Topics:

* **Function and Method (Parameters and Arguments)**
* **Advantages of method**
* **Types of methods**
* **Class and Object**
* **Member Variable and Member Methods**
* **Static and Instance Methods**
* **Properties**
* **Local variable and Global Variable**
* **Synchronous and Asynchronous (Async and await)** https://youtu.be/e6S79YqOTAc?si=kz4nKB4JrTaT\_roi
* Access Specifiers
* Types of Class
* Constructor
* This keyword
* Destructor
* Indexers
* **Delegates**  https://www.youtube.com/watch?v=sDwEByK9Tcs
* Anonymous Function https://www.youtube.com/watch?v=9FL\_W9mudyY&list=PLX07l0qxoHFLZftsVKyj3k9kfMca2uaPR&index=43
* Lambda Expression https://youtu.be/sY-dWXsdxeA?si=CMNEACS9GDWL7Efu
* Extension Method <https://www.youtube.com/watch?v=_Ln5gwBQJ5c> , https://youtu.be/YnOwWFoZeHY?si=eK3X81tbWY6RS6Nr
* Multithreading
* Exception Handling
* String & Stringbuilder And Difference
* Class and Structure
* OOPs
* Boxing and Unboxing https://www.youtube.com/watch?v=37MgjErWXeA&list=PLp\_RsiLZjwQRVqZhQIddhU6b6k7bk5Tqc&index=74
* Solid Principles
* Serialization
* Razor View and Razor View Engine
* LINQ
* Array, tuple, list, collection
* Lazy Loading
* Abstract class and interface
* difference between IQueryable and IEnumerable
* nullable values
* What is Dictionary

**Function or Method**

A methodis a **block of code defined within a class that performs a specific task**.

* Methods are **used to implement functionality** in OOPs approach.
* Function is **reusable can be called multiples time** within a program.
* Methods can **accept inputs in the form parameters** and optionally return a value.

**Parameters:** A parameter is a variable **defined in the function’s declaration**.

**Arguments:** An argument is the **actual value passed to a method** when calling it.

There are two types of Arguments:

* **Positional Arguments:** Positional parameters **passed in the same order** **as defined in** **the function signature**.
* **Named Arguments:** Named parameters can be **passed in any using parameters names**.

**Advantages of Methods:**

* Programs follow OOPs concept (like encapsulation by grouping related logic).
* Methods improve code readability.
* Methods can be called multiple times within a program.
* Method can accept inputs in the form of parameters and optionally return a value.

**Types of Methods:**

#### 1. Based on Purpose:

* **Built-in Methods**: These are predefined methods **provided by the framework** or programming language.

Console.WriteLine();

* **User-defined Methods**: These are **created by the programmer** to perform specific tasks.

public int Add(int a, int b)

{

return a + b;

}

**2. Based on Accessibility:**

* **Static Method:** It **belongs to class itself** and can be **called without creating an instance of the class**.

public static void ShowMessage()

{

Console.WriteLine("This is a static method.");

}

* **Non-static Method: Require an instance** of the class to be called.

public void ShowMessage()

{

Console.WriteLine("This is a non-static method.");

}

3. **Based on Parameters:**

* **Parameterized Methods:** Accept input values (parameters) to perform their task.

public void ShowMessage(string name)

{

Console.WriteLine(name);

}

* **Non-Parameterized Methods:** Do not take any input parameters.

public void ShowMessage()

{

Console.WriteLine("Hello, World!");

}

4. **Async Methods:** Support asynchronous programming for non-blocking operations.

public async Task FetchDataAsync()

{

await Task.Delay(1000);

Console.WriteLine("Data fetched");

}

**Example 1:**

internal class Program

{

public void Show1() //declaring a method, non-static instance method, non-parameterized method

{

Console.WriteLine("Hello Show1 method");

}

public static void Show2() //declaring a method, static method, non-parameterized method

{

Console.WriteLine("Hello Show2 method");

}

public void Add1(int num1, int num2) //declaring a method, non-static instance Method, parameterized method

{

int result = num1 + num2;

Console.WriteLine("result: " + result);

}

public static void Add2(int num1, int num2) //declaring a method, static method, parameterized method

{

int result = num1 + num2;

Console.WriteLine("result: " + result);

}

public void Name(string name) //declaring a method, non-static method, parameterized method

{

Console.Write("Hello Name method, ");

Console.WriteLine("Your name is : " + name);

}

public void Show\_Name\_Age(string name, int age) //declaring a method, non-static instance method, parameterized method

{

Console.Write("Hello Show\_Name\_Age method, ");

Console.Write("Your Name is : " + name + " , ");

Console.WriteLine("Your Age is : " + age);

}

public void Name1(string name = "Unknown") //declaring a method, non-static method, parameterized method, “Optional Parameter”

{

Console.Write("Hello Name1 method, ");

Console.WriteLine("Your name is : " + name);

}

public void Age2(int age = 18) //declaring a method, non-static method, parameterized method, “Optional Parameter”

{

Console.Write("Hello Age2 method, ");

Console.WriteLine("Your age is : " + age);

}

static void Main(string[] args)

{

Program program = new Program(); //program is an object of Program class.

//new keyword is used to allocate the memory.

