C# .NET Object Oriented Programming:

3.1 Introduction

- Object-Oriented Programming is a strategy that provides some principles for developing applications or developing software.
- It is a methodology. Like OOPs, other methodologies also exist such as Structured Programming, Procedural Programming, or Modular Programming.
- But nowadays, one of the well-known and famous styles is Object Orientation i.e. Object-Oriented Programming.
- As far as the .NET platform is concerned, the most fundamental programming construct is the class type.
- Formally, a class is a user-defined type that is composed of field data (often called member variables) and members that operate on this data (such as constructors, properties, methods, events, and so forth).
- Collectively, the set of field data represents the "state" of a class instance (otherwise known as an object).
- The power of object-oriented languages, such as C#, is that by grouping data and related functionality in a unified class definition, you are able to model your software after entities in the real world.
- C# is an object-oriented program. In object-oriented programming (OOP), we solve complex problems by dividing them into objects.
- To work with objects, we need to perform the following activities:
    - ✓ Create a class
    - ✓ Create objects from the class

3.2 Objects and Classes

- Objects and classes help us to divide a large project into smaller sub-problems.
- Suppose you want to create a game that has hundreds of enemies and each of them has fields like health, ammo, and methods like shoot() and run().
- With OOP we can create a single Enemy class with required fields and methods. Then, we can create multiple enemy objects from it.
- Each of the enemy objects will have its own version of health and ammo fields. And, they can use the common shoot() and run() methods.
- Now, instead of thinking of projects in terms of variables and methods, we can think of them in terms of objects.
- This helps to manage complexity as well as make our code reusable.

3.2.1 C# Class

- A class is a blueprint of an object that contains variables for storing data and functions to perform operations on the data.
- A class will not occupy any memory space and hence it is only a logical representation of data.

3.2.2 Create a class in C#

- We use the class keyword to create an object. For example,

```
class ClassName {

}
```

- Here, we have created a class named ClassName. A class can contain

  - ✓ fields - variables to store data
  - ✓ methods - functions to perform specific tasks

Example:

```
class college {

  //field
  string category;

  //method
  public void course() {

  }
 }
```

College=class name
Category field
Course- method.

C# Objects

- An object is an instance of a class. Suppose, we have a class Dog. Bulldog, German Shepherd, Pug are objects of the class.

3.2.3 Creating an Object of a class

- In C#, here's how we create an object of the class.

  ```
  ClassName obj = new ClassName();
  ```

- Here, we have used the new keyword to create an object of the class.
- And, obj is the name of the object. Now, let us create an object from the Dog class.

```
College artsCollege  = new College();
```

- Now, the artsCollege object can access the fields and methods of the Dog class.

3.2.4 Access Class Members using Object

- We use the name of objects along with the . operator to access members of a class. For example,

```
using System;

namespace ClassObject {

  class College {
    string category;

    public void course() {
      Console.WriteLine("BCA, MCA");

    }

    static void Main(string[] args) {

      // create object
      College UnivCollege  = new College();

      // access nature of the College
     UnivCollege.nature = "Arts College";
      Console.WriteLine(UnivCollege.nature);

      // access method of the College
      UnivCollege.course ();

      Console.ReadLine();

    }
  }
}
```

3.2.5 Creating Multiple Objects of a Class

```
using System;
namespace ClassObject {
  class Employee {
    string department;
    static void Main(string[] args) {
      // create Employee object
      Employee sheeran = new Employee();
      // set department for sheeran
      sheeran.department = "Development";
      Console.WriteLine("Sheeran: " + sheeran.department);
      // create second object of Employee
      Employee taylor = new Employee();
      // set department for taylor
      taylor.department = "Content Writing";
      Console.WriteLine("Taylor: " + taylor.department);
      Console.ReadLine();
    }
  }
}
```

3.2.6 Creating objects in a different class

In C#, we can also create an object of a class in another class.

```
using System;
namespace ClassObject {
  class Employee {
    public string name;
    public void work(string work) {
      Console.WriteLine("Work: " + work);

    }
  }
  class EmployeeDrive {
    static void Main(string[] args) {
      // create Employee object
      Employee e1= new Employee();
      Console.WriteLine("Employee 1");
      // set name of the Employee
      e1.name="Gloria";
      Console.WriteLine("Name: " + e1.name);
      //call method of the Employee
      e1.work("Coding");
      Console.ReadLine();
  }
  }
}
```

- In the above example, we have two classes: Employee and EmployeeDrive. Here, we are creating an object e1 of the Employee class in the EmployeeDrive class.
- We have used the e1 object to access the members of the Employee class from EmployeeDrive. This is possible because the members in the Employee class are public.
- Here, public is an access specifier that means the class members are accessible from any other classes.

3.4 C# Method

A method is a block of code that performs a specific task. Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

- ✓ a method to draw the circle
- ✓ a method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

Declaring a Method in C#

Here's the syntax to declare a method in C#.

```
returnType methodName() {
  // method body
}
```

- ➢ Here,
    - ✓ returnType - It specifies what type of value a method returns. For example, if a method has an int return type then it returns an int value.
- ➢ If the method does not return a value, its return type is void.
    - ✓ methodName - It is an identifier that is used to refer to the particular method in a program.
    - ✓ method body - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }

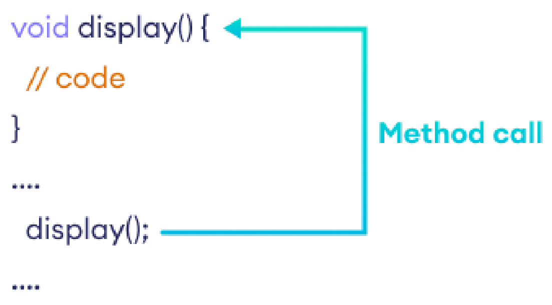
```
void display() {
// code
}
```

- ➢ Here, the name of the method is display(). And, the return type is void.

