Polymorphism

Polymorphism is considered one of the main pillars of object-oriented programming, along with encapsulation and inheritance. The term *polymorphism* is derived from a Greek word meaning "many forms."

Polymorphism allows objects to be treated in different forms depending on the context in which they are used

Polymorphism Types

Polymorphism occurs when a member function behaves differently depending on the object that invokes it. It can be simply defined as **"one name, multiple forms."**

In C#, polymorphism is classified into two main types:

- **Compile-time Polymorphism (Static Binding)**
- **Run-time Polymorphism (Dynamic Binding)**

1. Compile-time Polymorphism

This type is resolved at compile time.
It includes:

- Method Overloading
- Operator Overloading

2. Run-time Polymorphism

This type is resolved at runtime.
It is achieved using:

- Method Overriding
- Virtual methods

Method Overriding

Method overriding occurs when a derived class provides its own implementation of a method that is already defined in the base class with the same name and signature. When an object is created from the derived class and the method is called, the version defined in the derived class is executed instead of the inherited base class method.

In other words, the derived class overrides the inherited method to provide specialized behavior.

For example, see Program bellow.

```
using System;

class Animal
{
    // Base class method (can be overridden)
    public virtual void Move()
    {
        Console.WriteLine("Animal is moving.");
    }

    // Normal method (not overridden here)
    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}

class Dog : Animal
{
    // Overriding the Move method from the base class
    public override void Move()
    {
        Console.WriteLine("Dog is running.");
    }

    // New method specific to Dog
    public void Bark()
    {
        Console.WriteLine("Dog is barking.");
    }
}

class Program
{
    static void Main()
    {
        Dog d = new Dog();
        d.Move();   // overridden method
        d.Eat();    // base class method
        d.Bark();   // dog-specific method
    }
{
```

Operator Overloading

We have previously discussed the concept of **overloading**, such as function overloading and constructor overloading.

In general, *overloading* means giving more than one task to the same name.

- **Function overloading** allows multiple methods with the same name but different parameter lists.
- **Constructor overloading** allows creating objects in different ways using constructors with different parameters.

Operator Overloading

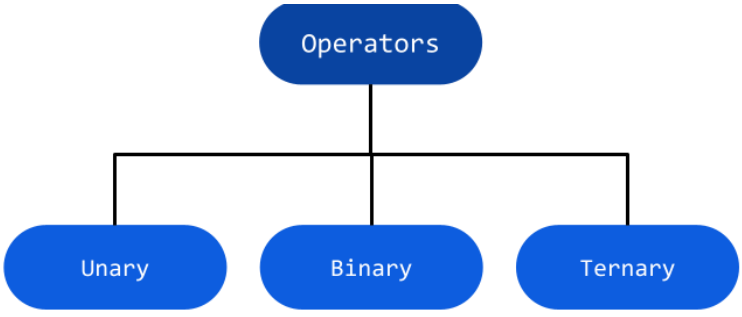
Operator overloading extends this concept to operators.

It allows redefining the behavior of operators (such as +, −, ==, etc.) when applied to user-defined classes, in addition to their built-in functionality.

Before studying operator overloading, we have already worked with different types of operators such as:

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators
- Assignment operators

Operator overloading enables us to customize the behavior of these operators when used with objects.



Types of Operators Based on Number of Operands

Operators are classified according to the number of operands they take:

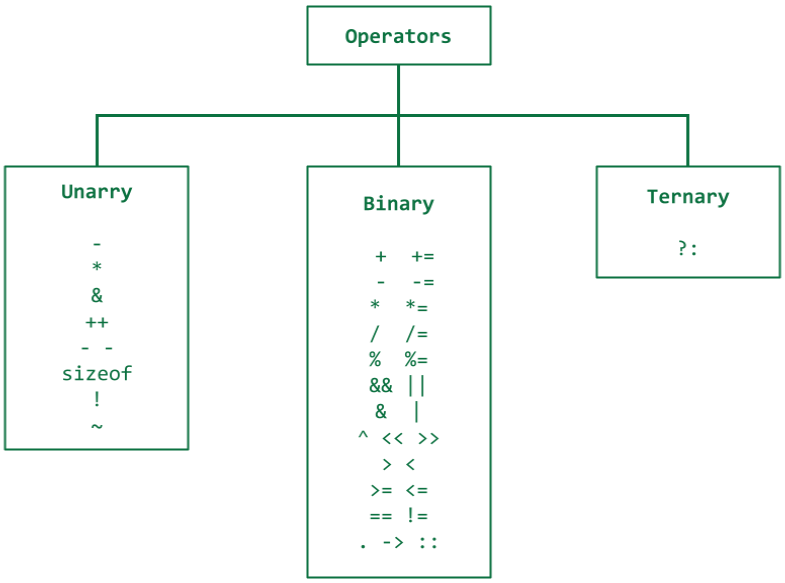
- **Unary Operator** → takes one operand.
  Example: `++A`
  The increment operator `++` is unary because it operates on one value (A).
- **Binary Operator** → takes two operands.
  Example: `A + B`
  The addition operator `+` is binary because it operates on two values (A and B).
- **Ternary Operator** → takes three operands.
  Example: `A > B ? true : false`
  The conditional operator `?:` is ternary because it works with three parts:
  condition, true result, and false result.

Important Note

Some operators can work as both unary and binary operators depending on how they are used.

For example:

- `-A` → unary (negation)
- `A - B` → binary (subtraction)



1. Unary Overloading Example

```
using System;

class Student
{
    private string Name;
    private int ID, Age, Deg1, Deg2, Deg3;

    public Student()
    {
        Console.Write("Enter student ID: ");
        ID = int.Parse(Console.ReadLine());

        Console.Write("Enter student Name: ");
        Name = Console.ReadLine();

        Console.Write("Enter student Age: ");
        Age = int.Parse(Console.ReadLine());

        Console.Write("Enter student Deg1: ");
        Deg1 = int.Parse(Console.ReadLine());

        Console.Write("Enter student Deg2: ");
        Deg2 = int.Parse(Console.ReadLine());

        Console.Write("Enter student Deg3: ");
        Deg3 = int.Parse(Console.ReadLine());
    }

    public void Print()
    {
        Console.WriteLine("Student Name: " + Name);
        Console.WriteLine("Student ID: " + ID);
        Console.WriteLine("Student Age: " + Age);
        Console.WriteLine("Student Deg1: " + Deg1);
        Console.WriteLine("Student Deg2: " + Deg2);
        Console.WriteLine("Student Deg3: " + Deg3);
    }

    // Unary ++ Operator Overloading
    public static Student operator ++(Student s)
    {
        s.Age++;
        return s;
    }
}

class Program
{
    static void Main()
    {
        Student s = new Student();
        ++s;      // Using overloaded ++ operator
        s.Print();
    }}
```

2. Binary Overloading Example

```
using System;

class Vector
{
   private int x, y;

   public Vector(int a, int b)
   {
      x = a;
      y = b;
   }

   public void Print()
   {
      Console.WriteLine("x = " + x + " y = " + y);
   }

   // Binary + Operator Overloading
   public static Vector operator +(Vector v1, Vector v2)
   {
      return new Vector(v1.x + v2.x, v1.y + v2.y);
   }
}

class Program
{
   static void Main()
   {
      Vector v1 = new Vector(1, 2);
      Vector v2 = new Vector(3, 4);

      v1.Print();
      v2.Print();

      Vector v3 = v1 + v2;   // Using overloaded + operator
      v3.Print();
   }
{
```