Chapter 3: Object Oriented Programming in C#

What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Example 1. class is Fruit and object is Apple, Banana and Mango.

2. Class is Cars and object is Volvo, Audi and Toyoto.

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

Classes and Objects

Everything in C# is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the class keyword:

Create a class named "Car" with a variable color:

```
class Car
{
  string color = "red";
}
```

Create an Object

An object is created from a class. We have already created the class named Car, so now we can use this to create objects.

To create an object of Car, specify the class name, followed by the object name, and use the keyword new:

Example

Create an object called "myObj" and use it to print the value of color:

```
class Car
{
  string color = "red";

  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }
}
```

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of Car:

```
class Car
{
  string color = "red";
  static void Main(string[] args)
  {
    Car myObj1 = new Car();
    Car myObj2 = new Car();
    Console.WriteLine(myObj1.color);
    Console.WriteLine(myObj2.color);
  }
}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the fields and methods, while the other class holds the Main() method (code to be executed)).

- prog2.cs
- prog.cs

prog2.cs

```
class Car
{
  public string color = "red";
}
```

prog.cs

```
class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }
}
```

Class Members

Fields and methods inside classes are often referred to as "Class Members":

Example

Create a Car class with three class members: **two fields** and **one method**.

```
// The class
class MyClass
{
  // Class members
  string color = "red";        // field
  int maxSpeed = 200;          // field
  public void fullThrottle()   // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

Fields

In the previous chapter, you learned that variables inside a class are called fields, and that you can access them by creating an object of the class, and by using the dot syntax (.).

The following example will create an object of the Car class, with the name myObj. Then we print the value of the fields color and maxSpeed:

Example

```
class Car
{
  string color = "red";
  int maxSpeed = 200;

  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);  } }
```

Constructors

A constructor is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

Example

Create a constructor:

```
// Create a Car class
class Car
{
  public string model;  // Create a field

  // Create a class constructor for the Car class
  public Car()
  {
    model = "Mustang"; // Set the initial value for model
  }

  static void Main(string[] args)
  {
    Car Ford = new Car();  // Create an object of the Car Class (this will call the constructor)
    Console.WriteLine(Ford.model);  // Print the value of model
  }
}

// Outputs "Mustang"
```

Constructor Parameters

Constructors can also take parameters, which is used to initialize fields.

The following example adds a string modelName parameter to the constructor. Inside the constructor we set model to modelName (model=modelName). When we call the constructor, we pass a parameter to the constructor ("Mustang"), which will set the value of model to "Mustang":

Example

```
class Car
{
  public string model;

  // Create a class constructor with a parameter
  public Car(string modelName)
  {
    model = modelName;
  }

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang");
    Console.WriteLine(Ford.model);
  } }

// Outputs "Mustang"
```

Destructor

A destructor works opposite to constructor, It destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

Destructors in C# are methods inside the class used to destroy instances of that class when they are no longer needed. The Destructor is called implicitly by the .NET Framework's Garbage collector and

therefore programmer has no control as when to invoke the destructor. An instance variable or an object is eligible for destruction when it is no longer reachable.

Important Points:

- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- A Destructor has no return type and has exactly the same name as the class name (Including the same case).
- It is distinguished apart from a constructor because of the *Tilde symbol (~)* prefixed to its name.
- A Destructor does not accept any parameters and modifiers.
- It cannot be defined in Structures. It is only used with classes.
- It cannot be overloaded or inherited.
- It is called when the program exits.
- Internally, Destructor called the Finalize method on the base class of object.

Syntax:

```
class Example
{
   // Rest of the class
   // members and methods.

  // Destructor
  ~Example()
   {
      // Your code
   }
}
```

Consructor and Destructor Example

Let's see an example of constructor and destructor in C# which is called automatically.

```
using System;
  public class Employee
   {
      public Employee()
      {
         Console.WriteLine("Constructor Invoked");
      }
      ~Employee()
      {
         Console.WriteLine("Destructor Invoked");
      }
  }
  class TestEmployee{
     public static void Main(string[] args)
      {
         Employee e1 = new Employee();
         Employee e2 = new Employee();
      }
  }
```

OUT PUT:-

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

Note: C# destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.
Note: Destructor can't be public. We can't apply any modifier on destructors.