3.4.1 Calling a Method in C#

- ➢ In the above example, we have declared a method named display(). Now, to use the method, we need to call it.
- ➢ Here's how we can call the display() method.

```
// calls the method

display();
```



3.4.2 Working of C# method call

Example: C# Method

```
using System;
namespace Method {
  class Program {
    // method declaration
    public void display() {
      Console.WriteLine("Method created");
    }
    static void Main(string[] args) {
      // create class object
      Program p1 = new Program();
      //call method
      p1.display();
     Console.ReadLine();
    }
  }
}
```

Output

```
Method created
```

In the above example, we have created a method named display(). We have created an object p1 of the Program class.

Notice the line,

```
p1.display();
```

Here, we are using the object to call the display() method.

3.4.3 C# Method Return Type

- A C# method may or may not return a value. If the method doesn't return any value, we use the void keyword (shown in the above example).
- If the method returns any value, we use the return statement to return any value. For example,

```
int addNumbers() {
  ...
  return sum;
}
```

- Here, we are returning the variable sum. One thing you should always remember is that the return type of the method and the returned value should be of the same type.
- In our code, the return type is int. Hence, the data type of sum should be of int as well.

Example: Method Return Type

```
using System;
namespace Method {

  class Program {

    // method declaration
    static int addNumbers() {
      int sum = 5 + 14;
      return sum;
         }
    static void Main(string[] args) {
      // call method
      int sum = addNumbers();
      Console.WriteLine(sum);
      Console.ReadLine();
    }
  }
}
```

Output

```
19
```

- In the above example, we have a method named addNumbers() with the int return type.
- int sum = addNumbers();
- Here, we are storing the returned value from the addNumbers() to sum. We have used int data type to store the value because the method returns an int value.
- Note: As the method is static we do not create a class object before calling the method. The static method belongs to the class rather than the object of a class.

3.5 C# Methods Parameters

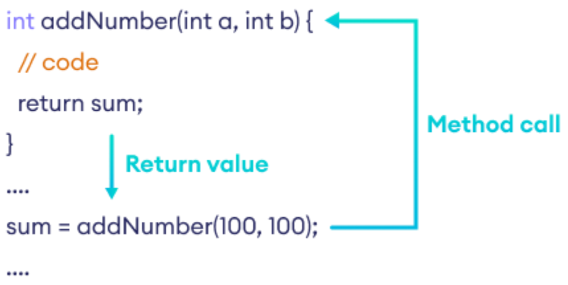
- In C#, we can also create a method that accepts some value. These values are called method parameters. For example,

```
int addNumber(int a, int b) {
//code
}
```

Here, a and b are two parameters passed to the addNumber() function.

- If a method is created with parameters, we need to pass the corresponding values(arguments) while calling the method. For example,

```
// call the method
addNumber(100, 100);
```



```
int addNumber(int a, int b) {
  // code
  return sum;
}
....
sum = addNumber(100, 100);
....
```

- Representation of the C# method returning a value Here, We have passed 2 arguments (100, 100).

Example 1: C# Methods with Parameters

```
using System;
namespace Method {
  class Program {
    int addNumber (int a, int b) {
       int sum = a + b;
      return sum;
     }
    static void Main(string[] args) {

      // create class object
      Program p1 = new Program();
      //call method
      int sum = p1.addNumber(100,100);
      Console.WriteLine("Sum: " + sum);
       Console.ReadLine();
       }
  }
}
```

Output

```
Sum: 200
```

3.5.1 C# Methods with Single Parameter

- In C#, we can also create a method with a single parameter. For example,

```
using System;
namespace Method {
  class Program {
    string work(string work) {
     return work;
    }
    static void Main(string[] args) {
      // create class object
      Program p1 = new Program();
      //call method
      string work = p1.work("Cleaning"); ;
      Console.WriteLine("Work: " + work);
      Console.ReadLine();
         }
  }
}
```

Output

```
Work: Cleaning
```

Here, the work() method has a single parameter work.

3.5.2 Built-in methods

- So far we have defined our own methods. These are called **user-defined methods**.

- However, in C#, there are various methods that can be directly used in our program. They are called **built-in methods**. For example,

  - ✓ Sqrt() - computes the square root of a number
  - ✓ ToUpper() - converts a string to uppercase

Example: Math.Sqrt() Method

```
using System;
namespace Method {
  class Program {
    static void Main(string[] args) {
      // Built in method
      double a = Math.Sqrt(9);
      Console.WriteLine("Square root of 9: " + a);
    }
  }
}
```

Output

```
Square root of 9: 3
```

- In the above program, we have used
- double a = Math.Sqrt(9); to compute the square root of 9. Here, the Sqrt() is a built-in method that is defined inside the Math class.
- We can simply use built-in methods in our program without writing the method definition. To learn more, visit C# built-in methods.

