# Object-oriented programming (OOP) Lecture 6:Polymorphism

* The word polymorph means many forms.
* We have seen the concept of multiple constructors in the previous lectures which is an example of polymorphism
* In C#, there are **two kind of polymorphism**: **static polymorphism** and **dynamic polymorphism**.

**Static polymorphism**

* It is a kind of polymorphism where the role of a method is determined at compilation time.
* Examples of static polymorphism include **method overloading** and **operator overloading**.

**Method overloading**

Let's take a look at an example of method overloading:

public class Calculator

{

public int AddNumbers(int firstNum, int secondNum)

{

return firstNum + secondNum;

}

public double AddNumbers(double firstNum, double secondNum)

{

return firstNum + secondNum;

}

}

In the above example code:

* we can see that we have two methods with the same name, **AddNumbers**. Normally, we can't have two methods that have the same name; however, as the parameters of those methods are different, methods are allowed to have the same name by the compiler.
* Writing a method with the same name as another method, but with different parameters, is called **method overloading**. This is a kind of polymorphism.

**Operator overloading**

* It’s an approach of defining multiple behaviors to an operator and these behaviors will vary based on the operands between which the operator is used.
* For example, '+' is an overloaded operator that works as addition operator when used between numeric operands and works as concatenation operator when used between string operands.

numeric + numeric => numeric

string + string = > concatenation

* if an operator is performing any action, this means that the logic is implemented internally under the libraries.
* for example

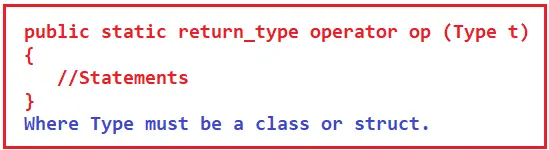
int x=10;

int y=20;

int z=x+y;

* in the above case '+' operator is adding the values of x and y and assigning it to z but internally to complete this action a special method known as **operator** method will be called.
* operator methods are defined in the base class libraries, so that they can be apply or used better different operand types

##### The syntax for C# Operator Overloading



In the above syntax

1. The return type is the return type of the function.
2. the operator is a keyword.
3. Op is the symbol of the operator that we want to overload. Like: +, <, -, ++, etc.
4. The type must be a class or struct. It can also have more parameters.
5. It should be a static function.

Let's look at an example of operator overloading: the example is to sum two matrix of size 2\*2

class Matrix

{

//declearing attributes for a 2\*2 matrix

int a, b, c, d;

public Matrix(int a, int b, int c, int d)

{

// initializing the matrix attributes.

this.a = a; this.b = b; this.c = c; this.d = d;

}

//defining an operator method for using '+' operator

//between two matrix operands

public static Matrix operator +(Matrix mx1, Matrix mx2)

{

Matrix mx3 = new Matrix(mx1.a + mx2.a, mx1.b + mx2.b, mx1.c + mx2.c, mx1.d + mx2.d);

return mx3;

}

//overriding the ToString() method inherited from object class

//to return values associated with matrix in place of class Name

public override string ToString()

{

return a + " " + b + "\n" + c + " " + d + "\n";

}

}

public class TestMatrix

{

static void Main()

{

Matrix m1 = new Matrix(10, 20, 30, 40);

Matrix m2 = new Matrix(100, 200, 300, 400);

Matrix m3 = m1 + m2;

Console.WriteLine(m3);

}

}

**Note1:**

* Whenever we pass the instance of any class to the WriteLine() method then it internally calls the WriteLine() method which takes object as a parameter and inside that method it calls ToString() on the class instance which returns the name of the class and that value gets printed. But if we don't like the printed output (the class name of the instance), then we can change the output by overriding the ToString() method under our class

**Note2:**

* Every class by default inherits form the object class, so in every class there are four method inherited from the Object class they are

1. Equals()
2. GetHashCode()
3. GetType()
4. ToString()

**Dynamic polymorphism**,

* The role of a method is determined at runtime.
* Dynamic polymorphism refers to the use of the abstract class. When you write an abstract class, no instance can be created from that abstract class. When any other class uses or implements that abstract class, the class also has to implement the abstract methods of that abstract class.
* As different classes can implement the abstract class and can have different implementations of abstract methods, polymorphic behavior is achieved. In this case, we have methods with the same name but different implementations. This is called **method overriding**.

## Sealed Class and Sealed Method

Sealed Class

* If a class is declared by using "**sealed**" modifier then it's known as sealed class and a sealed class can't be inherited by any other class.
* Example

public sealed class SealedDemo

{

//...

