
C
sharp

C# is pronounced "C-Sharp".
It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.
C# has roots from the C family, and the language is close to other popular languages like C++ and Java.

C# is used for:

- **Mobile applications**
- **Desktop applications**
- **Web applications**
- **Web services**
- **Web sites**
- **Games**
- **VR**
- **Database applications**
- **And much, much more!**

- **Line 1: using System means that we can use classes from the System namespace.**
- **Line 2: A blank line. C# ignores white space. However, multiple lines makes the code more readable.**
- **Line 3: namespace is a used to organize your code, and it is a container for classes and other namespaces.**
- **Line 4: The curly braces {} marks the beginning and the end of a block of code.**
- **Line 5: class is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.**

- **Line 7: Another thing that always appear in a C# program, is the Main method. Any code inside its curly brackets {} will be executed. You don't have to understand the keywords before and after Main. You will get to know them bit by bit while reading this tutorial.**
- **Line 9: Console is a class of the System namespace, which has a WriteLine() method that is used to output/print text. In our example it will output "Hello World!".**
- **If you omit the using System line, you would have to write System.Console.WriteLine() to print/output text.**
- **Note: Every C# statement ends with a semicolon ;.**
- **Note: C# is case-sensitive: "MyClass" and "myclass" has different meaning.**

variables

- In C#, there are different types of variables (defined with different keywords), for example:
- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

Type casting

- **Implicit Casting (automatically) - converting a smaller type to a larger type size**
- **char -> int -> long -> float -> double**
-
- **Explicit Casting (manually) - converting a larger type to a smaller size type**
- **double -> float -> long -> int -> char**

```
int myInt = 9;
double myDouble = myInt;       // Automatic casting: int to double
Console.WriteLine(myInt);      // Outputs 9
Console.WriteLine(myDouble);   // Outputs 9

double myDouble = 9.78;
int myInt = (int) myDouble;    // Manual casting: double to int
Console.WriteLine(myDouble);   // Outputs 9.78
Console.WriteLine(myInt);      // Outputs 9
```

It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long):

Input & output

You have already learned that Console.WriteLine() is used to output (print) values.
Now we will use Console.ReadLine() to get user input.

```
Console.WriteLine("Enter your age:");
int age = Console.ReadLine();
Console.WriteLine("Your age is: " + age);
```

this will give you error because c sharp only gets string as input

```
Console.WriteLine("Enter your age:");
int age =Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Your age is: " + age);
```

you need to typecast to make it work

Math methods

- The Math.Max(x,y) method can be used to find the highest value of x and y
- The Math.Min(x,y) method can be used to find the lowest value of of x and y
- The Math.Sqrt(x) method returns the square root of x
- The Math.Abs(x) method returns the absolute (positive) value of x
- Math.Round() rounds a number to the nearest whole number

String interpolation

Another option of string concatenation, is string interpolation, which substitutes values of variables into placeholders in a string. Note that you do not have to worry about spaces, like with concatenation

```
string firstName = "John";
string lastName = "Doe";
string name = $"My full name is: {firstName} {lastName}";
Console.WriteLine(name);
```

```
string myString = "Hello";
Console.WriteLine(myString[0]);  // Outputs "H"
```

access string

if statement

```
if (condition){
// block of code to be executed if the condition is True
}
```

Short Hand If...Else (Ternary Operator)

```
variable = (condition) ? expressionTrue :  expressionFalse
```

switch statement

```
switch(expression){
case x:
// code blockbreak;
case y:
// code blockbreak;
default:
// code blockbreak;
}
```


OOPS

class

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

Constructor

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:

```
using System;

namespace MyApplication
{
  // 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
    }
  }
}
```