3.6 Method Overloading in C#

- In C#, we can create two or more methods with the same name. It is known as method overloading. For example,

```
using System;

namespace MethodOverload {

  class Program {

    // method with one parameter
    void display(int a) {
      Console.WriteLine("Arguments: " + a);
    }

    // method with two parameters
    void display(int a, int b) {
      Console.WriteLine("Arguments: " + a + " and " + b);
    }
    static void Main(string[] args) {

      Program p1 = new Program();
      p1.display(100);
      p1.display(100, 200);
      Console.ReadLine();
    }
  }
}
```

Output

```
Arguments: 100
Arguments: 100 and 200
```

- In the above example, we have overloaded the display() method. It is possible because:
  - ✓ one method has one parameter
  - ✓ another has two parameter

3.7 C# CONSTRUCTORS:

Constructor is a special type of method which is automatically executed when the object is created. A constructor has exactly the same name as that of the class name and it does not have any return type. Constructors are responsible for object initialization and memory allocation of its class. If we create any class without constructor, the compiler will automatically create one default constructor for that class. The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.
Some of the key points regarding the Constructor are:

- ✓ A class can have any number of constructors.
- ✓ A constructor doesn't have any return type, not even void.
- ✓ A static constructor cannot be a parameterized constructor.
- ✓ Within a class you can create only one static constructor

3.7.1 Types of Constructors:

Basically constructors are 5 types those are

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

3.7.1.1 Default Constructor:

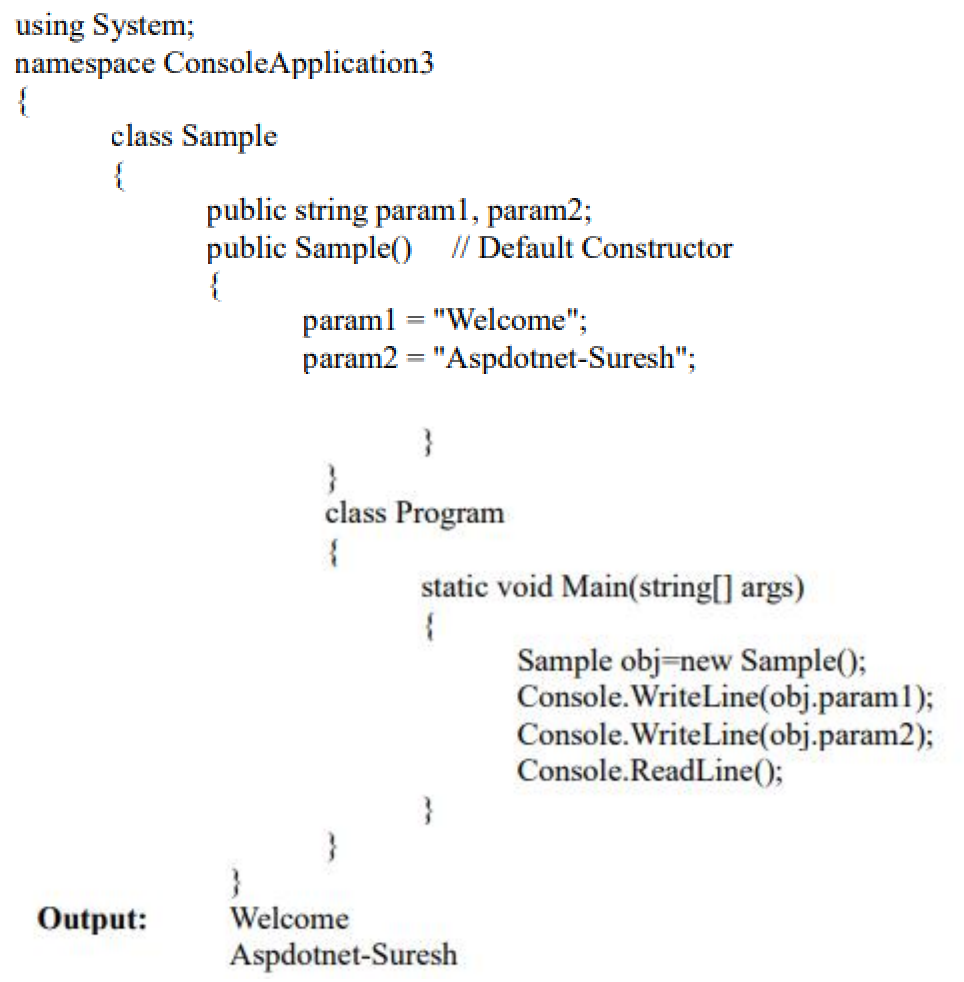
A constructor without any parameters is called a default constructor; in other words this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class to different values.

The default constructor initializes.

