**OOPS Concepts**

Object-oriented programming (OOP) is a programming language model in which programs are organized around data, or objects, rather than functions and logic.

Object oriented programming or OOPS concept in C# is a type of programming model that contains the collection of objects. Each Object contains data fields and data members

**Why OOPs is important?**

OOP provides a clear modular structure for programs. It is good for defining abstract data types. Implementation details are hidden from other modules and other modules has a clearly defined interface. It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones

**OOPS Concepts:**

· Class & Object

· Encapsulation

· Inheritance

· Polymorphism

· Abstraction

**Class and Objects**

Classes and Objects are interrelated.

The class in c# is a collection of various data members (fields, properties, etc.) and member functions.

The object in c# is an instance of a class to access the defined properties and methods.

To define a class:

**class SampleClass**

**{**

**}**

C# also provides a light version of classes called structures that are useful when you need to create large array of objects and do not want to consume too much memory for that.

**struct SampleStruct**

**{**

**}**

**public class Point**

**{**

**public int x, y;**

**public Point(int x, int y)**

**{**

**this.x = x;**

**this.y = y;**

**}**

**}**

Instances of classes are created using the **new** operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two Point objects and store references to those objects in two variables:

**Point p1 = new Point(0, 0);**

**Point p2 = new Point(10, 20);**

**Note:**

Class is just a blueprint, It doesn't occupy any space as long as variable of type class has not defined. Once the object/instance of type class is define, the class member will occupy some space in memory. And the size of instance is equal to the sum of the size of members define in class.

**Types of Classes:**

There are four different types of classes available in C#. They are as follows:

1. Static class

2. Abstract class

3. Partial class

4. Sealed class

**Refer for more details**

<https://www.c-sharpcorner.com/article/types-of-classes-of-in-c-sharp/>

**Encapsulation**

In c#, **Encapsulation** is a process of binding the [data members](https://www.tutlane.com/tutorial/csharp/csharp-variables-with-examples) and [member functions](https://www.tutlane.com/tutorial/csharp/csharp-methods-functions-with-examples) into a single unit. In c#, class is the real time example for encapsulation because it will combine a various type of [data members](https://www.tutlane.com/tutorial/csharp/csharp-variables-with-examples) and [member functions](https://www.tutlane.com/tutorial/csharp/csharp-methods-functions-with-examples) into a single unit.

Generally, in c# the encapsulation is used to prevent an alteration of code (data) accidentally from the outside of [functions](https://www.tutlane.com/tutorial/csharp/csharp-methods-functions-with-examples). In c#, by defining the class fields with [properties](https://www.tutlane.com/tutorial/csharp/csharp-properties-get-set) we can protect the data from accidental corruption.

If we define a class fields with [properties](https://www.tutlane.com/tutorial/csharp/csharp-properties-get-set), then the encapsulated [class](https://www.tutlane.com/tutorial/csharp/csharp-classes-and-objects-with-examples) won’t allow us to access the fields directly, instead we need to use getter and setter [functions](https://www.tutlane.com/tutorial/csharp/csharp-methods-functions-with-examples) to read or write a data based on our requirements.

Real Life Example: Tv

Following is an example of defining an **encapsulation** [class](https://www.tutlane.com/tutorial/csharp/csharp-classes-and-objects-with-examples) using [properties](https://www.tutlane.com/tutorial/csharp/csharp-properties-get-set) with **get** and **set** accessors.