Access modifiers

ModifierDescription

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

Inheritance

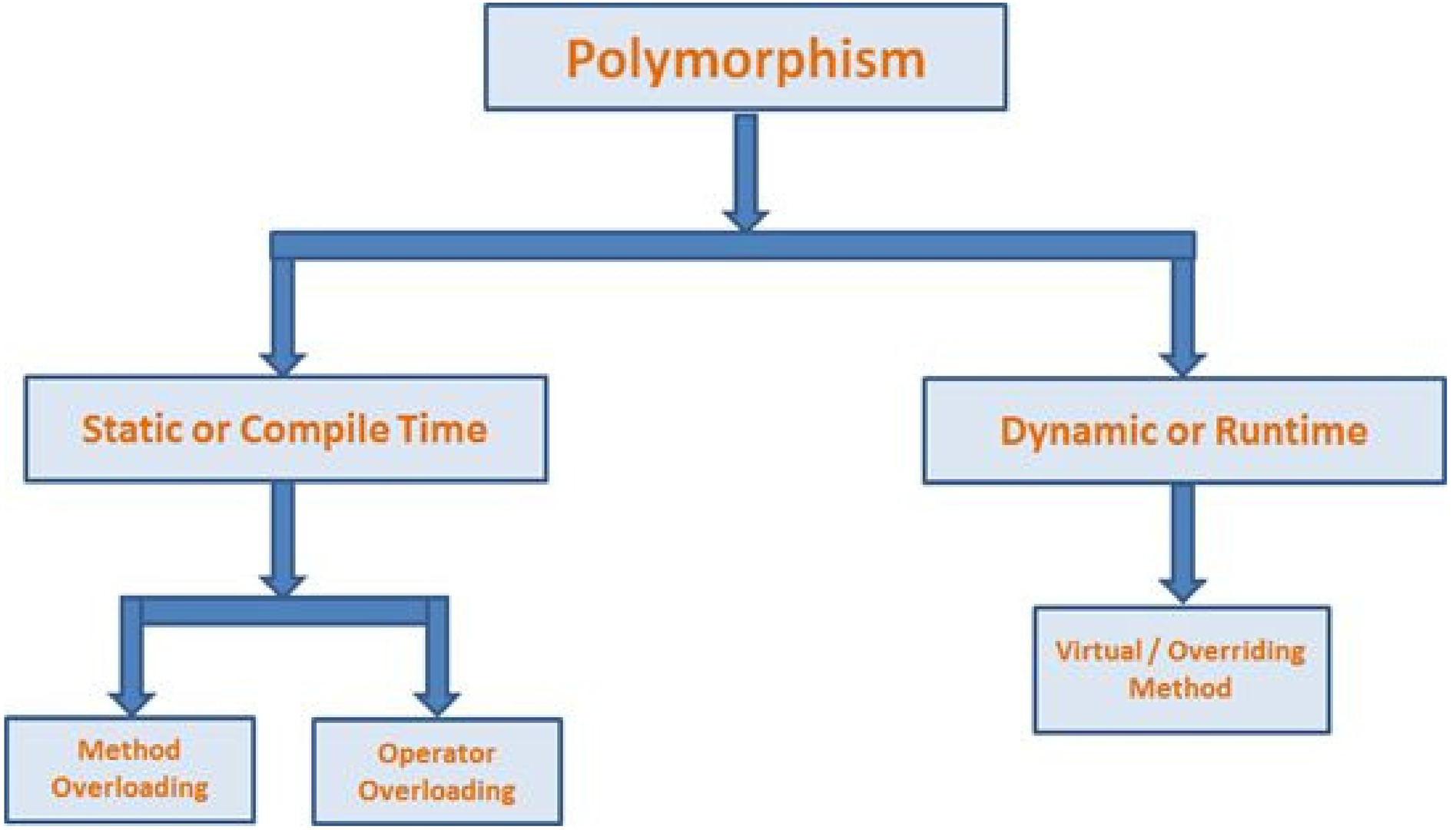
- In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:
- Derived Class (child) - the class that inherits from another class
- Base Class (parent) - the class being inherited from

```
using System;
  public class Employee
   {
      public float salary = 40000;
  }
  public class Programmer: Employee
  {
      public float bonus = 10000;
  }
  class TestInheritance{
      public static void Main(string[] args)
       {
          Programmer p1 = new Programmer();

          Console.WriteLine("Salary: " + p1.salary);
          Console.WriteLine("Bonus: " + p1.bonus);

       }
   }
```

If you don't want other classes to inherit from a class, use the sealed keyword

Polymorphism

- **Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.**
- **Like we specified in the previous chapter; 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.):**


Polymorphism
Static or Compile Time
Dynamic or Runtime
Method Overloading
Operator Overloading
Virtual / Overriding Method

Method overloading

```
public class TestData
{
    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    public int Add(int a, int b)
    {
        return a + b;
    }
}
class Program
{
    static void Main(string[] args)
    {
        TestData dataClass = new TestData();
        int add2 = dataClass.Add(45, 34, 67);
        int add1 = dataClass.Add(23, 34);
    }
}
```