1. All numeric fields in the class to zero.

2. All string and object fields to null.

Example:

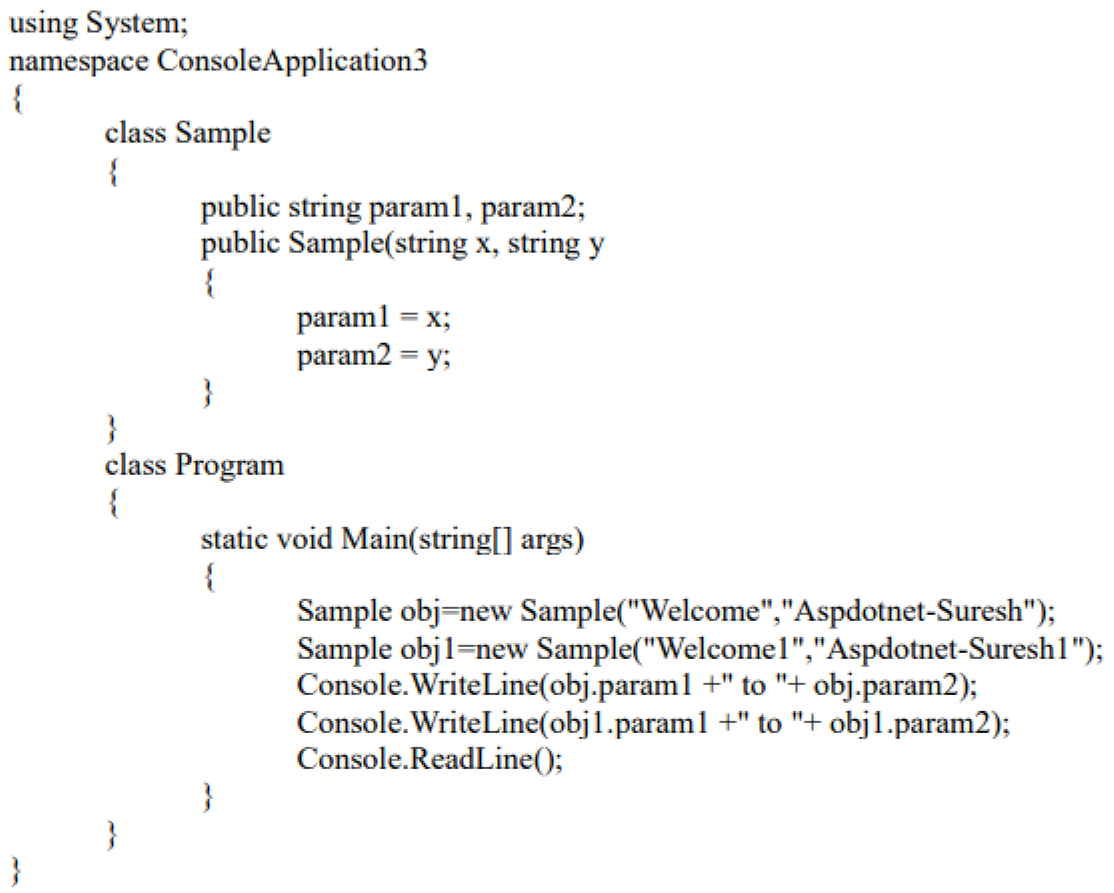


```
using System;
namespace ConsoleApplication3
{
        class Sample
        {
                public string param1, param2;
                public Sample()     // Default Constructor
                {
                        param1 = "Welcome";
                        param2 = "Aspdotnet-Suresh";

                        }
                }
                class Program
                {
                        static void Main(string[] args)
                        {
                                Sample obj=new Sample();
                                Console.WriteLine(obj.param1);
                                Console.WriteLine(obj.param2);
                                Console.ReadLine();
                        }
                }
        }
Output:         Welcome
                Aspdotnet-Suresh
```

3.7.1.2 Parameterized Constructors:

A constructor with at least one parameter is called as parameterized constructor. The advantage of a parameterized constructor is that you can initialize each instance of the class to different values.