**using System;**

**using System.Text;**

**namespace Tutlane**

**{**

**class User**

**{**

**private string location;**

**private string name;**

**public string Location**

**{**

**get**

**{**

**return location;**

**}**

**set**

**{**

**location = value;**

**}**

**}**

**public string Name**

**{**

**get**

**{**

**return name;**

**}**

**set**

**{**

**name = value;**

**}**

**}**

**}**

**class Program**

**{**

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

**{**

**User u = new User();**

**// set accessor will invoke**

**u.Name = "Suresh Dasari";**

**// set accessor will invoke**

**u.Location = "Hyderabad";**

**// get accessor will invoke**

**Console.WriteLine("Name: " + u.Name);**

**// get accessor will invoke**

**Console.WriteLine("Location: " + u.Location);**

**Console.WriteLine("\nPress Enter Key to Exit..");**

**Console.ReadLine();**

**}**

**}**

**}**

**Explanation:** In the above program the class **Tutlane** is encapsulated as the variables are declared as private. To access these private variables, we are using the Name and Location accessors which contains the get and set method to retrieve and set the values of private fields. Accessors are defined as public so that they can access in other class.

**Advantages of Encapsulation:**

**Data Hiding**: The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is stored values in the variables. He only knows that we are passing the values to accessors and variables are getting initialized to that value.

· **Increased Flexibility**: We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to only use Get Accessor in the code. If we wish to make the variables as write-only then we have to only use Set Accessor.

· **Reusability**: Encapsulation also improves the re-usability and easy to change with new requirements.

· **Testing code is easy**: Encapsulated code is easy to test for unit testing.

**Inheritance**

**What is Inheritance?**

Inheritance allows us to define a class in terms of another class. Which means in C# one class is allowed to inherit the features (Data Members / Property, Functions, Methods) of another class.

**Terminology:**

3 terminologies in Inheritance as follows:

1. Base Class: The class whose features are inherited is known as super class (or a parent class).

2. Sub Class: The class that inherits the other class is knows as subclass (or a derived class, extended class, or child class). The Subclass can add its own properties function and methods in addition to base class.

3. Reusability: Inheritance supports concept of “Reusability”.

**Types of Inheritance:**

Below is the different types of inheritance which is supported by the C# in different combinations.

* **Single Inheritance:** in single inheritance subclass inherit with the features of Base Class.

**Example:**

public class Accountcreditinfo //base class

{

public string Credit()

{

return "balance is credited";

}

}

public class debitinfo : Accountcreditinfo //derived class

{

public string debit()

{

Credit(); ////derived class method

return "balance is debited";

}

}

* **Multi Level Inheritance:** In multi-level inheritance, one class is derived from another derived class then this type of inheritance is called multilevel inheritance.

**Example:**

public class Person

{

public string persondet()

{

return "this is the person class";

}

}

public class Bird : Person

{

public string birddet()

{

persondet();

return "this is the birddet Class";

}

}

public class Animal : Bird

{

public string animaldet()

{

persondet();

birddet();

return "this is the Animal Class";

}

}

* **Hierarchical Inheritance:** In Hierarchical inheritance one class serves as a base class for more than one sub class.

**Example:**

class A //base class

{

public string msg()

{

return "this is A class Method";

}

}

class B : A

{

public string info()

{

msg();

return "this is B class Method";

}

}

class C : A

{

public string getinfo()

{

msg();

return "this is B class Method";

}

}

* **Multiple Inheritances using Interface:** C# does not support multiple inheritances of classes. To overcome this problem we can use interfaces.

**Example:**

public interface C1 // interface 1

{

string setText(string a);

}

public interface C2 // interface 2

{

int getAmount(int Amt);

}

public class ICar : C1,C2 //implementatin

{

public int getAmount(int Amt)

{

return 100;

}

public string setText(string a)

{

return “this is the car”;

}

}

Constructor

**What is constructor?**

A special method of the class that is automatically invoked when an instance of the class is created is called a constructor.

The main use of constructors is to initialize private fields of the class while creating an instance for the class.

When you have not created a constructor in the class, the compiler will automatically create a default constructor of the 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 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 one static constructor only.

**Types of Constructor:**

There is 5 types of constructor

1. Default Constructor

2. Parameterize constructor

3. Copy constructor

4. Static Constructor

5. Private Constructor

* **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 with different values. The default constructor initializes:
  + All numeric fields in the class to zero.
  + All string and object fields to null.