Inheritance :

Inheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- **Derived Class** (child) - the class that inherits from another class
- **Base Class** (parent) - the class being inherited from

To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

Example

```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()            // Vehicle method
  {
    Console.WriteLine("Tuut, tuut!");
  }
}

class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}

class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
    Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```

Polymorphism:

Polymorphism and Overriding Methods

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Inheritance lets us inherit fields and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}
```

Interfaces

Another way to achieve abstraction in C#, is with interfaces.

An interface is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies):

Example

```
// interface
interface Animal
{
  void animalSound(); // interface method (does not have a body)
  void run(); // interface method (does not have a body)
}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class. To implement an interface, use the : symbol (just like with inheritance). The body of the interface method is provided by the "implement" class. Note that you do not have to use the override keyword when implementing an interface:

Example

```
// Interface
interface IAnimal
{
  void animalSound(); // interface method (does not have a body)
```

```
}

// Pig "implements" the IAnimal interface
class Pig : IAnimal
{
  public void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The pig says: wee wee");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
  }
}
```

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interfaces can contain properties and methods, but not fields/variables
- Interface members are by default abstract and public
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface IFirstInterface
{
  void myMethod(); // interface method
}

interface ISecondInterface
{
  void myOtherMethod(); // interface method
}

// Implement multiple interfaces
class DemoClass : IFirstInterface, ISecondInterface
{
  public void myMethod()
```

```
  {
    Console.WriteLine("Some text..");
  }
  public void myOtherMethod()
  {
    Console.WriteLine("Some other text...");
  }
}

class Program
{
  static void Main(string[] args)
  {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Access Modifiers

By now, you are quite familiar with the `public` keyword that appears in many of our examples:

```
public string color;
```

The `public` keyword is an **access modifier**, which is used to set the access level/visibility for classes, fields, methods and properties.

C# has the following access modifiers:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the same class
`protected`	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter
`internal`	The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter

There's also two combinations: `protected internal` and `private protected`.

For now, lets focus on `public` and `private` modifiers.

Private Modifier

If you declare a field with a `private` access modifier, it can only be accessed within the same class:

Example

```
class Car

{
  private string model = "Mustang";

  static void Main(string[] args)
```

```
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

If you try to access it outside the class, an error will occur:

Example

```
class Car
{
  private string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
'Car.model'     is     inaccessible     due     to     its     protection     level
The field 'Car.model' is assigned but its value is never used
```

Public Modifier

If you declare a field with a `public` access modifier, it is accessible for all classes:

Example

```
class Car
{
  public string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
 The output will be:
  Mustang
```

Note: By default, all members of a class are `private` if you don't specify an access modifier:

Example

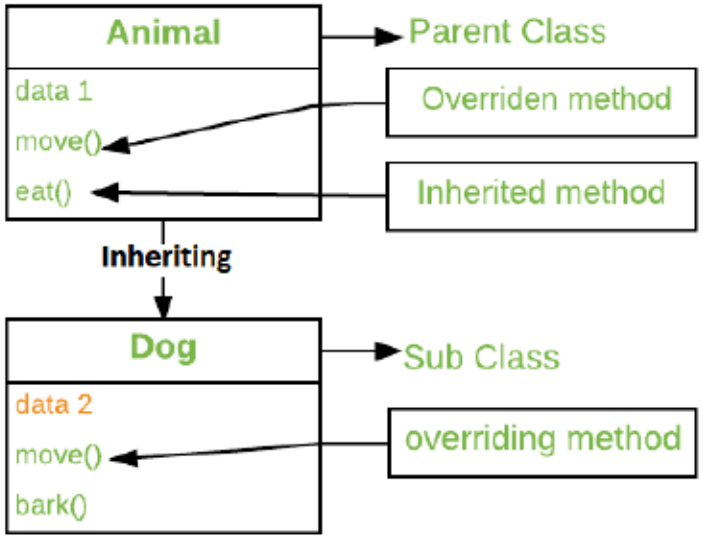
```
class Car
{
  string model;  // private
  string year;   // private
}
```

Method Overriding:

Method Overriding in C# is similar to the **virtual function in C++**. Method Overriding is a technique that allows the invoking of functions from another class (base class) in the derived class. Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.