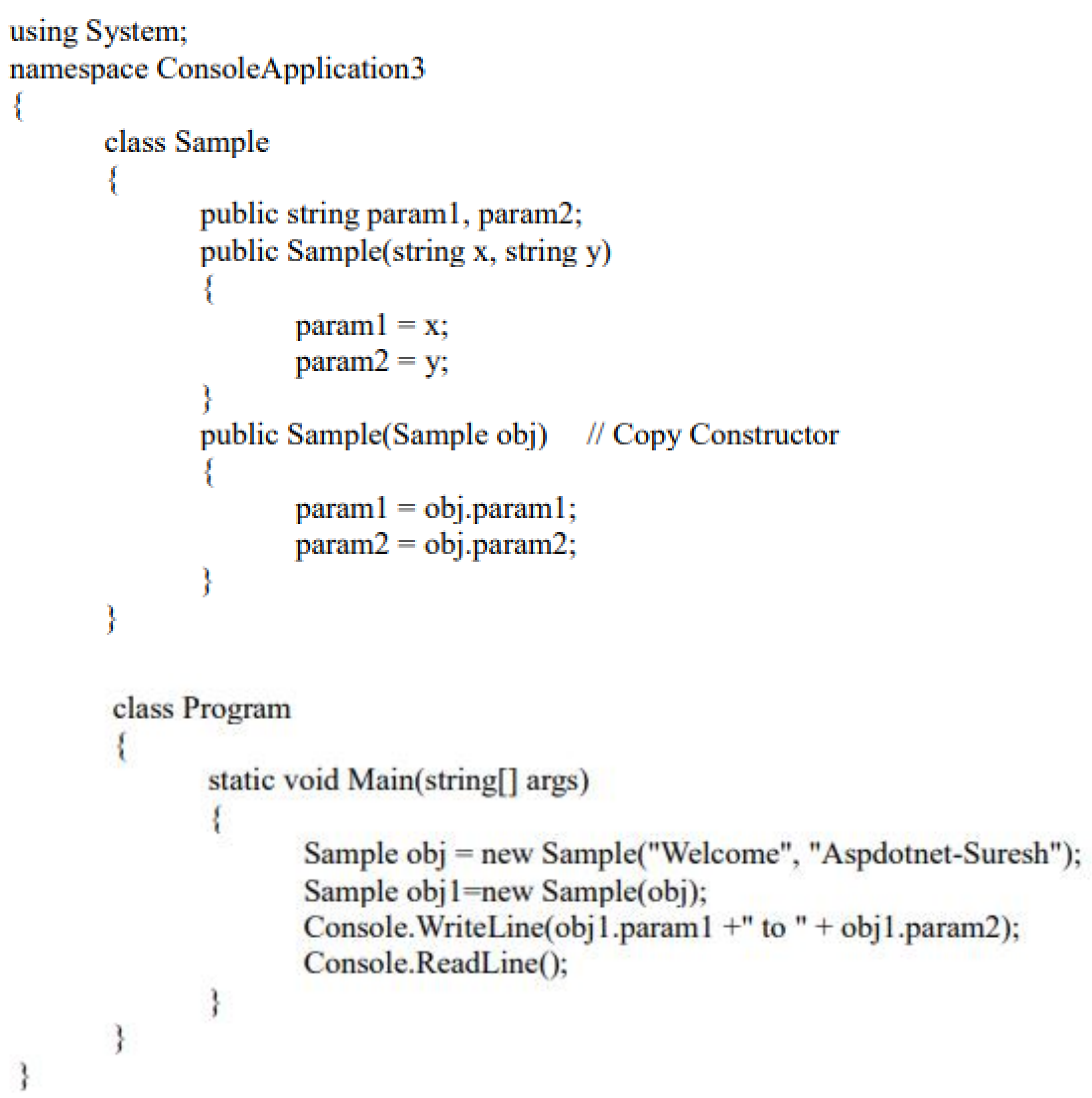
Example:



```
using System;
namespace ConsoleApplication3
{
        class Sample
        {
                public string param1, param2;
                public Sample(string x, string y
                {
                        param1 = x;
                        param2 = y;
                }
        }
        class Program
        {
                static void Main(string[] args)
                {
                        Sample obj=new Sample("Welcome","Aspdotnet-Suresh");
                        Sample obj1=new Sample("Welcome1","Aspdotnet-Suresh1");
                        Console.WriteLine(obj.param1 +" to "+ obj.param2);
                        Console.WriteLine(obj1.param1 +" to "+ obj1.param2);
                        Console.ReadLine();
                }
        }
}
```

3.7.1.4 Copy Constructor:

The constructor which creates an object by copying variables from another object is called a copy constructor. Main purpose of copy constructor is to initialize new object to the values of an existing object. Here the constructor contains a parameter, which is of class type.



```
using System;
namespace ConsoleApplication3
{
        class Sample
        {
                public string param1, param2;
                public Sample(string x, string y)
                {
                        param1 = x;
                        param2 = y;
                }
                public Sample(Sample obj)    // Copy Constructor
                {
                        param1 = obj.param1;
                        param2 = obj.param2;
                }
        }

        class Program
        {
                static void Main(string[] args)
                {
                        Sample obj = new Sample("Welcome", "Aspdotnet-Suresh");
                        Sample obj1=new Sample(obj);
                        Console.WriteLine(obj1.param1 +" to " + obj1.param2);
                        Console.ReadLine();
                }
        }
}
```

3.7.1.5 Static Constructor:

A static constructor is used to initialize any static data (or) to perform a particular action that needs to be performed once only.
Static constructor will be invoked only once for all of objects of the class. Static constructor gets called before the first object of the class is created.
Importance points of static constructor

- ✓ Static constructor will not accept any parameters because it is automatically called by CLR.
- ✓ Static constructor will not have any access modifiers.
- ✓ Static constructor will execute automatically whenever we create first instance of class
- ✓ Only one static constructor will allowed

Syntax
class employee
{

```
   // Static constructor
   static employee(){}
}
```

Example

```
using System;
namespace staticConstractor
{
   public class employee
   {
      static employee() // Static constructor
      declaration{Console.WriteLine("The static constructor ");
   }
   public static void Salary()
   {
      Console.WriteLine();
      Console.WriteLine("The Salary method");
   }
}
class details
{
   static void Main()
   {
      Console.WriteLine("----------Static constrctor example by vithal wadje------------------");
      Console.WriteLine();
      employee.Salary();
      Console.ReadLine();
   }
  }
}
```

3.7.1.6 Private Constructor

- When a constructor is created with a private specifiers, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class.
- They are usually used in classes that contain static members only. Some key points of a private constructor are:

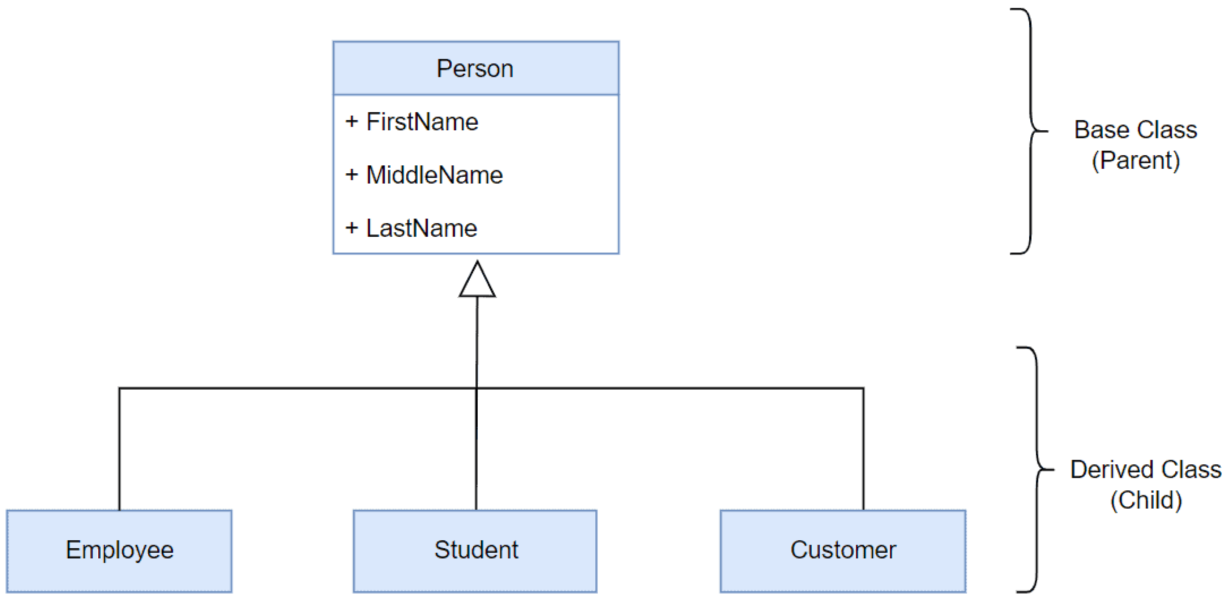
1. One use of a private constructor is when we have only static members.
2. It provides an implementation of a singleton class pattern
3. Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.

```
using System;
namespace defaultConstractor
```

```
{
    public class Counter
    {
        private Counter()   //private constrctor declaration
        {
        }
        public static int currentview;
        public static int visitedCount()
        {
            return ++ currentview;
        }
    }
    class viewCountedetails
    {
        static void Main()
        {
            // Counter aCounter = new Counter();   // Error
            Console.WriteLine("-------Private constructor example by vithal wadje----------");
            Console.WriteLine();
            Counter.currentview = 500;
            Counter.visitedCount();
            Console.WriteLine("Now the view count is: {0}", Counter.currentview);
            Console.ReadLine();
        }
    }
}
```

3.8 Inheritance in C#

- In object-oriented programming, inheritance is another type of relationship between classes. Inheritance is a mechanism of reusing the functionalities of one class into another related class.
- Inheritance is referred to as "is a" relationship. In the real world example, a customer is a person. In the same way, a student is a person and an employee is also a person.
- They all have some common things, for example, they all have a first name, middle name, and last name.
- So to translate this into object-oriented programming, we can create the Person class with first name, middle name, and last name properties and inherit the Customer, Student, and Employee classes from the Person class.
- Note that the inheritance can only be used with related classes where they should have some common behaviors and perfectly substitutable



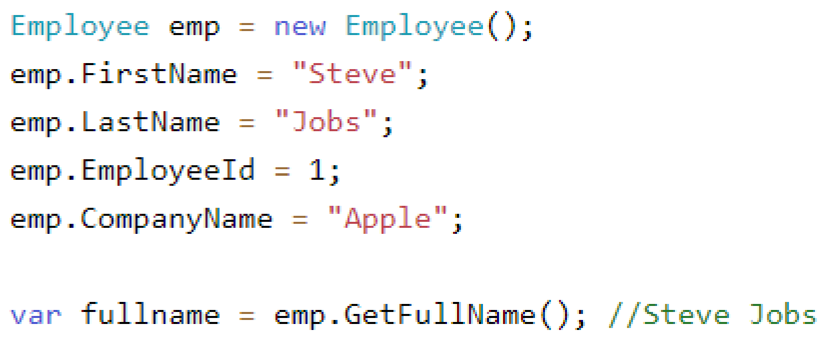
- In C#, use the : symbol to inherit a class from another class. For example, the following Employee class inherits from the Person class in C#.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string GetFullName(){
        return FirstName + " " + LastName;
    }
}

class Employee : Person
{
    public int EmployeeId { get; set; }
    public string CompanyName { get; set; }

}
```

- In the above example, the Person class is called the base class or the parent class, and the Employee class is called the derived class or the child class.

- ➢ The Employee class inherits from the Person class and so it automatically acquires all the public members of the Person class.
- ➢ It means even if the Employee class does not include FirstName, LastName properties and GetFullName() method, an object of the Employee class will have all the properties and methods of the Person class along with its own members.

Example: Inherited Members



```
Employee emp = new Employee();
emp.FirstName = "Steve";
emp.LastName = "Jobs";
emp.EmployeeId = 1;
emp.CompanyName = "Apple";

var fullname = emp.GetFullName(); //Steve Jobs
```

Program

```
using System;
public class Program
{
public static void Main()
        {
                Employee emp = new Employee();
                emp.FirstName = "Steve";
                emp.LastName = "Jobs";
                emp.EmployeeId = 1;
                emp.CompanyName = "Apple";

                Console.WriteLine(emp.GetFullName()); //base class method
                Console.WriteLine(emp.EmployeeId);
                Console.WriteLine(emp.CompanyName);

        }
}

class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }

   public string GetFullName(){
      return FirstName + " " + LastName;
   }
}

class Employee : Person
{
   public int EmployeeId { get; set; }
   public string CompanyName { get; set; }

