access specifiers: private, public, and protected.

private: means the data are private and can be accessed only from inside the class and some friends (more about friends in the next chapter). Note that this is the default access specifier for classes; if you don't write an access specifier, it will be considered private to ensure the encapsulation of data.

public: means the data and methods are public and can be accessed from anywhere in the program. Usually, methods are specified as public so we can call them from the main function; this section is also called the class interface.

So, for example, if we want to **find the area and circumference of any rectangle** in the OOP style, first of all, we need to think about what we should name the class; in this case, it's obvious, so let us call it rectangle. Then we need to think about what attributes or data this class will hold. As long as our problem is to find the area and the circumference, we need the dimensions width and length, so we can write the definition of the class as follows:

The following program creates a member function called read() that will initialize the data members for each object.

Program 1

```
using System;

class Rectangle
{
    private int width, length;

    public void Read(int x, int y)
    {
        width = x;
        length = y;
    }

    public int Area() => width * length;
    public int Circumference() => (width + length) * 2;
}

class Program
{
    static void Main()
    {
        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle();

        r1.Read(5, 9);
        r2.Read(2, 6);

        Console.WriteLine("r1 area is: " + r1.Area());
        Console.WriteLine("r2 area is: " + r2.Area());
        Console.WriteLine("r1 circumference is: " + r1.Circumference());
        Console.WriteLine("r2 circumference is: " + r2.Circumference());
    }
}
```

Constructors

There is a better way to initialize the class data members by using a special method called a constructor. The class constructor performs these tasks:

- Initialize the class data members.
- Reserve memory space for the object depending on its contents.

A constructor has the following properties:

- It has the same name as the class.
- It has no return type (not even void).
- Called automatically when the object is declared.
- It can be overloaded (we can have more than one constructor).

Since the constructor is called automatically, we don't have to call it as we did with the read() method in Program 1. To add a constructor to our previous program, see the following Program 2.

Program 2

```
using System;

class Rectangle
{
    private int width, length;

    public Rectangle(int x, int y)
    {
        width = x;
        length = y;
    }

    public int Area() => width * length;
    public int Circumference() => (width + length) * 2;
}

class Program
{
    static void Main()
    {
        Rectangle r1 = new Rectangle(5, 9);
        Rectangle r2 = new Rectangle(2, 6);
```

```
        Console.WriteLine("r1 area is: " + r1.Area());
        Console.WriteLine("r2 area is: " + r2.Area());
        Console.WriteLine("r1 circumference is: " + r1.Circumference());
        Console.WriteLine("r2 circumference is: " + r2.Circumference());
    }
}
```

The read() method was replaced by the class constructor rectangle(). Note that the constructor has the same name as the class, has no return type, and initializes the class data.

The main function has the declaration of two objects; note that we send the initial values for each object in the object declaration statement. Since the constructors will be called automatically, we don't have to call them in code as we did in the read() method in Program 1.

We can also write the constructor in the member initialization list syntax like this:

rectangle(int x, int y) : width(x), length(y) {}

Constructor Overloading

The overloading concept in programming means assigning more than one job to a function. Function overloading means multiple implementations for the same function; the correct function is called depending on the number or type of arguments it receives. A class constructor is also a function, and it can be overloaded. In the following program, another constructor is added.

Program 3

```
using System;

class Rectangle
{
    private int width, length;

    public Rectangle()
    {
        Console.Write("Enter Dimensions: ");
        width = int.Parse(Console.ReadLine()!);
        length = int.Parse(Console.ReadLine()!);
    }

    public Rectangle(int x, int y)
    {
```

```
        width = x;
        length = y;
    }

    public int Area() => width * length;
    public int Circumference() => (width + length) * 2;
}

class Program
{
    static void Main()
    {
        Rectangle r1 = new Rectangle();      // constructor 1
        Rectangle r2 = new Rectangle(2, 6);  // constructor 2

        Console.WriteLine("r1 area is: " + r1.Area());
        Console.WriteLine("r2 area is: " + r2.Area());
        Console.WriteLine("r1 circumference is: " + r1.Circumference());
        Console.WriteLine("r2 circumference is: " + r2.Circumference());
    }
}
```

Note that the class in Program 3 has two constructors: the first one receives no parameters because it reads the rectangle dimensions, and the other constructor receives two values and assigns them to the class data.

The first constructor will be called automatically when object r1 is declared, and the second constructor will be called automatically when object r2 is declared.

Destructors

The other special function that can be added to the class definition is the destructor. The destructor job is the opposite of the constructor; it destructs the object to free the memory occupied by it.

A destructor has the following properties:

- It has the same name as the class.
- The name begins with a tilde ~
- It has no return type.
- It has no arguments.
- Called automatically at the end of the object's scope.
- Cannot be overloaded.

Removing the object from memory is done by deleting the object's data, but we cannot delete variables since they represent fixed memory locations, so we have to work with pointers. We can delete a pointer by making it point to something else, so the location it was pointing to will be freed and returned to the operating system.

Program 4

```
using System;
class Rectangle
{
   private int width;
   private int length;
   // Constructor
   public Rectangle(int x, int y)
   {
      width = x;
      length = y;
      Console.WriteLine("Constructor is called");
   }
   // Destructor
   ~Rectangle()
   {
       Console.WriteLine("Destructor is called");
   }

   public int Area()
   { return width * length;
   }
   public int Circumference()
   {return (width + length) * 2;
   }
}
class Program
{
   static void Main()
   {
      Rectangle r1 = new Rectangle(5, 9);
      Rectangle r2 = new Rectangle(2, 6);
      Console.WriteLine("r1 area is: " + r1.Area());
      Console.WriteLine("r2 area is: " + r2.Area());
      Console.WriteLine("r1 circumference is: " + r1.Circumference());
```

```
        Console.WriteLine("r2 circumference is: " + r2.Circumference());
        r1 = null;
        r2 = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```