**Example**

using System;

namespace DefaultConstractor

{

class addition

{

int a, b;

public addition() //default contructor

{

a = 100;

b = 175;

}

public static void Main()

{

addition obj = new addition(); //an object is created , constructor is called

Console.WriteLine(obj.a);

Console.WriteLine(obj.b);

Console.Read();

}

}

}

* **Parameterized Constructor:** A constructor with at least one parameter is called a parameterized constructor. The advantage of a parameterized constructor is that you can initialize each instance of the class with a different value.

**Example**

using System;

Namespace Constructor

class paraconstrctor

{

public int a, b;

public paraconstrctor(int x, int y) // decalaring Paremetrized Constructor with ing x,y parameter

{

a = x;

b = y;

}

}

class MainClass

{

static void Main()

{

paraconstrctor v = new paraconstrctor(100, 175); // Creating object of Parameterized Constructor and ing values

Console.WriteLine("-----------parameterized constructor example by---------------");

Console.WriteLine("\t");

Console.WriteLine("value of a=" + v.a );

Console.WriteLine("value of b=" + v.b);

Console.Read();

}

}

}

## **Copy Constructor:** The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

**Example:**

using System;

namespace copyConstractor

{

class employee

{

private string name;

private int age;

public employee(employee emp) // declaring Copy constructor.

{

name = emp.name;

age = emp.age;

}

public employee(string name, int age) // Instance constructor.

{

this.name = name;

this.age = age;

}

public string Details // Get deatils of employee

{

get

{

return " The age of " + name +" is "+ age.ToString();

}

}

}

class empdetail

{

static void Main()

{

employee emp1 = new employee("Vithal", 23); // Create a new employee object.

employee emp2 = new employee(emp1); // here is emp1 details is copied to emp2.

Console.WriteLine(emp2.Details);

Console.ReadLine();

}

}

}

## **Static Constructor:** When a constructor is created using a static keyword, it will be invoked only once for all of instances of the class, and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

**Some key points of a static constructor are:**

* A static constructor does not take access modifiers or have parameters.
* A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
* A static constructor cannot be called directly.
* The user has no control on when the static constructor is executed in the program.
* A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

**Example:**

using System;

namespace StaticConstructor

{

public static class employee

{

static employee() // Static constructor

{

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();

}

}

}

## **Private Constructor:** When a constructor is created with a private specifier, 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:

* + One use of a private constructor is when we have only static members.
  + It provides an implementation of a singleton class pattern
  + 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.

**Example:**

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 ----------");

Console.WriteLine();

Counter.currentview = 500;

Counter.visitedCount();

Console.WriteLine("Now the view count is: {0}", Counter.currentview);

Console.ReadLine();

}

}

}

**Reference Link:**

<https://www.c-sharpcorner.com/UploadFile/0c1bb2/constructors-and-its-types-in-C-Sharp/>

**Destructor**

**What is destructor ?**

Destructors are used to destruct instances of classes. In C# you can never call them, the reason is one cannot destroy an object. So who has the control over the destructor (in C#)? It’s the .NET frameworks Garbage Collector (GC).

**Characteristics of Destructor**

* Destructors (~) cannot be defined in Structs.
* Destructors (~) are only used with classes.
* Destructor cannot be inherited or overloaded.
* Destructor does not take modifiers or have parameters.
* Destructor cannot be called. They are invoked automatically.
* An instance becomes eligible for destruction when it is no longer possible for any code to use the instance.
* The Programmer has no control over when destructor is called because this is determined by Garbage Collector.
* Destructor is called when program exits.
* Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction.
* Destructor implicitly calls Finalize on the base class of object.

**Example:·**

class First

{

~First()

{

System.Console.WriteLine("First's destructor is called");

}

}

class Second : First

{

~Second()

{

System.Console.WriteLine("Second's destructor is called");

}

}

class Third : Second

{

~Third()

{

System.Console.WriteLine("Third's destructor is called");

}

}

class TestDestructors

{

static void Main()

{

Third t = new Third();

GC.Collect();

Console.ReadLine();

}

}

**Garbage collector**

* Garbage collector checks for objects that are no longer being used by application, if it treated an object eligible for destruction, it calls the destruction and reclaims the memory used to store the object.
* GC keeps tracks of all the objects and ensures that each object gets destroyed once.
* GC ensures that objects, which are being referenced, are not destroyed.
* GC destroys the objects only when necessary. Some situations of necessity are memory is exhausted or user explicitly calls System.GC.Collect() method.