In simple words, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class. Method overriding is one of the ways by which C# achieve *Run Time Polymorphism(Dynamic Polymorphism)*.

The method that is overridden by an override declaration is called the overridden base method. An override method is a new implementation of a member that is inherited from a base class. The overridden base method must be virtual, abstract, or override.



Example:-

```
class base_class
{
    public void gfg();
}

class derived_class : base_class
{
    public void gfg();
}

class Main_Method
{
 static void Main()
 {
    derived_class d = new derived_class();
    d.gfg();
 }
}
```

Here the base class is inherited in the derived class and the method *gfg()* which has the same signature in both the classes, is overridden.

In C# we can use 3 types of keywords for Method Overriding:

- **virtual keyword:** This modifier or keyword use within base class method. It is used to modify a method in *base class* for *overridden* that particular method in the derived class.
- **override:** This modifier or keyword use with derived class method. It is used to modify a *virtual* or *abstract* method into *derived class* which presents in base class.

```
class base_class
{
    public virtual void gfg();
}

class derived_class : base_class
{
    public override void gfg();
}
class Main_Method
{
 static void Main()
 {
    derived d_class = new derived_class();
    d.gfg();

      base_class b = new derived_class();
        b.gfg();
 }
}
```

here first, *d* refers to the object of the class *derived_class* and it invokes *gfg()* of the class *derived_class* then, *b* refers to the reference of the class base and it hold the object of class derived and it invokes *gfg()* of the class derived. Here *gfg()* method takes permission from base class to overriding the method in derived class.

Method Overloading:

Method Overloading is the common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name. C# can distinguish the methods with **different method signatures**. i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class.

- Overloaded methods are differentiated based on the number and type of the parameters passed as arguments to the methods.
- You can not define more than one method with the same name, Order and the type of the arguments. It would be compiler error.
- The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile-time error. If both methods have the same parameter types, but different return type, then it is not possible.

Why do we need Method Overloading ??

If we need to do the same kind of the operation in different ways i.e. for different inputs. In the example described below, we are doing the addition operation for different inputs. It is hard to find many different meaningful names for single action.

Different ways of doing overloading methods-
Method overloading can be done by changing:

1. The number of parameters in two methods.
2. The data types of the parameters of methods.
3. The Order of the parameters of methods.

```
// C# program to demonstrate the function
// overloading by changing the Number
// of parameters
using System;
class GFG {

    // adding two integer values.
    public int Add(int a, int b)
    {
        int sum = a + b;
        return sum;
    }

    // adding three integer values.
    public int Add(int a, int b, int c)
    {
        int sum = a + b + c;
        return sum;
    }

    // Main Method
    public static void Main(String[] args)
    {

        // Creating Object
        GFG ob = new GFG();

        int sum1 = ob.Add(1, 2);
        Console.WriteLine("sum of the two "
                          + "integer value : " + sum1);

        int sum2 = ob.Add(1, 2, 3);
        Console.WriteLine("sum of the three "
                          + "integer value : " + sum2);
    }
}
```

Output:

```
sum of the two integer value : 3
sum of the three integer value : 6
```

Sealed Class:

Sealed classes are used to restrict the users from inheriting the class. A class can be sealed by using the ***sealed*** keyword. The keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class.

The following is the **syntax** of a sealed class :

```
sealed class class_name
{
    // data members
    // methods
```

```
    .
    .
    .
}
```

A method can also be sealed, and in that case, the method cannot be overridden. However, a method can be sealed in the classes in which they have been inherited. If you want to declare a method as sealed, then it has to be declared as **virtual** in its base class.

The following class definition defines a sealed class in C#:

In the following code, create a sealed class **SealedClass** and use it from Program. If you run this code then it will work fine.

```
// C# code to define
// a Sealed Class
using System;

// Sealed class
sealed class SealedClass {

    // Calling Function
    public int Add(int a, int b)
    {
        return a + b;
    }
}

class Program {

    // Main Method
    static void Main(string[] args)
    {

        // Creating an object of Sealed Class
        SealedClass slc = new SealedClass();

        // Performing Addition operation
        int total = slc.Add(6, 4);
        Console.WriteLine("Total = " + total.ToString());
    }
}
```

Output :

```
Total = 10
```