The compiler requires an Area() method and it compiles successfully but the right version of the Area() method is not being determined at compile time but determined at runtime. Finally the overriding methods must have the same name and signature (number of parameters and type), as the virtual or abstract method defined in the base class method and that it is overriding in the derived class.

```
public class Drawing
{
   public virtual double Area()
  {
       return 0;
  }
}

public class Circle : Drawing
{
   public double Radius { get; set; }
   public Circle()
   {
      Radius = 5;
   }
   public override double Area()
   {
      return (3.14) * Math.Pow(Radius, 2);
   }
}

public class Square : Drawing
{
   public double Length { get; set; }
   public Square()
   {
      Length = 6;
   }
   public override double Area()
   {
      return Math.Pow(Length, 2);
   }
}
```

```
public class Rectangle : Drawing
{
 public double Height { get; set; }
 public double Width { get; set; }
 public Rectangle()
 {
 Height = 5.3;
 Width = 3.4;
 }
 public override double Area()
 {
 return Height * Width;
 }
}

class Program
{
 static void Main(string[] args)
 {

 Drawing circle = new Circle();
 Console.WriteLine("Area :" + circle.Area());

 Drawing square = new Square();
 Console.WriteLine("Area :" + square.Area());

 Drawing rectangle = new Rectangle();
 Console.WriteLine("Area :" + rectangle.Area());
 }
}
```

A method or function of the base class is available to the child (derived) class without the use of the "overriding" keyword. The compiler hides the function or method of the base class. This concept is known as shadowing or method hiding

- **A method cannot be overriden if:**
- **Methods have a different return type**
- **Methods have a different access modifier**
- **Methods have a different parameter type or order**
- **Methods are non virtual or static**

```
public class BaseClass
{
   public virtual string GetMethodOwnerName()
   {
      return “Base Class”;
   }
}
public class ChildClass : BaseClass
{
  public override string GetMethodOwnerName()
  {
      return “Child Class”;
  }
}
```

A method or function of the base class is available to the child (derived) class without the use of the "overriding" keyword. The compiler hides the function or method of the base class. This concept is known as shadowing or method hiding. In the shadowing or method hiding, the child (derived) class has its own version of the function, the same function is also available in the base class.

```
Public class BaseClass
{
  public string GetMethodOwnerName()
  {
    return "Base Class";
  }
}
public class ChildClass : BaseClass
{
  public new string GetMethodOwnerName()
  {
    return "ChildClass";
  }
}
```

Mixing Method (Overriding and shadowing (Method Hiding))

We can also use shadowing and method overriding together using the virtual and new keywords. This is useful when we want to further override a method of the child (derived) class.

```
public class BaseClass
{
  public virtual string GetMethodOwnerName()
  {
     return "Base Class";
  }
}
public class ChildClass : BaseClass
{
  public new virtual string GetMethodOwnerName()
  {
     return "ChildClass";
  }
}
public class SecondChild : ChildClass
{
  public override virtual string GetMethodOwnerName()
  {
     return "Second level Child";
  }
}
```

abstraction

Data abstraction is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either abstract classes or interfaces (which you will learn more about in the next chapter).
The abstract keyword is used for classes and methods:

- **Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).**
-
- **Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).**

```
using System;
namespace AbstractionDemo
{
  abstract class Shape
  {
    public abstract double area();
  }
  class Circle: Shape
  {
    private double radius;
    public Circle( double r)
    {
      radius = r;
    }
    public override double area ()
    {
      return (3.14*radius*radius);
    }
  }
  class Square: Shape
  {
    private double side;
    public Square( double s)
    {
      side = s;
    }
    public override double area ()
    {
      return (side*side);
    }
  }
class Triangle: Shape
  {
    private double tbase;
    private double theight;
    public Triangle( double b, double h)
    {
      tbase = b;
      theight = h;
    }
    public override double area ()
    {
      return (0.5*tbase*theight);
    }
  }
  class Test
  {
    static void Main(string[] args)
    {
      Circle c = new Circle(5.0);
      Console.WriteLine("Area of Circle = {0}", c.area());
      Square s = new Square(2.5);
      Console.WriteLine("Area of Square = {0}", s.area());
      Triangle t = new Triangle(2.0, 5.0);
      Console.WriteLine("Area of Triangle = {0}", t.area());
    }
  }
}
```

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).

```
using System;

namespace MyApplication
{
  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();
 }
 }
}
```

Enum

An enum is a special "class" that represents a group of constants (unchangeable/read-only variables).
To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma

```
using System;

namespace MyApplication
{
  enum Level
  {
    Low,
    Medium,
    High
  }
  class Program
  {
    static void Main(string[] args)
    {
      Level myVar = Level.Medium;
      Console.WriteLine(myVar);
    }
  }
}
```

By default, the first item of an enum has the value 0. The second has the value 1, and so on.
To get the integer value from an item, you must explicitly convert the item to an int

```
using System;

namespace MyApplication
{
  class Program
  {
    enum Months
    {
      January,    // 0
      February,   // 1
      March,      // 2
      April,      // 3
      May,        // 4
      June,       // 5
      July        // 6
    }
    static void Main(string[] args)
    {
      int myNum = (int) Months.April;
      Console.WriteLine(myNum);
    }
  }
}
```