Reference Link:

<https://www.c-sharpcorner.com/UploadFile/72d20e/concept-of-destructor-in-C-Sharp/>

**Polymorphism**

Derived from two Greek words: "Poly" and "morph"

***Poly*** means many and ***morph*** means forms

In sort its ***one name many forms***

**Real life example:**

1 Human Being: Different relations/roles and responsibilities

2 Smartphone: It can be a call, a text message, a picture message, mail, etc. So, the goal is common that is communication, but their approach is different.

**Two Types:**

**compile time** (Static) polymorphism and **runtime** (Dynamic) polymorphism

Compile time (Static polymorphism) >> Method (Function) Overloading OR Operator Overloading

Run Time (Dynamic polymorphism) >> Method (Function) Overriding

1) **Method Overloading (Compile Time Polymorphism/ Static Polymorphism/ Early Binding)**

a. Same name with different signatures in same class.

b. Not valid if Return type different with same signature.

c. Ex: Sum function with different signatures in single class.

2) **Method Overriding (Run Time Polymorphism/** **Dynamic Polymorphism/ Late Binding)**

a. Implement with Inheritance (Base and Derived class scenario) with same function and signatures

b. Method hiding (New Keyword)

c. Virtual and overriding.

Virtual: Give permission to child class to override the function, it will hide itself to override from child class

**Operator** **overloading**

We can say it one type of polymorphism in which operator overloaded.

Eg.: + Operator (Adding two integers as well use for concat two strings)

**Message passing**

Exchanging data between multiple objects calls Message Passing

class A

{

public void Method(Object obj)

{

*// Method does somthing*

}

}

class B

{

Object obj1 = new Object();

A a = new A();

a.Method(obj1);

}

**Interface**

An interface is like a contract. In the human world, the contract between the two or more humans binds them to act as per the contract. In the same way, the interface includes the declaration of one or more functionalities. Entities that implement the interface must define functionalities declared in the interface.

**Characteristics**

- All the members by default are public members

- An interface and class can inherit one or more interfaces.

**Types**

**1.implicit**

-Class will implement interface method with “public” modifier.

**#region Implicait Interface**

**interface IPen**

**{**

**string Write(string text);**

**}**

**public class Program : IPen**

**{**

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

**{**

**Program program = new Program();**

**Console.WriteLine(program.Write("hi"));**

**Console.ReadLine();**

**}**

**public string Write(string text)**

**{**

**return text;**

**}**

**}**

**#endregion**

**2.explicit**

-Method implementation will contain interface name e.g.void ICreditCard.getCardInfo()in the class.

-It should not contain public modifier.

**#region Explicit Interface**

**interface ILogger**

**{**

**string WriteLog(string Message);**

**}**

**interface IErrorLogger**

**{**

**string WriteLog(string Message);**

**}**

**public class ExplicitProgram : ILogger, IErrorLogger**

**{**

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

**{**

**ILogger logger = new ExplicitProgram();**

**IErrorLogger errorLogger = new ExplicitProgram();**

**logger.WriteLog("ILogger");**

**errorLogger.WriteLog("IErrorLogger");**

**Console.ReadLine();**

**}**

**string ILogger.WriteLog(string Message)**

**{**

**return Message;**

**}**

**string IErrorLogger.WriteLog(string Message)**

**{**

**return Message;**

**}**

**}**

**#endregion**

**Access Modifier**

Access Modifiers are keywords that define the accessibility of a member, class.

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  (added in C# 7.2) | The type or member can only be accessed by code in the same class or struct, or in a derived class from the same assembly, but not from another assembly. |