}

public class DerivedCalass:SealedDemo //invalid inheritance

{

//...

}

Sealed Method

* A method which can't be overridden under the child class is known as a **sealed method**.
* By default every method is a sealed method because we can never override any method until and unless it is declared as **virtual**.

## Virtual Methods

* If a method is declared as **virtual** under a class then any child class of it in the linear hierarchy will get a right to override that method without any restrictions.
* Example

Class1

public virtual void Show()

Class2: Class1

public override void Show()

Class3:Class2

public override void Show() //**valid**

**Note 1**

* In the above case even if Class2 is not overriding the method, Class3 can override the method.

**Note 2**

* A child class while overriding any of its parent class virtual methods can also sealed the method by using "sealed" modifier, so that further overriding of that method will not be possible.

Class1

public virtual void Show()

Class2: Class1

public **sealed** override void Show()

Class3:Class2

public override void Show() //**invalid**

Complete Example for virtual method

public class Class1

{

public virtual void Show()

{

Console.WriteLine("From Show method of Class1");

}

}

public class Class2:Class1

{

public override void Show()

{

Console.WriteLine("from Show method of Class2");

}

}

public class Class3 : Class2

{

public override void Show()

{

Console.WriteLine("from Show method of Class3");

}

}

public class Test

{

static void Main(string[] args)

{

Class3 c3= new Class3();

c3.Show();

}

}

## Difference between method overloading and method overriding

|  |  |
| --- | --- |
| Method **Overloading** | Method **Overriding** |
| Creating more than one method or function having same name but different signatures or the parameters in the same class is called method overloading. | Creating a method in the derived class with the same signature as a method in the base class is called as method overriding |
| It is called the compile time polymorphism | It is called runtime polymorphism |
| It has the same method name but with different signatures or the parameters | It must have same method name as well as the signatures or the parameters. |
| Method overloading doesn’t need inheritance | Method overriding needs inheritance |
| Method overloading is possible in single class only | Method overriding needs hierachy level of the classes i.e. one parent class and other child class. |
| Access modifier can be any | Access modifier must be public. |
| Method overloading is also called early binding. | Method overriding is also called late binding. |

## Abstract method and abstract class

* A method without any method body is known as abstract method, which contains only declaration for the method without any implementation (method body)
* To declare an abstract method we need to use an abstract modifier on it.
* The class under which we declare abstract method are also known as **abstract classes** which should be also declared by using abstract modifier as shown in the example below

abstract class Math

{

public abstract void Add(int x, int y);

}

* Each and every abstract method in the abstract class **must be implemented** by the child class without fail.
* The concept of **abstract method** is similar to the concept of **method overriding**. In case of method overriding if a parent class contains any method declared as **virtual** then the child class **can** **re-implement** the method **if required** by using the **override** modifier, whereas in case of **abstract methods** if the parent class contains any method declared as abstract then its child class **must implement those methods without fail**.

|  |  |
| --- | --- |
| **Method overriding** | **Abstract method** |
| optional  override  virtual | must  override  abstract |

* An **abstract class** can **contain both abstract and non-abstract** methods.
* If at all a child class of an abstract class wants to consume the non-abstract methods of its parent class it has to first implement all the abstract methods of its parent class.

abstract class

abstract methods

non-abstract methods

child class of an abstract class

first implement all the abstract methods of the parent class.

Then only we can consume the non-abstract methods of the parent class

Complete example of abstract class and abstract methods

using System;

namespace PolymorphismDemo

{

abstract class AbsParent

{

//non-abstract mehod

public void Add(int x, int y)

{

Console.WriteLine(x + y);

}

//abstract method

public abstract void Sub(int x, int y);

}

class AbsChild : AbsParent

{

public override void Sub(int x, int y)

{

Console.WriteLine(x - y);

}

static void Main(string[] args)

{

AbsChild c = new AbsChild();

c.Add(1, 2);

c.Sub(10, 2);

}

}

}

**Note**

* Even if the instance of an abstract class can not be created, it is still possible to **create a reference of abstract class** by using the **child class instance** and with that reference we can call the non-abstract methods of the abstract class as well as the abstract methods of the abstract class that are implemented under the child class.
* To test this rewrite the code under Main() method in the AbsChild class as following

AbsChild c = new AbsChild();

AbsParent p = c;

p.Add(20, 30);

p.Sub(40, 5);