}
```

3.9 Polymorphism

- Polymorphism is a Greek word that means multiple forms or shapes. You can use polymorphism if you want to have multiple forms of one or more methods of a class with the same name.
- In C#, polymorphism can be achieved in two ways:

    1. Compile-time Polymorphism
    2. Run-time Polymorphism

1. Compile-time Polymorphism (Method Overloading):

- Compile-time polymorphism is also known as method overloading.
- C# allows us to define more than one method with the same name but with different signatures. This is called method overloading.
- Method overloading is also known as early binding or static binding because which method to call is decided at compile time, early than the runtime.

Rules for Method Overloading:

1. Method names should be the same but method signatures must be different. Either the number of parameters, type of parameters, or order of parameters must be different.
2. The return type of the methods does not play any role in the method overloading.
3. Optional Parameters take precedence over implicit type conversion when deciding which method definition to bind.

The following example demonstrates the method overloading by defining multiple Print() methods with a different number of parameters of the same type.

```
using System;

public class Program
{
        public static void Main()
        {
                ConsolePrinter cp = new ConsolePrinter();
                cp.Print("Hello World!");
                cp.Print(1, "John");

        }
}

class ConsolePrinter
{
   public void Print(string str){
      Console.WriteLine(str);
   }
   public void Print(string str1, string str2){
      Console.WriteLine($"{str1}, {str2}");
   }
   public void Print(string str1, string str2, string str3){
      Console.WriteLine($"{str1}, {str2}, {str3}");
   }
   public void Print(int a){
      Console.WriteLine($"Integer {a}");
   }
         public void Print(int a, string str){
      Console.WriteLine($"{a}, {str}");
   }
   public void Print(string str, int a){
      Console.WriteLine($"{a}, {str}");
   }
}
```

2. Runtime Polymorphism: Method Overriding

- Run-time polymorphism is also known as inheritance-based polymorphism or method overriding.
- Inheritance allows you to inherit a base class into a derived class and all the public members of the base class automatically become members of the derived class.
- However, you can redefine the base class's member in the derived class to provide a different implementation than the base class. This is called method overriding that also known as runtime polymorphism.
- In C#, by default, all the members of a class are sealed and cannot be redefined in the derived class.
- Use the virtual keyword with a member of the base class to make it overridable, and use the override keyword in the derived class to indicate that this member of the base class is being redefined in the derived class

```
class Person
{
    public virtual void Greet()
    {
        Console.WriteLine("Hi! I am a person.");
    }
}

class Employee : Person
{
    public override void Greet()
    {
        Console.WriteLine("Hello! I am an employee.");
    }
}
```

- As you can see, Greet() method in the Person class is defined with the virtual keyword. It means this method can be overridden in the derived class using the override keyword.
- In the Employee class, the Greet() method is redefined with the override keyword. Thus, the derived class extends the base class's method.

3.10 Exception Handling in C#

- Exceptions in the application must be handled to prevent crashing of the program and unexpected result, log exceptions and continue with other functionalities.
- C# provides built-in support to handle the exception using try, catch & finally blocks.

```
try
{
    // put the code here that may raise exceptions
}
catch
{
    // handle exception here
}
finally
{
    // final cleanup code
}
```

- try block: Any suspected code that may raise exceptions should be put inside a try{ } block. During the execution, if an exception occurs, the flow of the control jumps to the first matching catch block.
- catch block: The catch block is an exception handler block where you can perform some action such as logging and auditing an exception. The catch block takes a parameter of an exception type using which you can get the details of an exception.
- finally block: The finally block will always be executed whether an exception raised or not. Usually, a finally block should be used to release resources, e.g., to close any stream or file objects that were opened in the try block.

The following may throw an exception if you enter a non-numeric character.

```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Enter a number: ");

      var num = int.Parse(Console.ReadLine());

      Console.WriteLine($"Squre of {num} is {num * num}");
   }
}
```

- To handle the possible exceptions in the above example, wrap the code inside a try block and handle the exception in the catch block, as shown below.

```
class Program
{
   static void Main(string[] args)
   {
      try
      {
         Console.WriteLine("Enter a number: ");

         var num = int.parse(Console.ReadLine());

         Console.WriteLine($"Squre of {num} is {num * num}");
      }
      catch
      {
         Console.Write("Error occurred.");
      }
      finally
      {
         Console.Write("Re-try with a different number.");
```

```
        }
    }
}
```

- In the above example, we wrapped this code inside a try block. If an exception occurs inside a try block, then the program will jump to the catch block. Inside a catch block, we display a message to instruct the user about his mistake, and in the finally block, we display a message about what to do after running a program.

- A try block must be followed by catch or finally or both blocks. The try block without a catch or finally block will give a compile-time error.

- Ideally, a catch block should include a parameter of a built-in or custom exception class to get an error detail. The following includes the Exception type parameter that catches all types of exceptions.

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Console.WriteLine("Enter a number: ");

            var num = int.parse(Console.ReadLine());

            Console.WriteLine($"Squre of {num} is {num * num}");
        }
        catch(Exception ex)
        {
            Console.Write("Error info:" + ex.Message);
        }
        finally
        {
            Console.Write("Re-try with a different number.");
        }
    }
}
```

3.10.1 Exception Filters

- You can use multiple catch blocks with the different exception type parameters. This is called exception filters.

- Exception filters are useful when you want to handle different types of exceptions in different ways.

```
class Program
{
   static void Main(string[] args)
   {
      Console.Write("Please enter a number to divide 100: ");

      try
      {
         int num = int.Parse(Console.ReadLine());

         int result = 100 / num;

         Console.WriteLine("100 / {0} = {1}", num, result);
      }
      catch(DivideByZeroException ex)
      {
         Console.Write("Cannot divide by zero. Please try again.");
      }
      catch(InvalidOperationException ex)
      {
         Console.Write("Invalid operation. Please try again.");
      }
      catch(FormatException  ex)
      {
         Console.Write("Not a valid format. Please try again.");
      }
      catch(Exception  ex)
      {
         Console.Write("Error occurred! Please try again.");
      }
   }

}
```

- In the above example, we have specified multiple catch blocks with different exception types.
- We can display an appropriate message to the user, depending upon the error, so the user does not repeat the same mistake again.

3.10.2 Invalid catch Block

- A parameter-less catch block and a catch block with the Exception parameter are not allowed in the same try-catch statements, because they both do the same thing.

```
try
{
   //code that may raise an exception
}
catch //cannot have both catch and catch(Exception ex)
{
   Console.WriteLine("Exception occurred");
}
catch(Exception ex) //cannot have both catch and catch(Exception ex)
{
   Console.WriteLine("Exception occurred");
}
```

- Also, parameter-less catch block catch{ } or general catch block catch(Exception ex){ } must be the last block.
- The compiler will give an error if you have other catch blocks after a catch{ } or catch(Exception ex) block.

```
try
{
   //code that may raise an exception
}
catch
{
   // this catch block must be last block
}
catch (NullReferenceException nullEx)
{
   Console.WriteLine(nullEx.Message);
}
catch (InvalidCastException inEx)
{
   Console.WriteLine(inEx.Message);
}
```

3.10.3 finally Block

- The finally block is an optional block and should come after a try or catch block.
- The finally block will always be executed whether or not an exception occurred.
- The finally block generally used for cleaning-up code e.g., disposing of unmanaged objects.

```
   static void Main(string[] args)
{
    FileInfo file = null;

    try
    {
        Console.Write("Enter a file name to write: ");
        string fileName = Console.ReadLine();
        file = new FileInfo(fileName);
        file.AppendText("Hello World!")
    }
    catch(Exception ex)
    {
        Console.WriteLine("Error occurred: {0}", ex.Message );
    }
    finally
    {
        // clean up file object here;
        file = null;
    }
}
```

3.10.4 Nested try-catch

- C# allows nested try-catch blocks. When using nested try-catch blocks, an exception will be caught in the first matching catch block that follows the try block where an exception occurred.

```
static void Main(string[] args)
{
    var divider = 0;

    try
    {
        try
        {
            var result = 100/divider;
        }
        catch
        {
            Console.WriteLine("Inner catch");
        }
    }
    catch
    {
        Console.WriteLine("Outer catch");
    }
```

}

- An inner catch block will be executed in the above example because it is the first catch block that handles all exception types.

3.11 Access Modifiers in C#

- Access modifiers in C# are used to specify the scope of accessibility of a member of a class or type of the class itself.
- For example, a public class is accessible to everyone without any restrictions, while an internal class may be accessible to the assembly only.
- Access modifiers are an integral part of object-oriented programming. Access modifiers are used to implement encapsulation of OOP.

- Access modifiers allow you to define who does or who doesn't have access to certain features.

- In C# there are 6 different types of Access Modifiers.

Modifier	Description
public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified
protected	Access is limited to within the class definition and any class that inherits from the class
internal	Access is limited exclusively to classes defined within the current project assembly
protected internal	Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

3.12 Interfaces

- Like a class, Interface can have methods, properties, events, and indexers as its members. But interfaces will contain only the declaration of the members.

- The implementation of the interface's members will be given by class who implements the interface implicitly or explicitly.

  - ✓ Interfaces specify what a class must do and not how.
  - ✓ Interfaces can't have private members.
  - ✓ By default all the members of Interface are public and abstract.

- ✓ The interface will always defined with the help of keyword ‘interface‘.
- ✓ Interface cannot contain fields because they represent a particular implementation of data.
- ✓ Multiple inheritance is possible with the help of Interfaces but not with classes.

3.12.1 Advantage of Interface:

- It is used to achieve loose coupling.
- It is used to achieve total abstraction.
- To achieve component-based programming
- To achieve multiple inheritance and abstraction.
- Interfaces add a plug and play like architecture into applications.

3.12.2 Syntax for Interface Declaration:

```
interface  <interface_name >
{
   // declare Events
   // declare indexers
   // declare methods
   // declare properties
}
```

Syntax for Implementing Interface:

```
class class_name : interface_name
```

- ➢ To declare an interface, use interface keyword. It is used to provide total abstraction. That means all the members in the interface are declared with the empty body and are public and abstract by default.
- ➢ A class that implements interface must implement all the methods declared in the interface.

Example 1:

```
// C# program to demonstrate working of
// interface
using System;

// A simple interface
interface inter1
{
   // method having only declaration
   // not definition
   void display();
}

// A class that implements interface.
class testClass : inter1
{

   // providing the body part of function
   public void display()
   {
       Console.WriteLine("Sudo Placement GeeksforGeeks");
   }

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

       // Creating object
       testClass t = new testClass();

       // calling method
       t.display();
   }
}
```