//Program() is construtor

program.Show1(); //Calling Show1 method / invoke the method

//non-static method

//non-parameterized method

Program.Show2(); //Calling Show2 method / invoke the method

//Static method

//non-parameterized method

program.Add1(10, 20); //Calling Add1 method / invoke the method

//non-Static method

//parameterized method

//actual values are called arguments

Program.Add2(30, 40); //Calling Add2 method / invoke the method

//Static method

//parameterized method

//actual values are called arguments

program.Name("Sudarshan"); //Calling Name method / invoke the method

//non-Static method

//parameterized method

//actual values are called arguments

program.Name1(); //Calling Name1 method / invoke the method

//Optional Parameter

program.Age2(); //Calling Age2 method / invoke the method

//Optional Parameter

program.Show\_Name\_Age("Sudarshan", 27); //Calling Name1 method / invoke the method

Console.ReadLine();

}

**Example 2:**

class Student

{

int roll\_no; //instance non-static member variable, it is by default private

string firstname; //instance non-static member variable, it is by default private

public int roll\_no; //instance non-static member variable, it is public variable

public string first\_name; //instance non-static member variable, it is public variable

public string last\_name; //instance/non-static member variable, it is public variable

public static int fees= 4000; //static public member variable

public static string SchoolName = "XY School"; //static public member variable

public void Show\_Full\_Name() //non-static instance method

{ //in instance method, we can use both static and non-static members variables

string fullname = this.first\_name + " " + this.last\_name;

Console.WriteLine("student name is : " + fullname);

}

public static void get\_fees() //static method

{ //in static method, we can use only static members variables

Console.WriteLine("School Fess: " + fees);

}

}

static void Main(string[] args)

{

Student student = new Student();

student.roll\_no = 23; //calling non-static member variable using object

student.first\_name = "Bruce"; //calling non-static member variable using object

student.last\_name = "Wayne"; //calling non-static member variable using object

Console.WriteLine("School Name: " + Student.SchoolName); //calling static member variable

student.Show\_Full\_Name(); //calling non-static instance method

Student.get\_fees(); //calling static method

Console.ReadLine();

}

**Class and Object**

C# is a **modern fully Server Side Object Oriented Programming Language**. So basically OOPs is a **programming paradigm based on classes and objects** to create applications.

**Class:** Class is a blueprint for creating objects. We can say class is a **collection of properties, member variables and member methods** used to **represent logical entities** of the program.

* Class is a **user defined data type and also reference type** and acts as a **template for creating objects.**
* Class contains **(variables called 🡪 member variables/fields) and (methods called 🡪member methods)**

**Object:** Object is an **instance of Class** and **created at runtime**. The Object has state and behavior.

State 🡪 Data Console.WriteLine(“Hello World”);

Behavior🡪functionality Hello World

**Program 1:** **Only with one class**

Class Student //Class is a keyword, //Student is a class name

{

int num1 = 20; //all are class member variables //by default member variables are private.

int num2;

string firstname = "Sudarshan";

string lastname;

char ch1 = 'A';

char ch2;

private int Student\_Id; //private member variable

private string \_firstname; //private member variable

private string \_email; //private member variable

public string Fullname; //public member variable

public int Id { get; set; } //properties

public string first\_name { get; set; } //properties

public void Show() //declaring a non-static instance method

{

Console.WriteLine("Hello Show method");

}

static void Main(string[] args) //main method , entry point of program.

{ 

int num1; //num1 is a local variable.

Console.WriteLine(num1); //it will give an error, we have to initialize it.

Student student = new Student(); //instance of class //invoke the object //student is a local variable and object of Student class.

Console.WriteLine(student.num1); //output= 20

Console.WriteLine(student.num2); //output= 0

Console.WriteLine(student.firstname); //output= Sudarshan

Console.WriteLine(student.lastname); //output= null

Console.WriteLine(student.ch1); //output= A

Console.WriteLine(student.ch2); //output= null

Console.WriteLine(student.Student\_Id); //output= 0

Console.WriteLine(student.\_firstname); //output= null

Console.WriteLine(student.\_email); //output= null

Console.WriteLine(student.Fullname); //output= null

Console.WriteLine(student.first\_name); //output= null

student.Show(); //output= Hello Show method

Console.ReadLine();

}

**Members Variables and Member Methods**

Variables and methods **declared inside a class** are called class members. "Variables called 🡪 **Class Member Variables and Methods called🡪 Class Member Methods**"

Class Program //Class is a keyword, //Program is a class name

{

int num1 = 20; //member variables/fields //by default member variables are private.

string str1 = "Sudarshan"; //member variables/fields //by default member variables are private.

void Show() //without access modifiers are private member method

{

Console.WriteLine("Hello Show method");

}

}

**Static and Instance Members**

**Instance Method:** When we declare a method **without using of static keyword** then that method is known as **Non-static Method or Instance Method**. Instance methods are **invoked using objects** of the class.

* **Instance Members:** when we **declare variable and methods without static keyword, these are called Member variables and Member Methods**.

Instance members are accessed through an object of the class. Instance methods can **access both instance variables and static members**.

class Program

{

public int num1; // Instance public member variable

public void Show() // Instance public member method

{

Console.WriteLine("hello show method”);

}

}

static void Main(string[] args)

{

Program program= new Program(); // invoke the object,

program.num1= 20; // Calling instance public member variable

program.Show(); // Calling instance public member method

}

**Static Method:** When we declare a method **using of static keyword.**

* **Static Members:** Define the **static member variables and static member methods** using static keywords.
* we can initialize the static variables inside the class and also outside the class.
* Static methods contain only static variables. Accessed using the class name.

class Program

{

public static int num1= 10; // static public member variable

public static void Show() // Static public member method

{

Console.WriteLine("hello from static show method”);

}

}

static void Main(string[] args)

{

Console.WriteLine(Program.num1); // Calling static variable, accessed via class name

Program.Show(); // Calling static method, accessed via class name

}

**Properties**

Properties allow you to **control accessibility of variables** of class.

Types of properties: <https://youtu.be/UVC3UhdFqrY?si=Fp4lrjcY8nKvJLyJ> , https://www.youtube.com/watch?v=fmK8YzP-elQ&list=PLp\_RsiLZjwQRVqZhQIddhU6b6k7bk5Tqc&index=71

* Read write Properties: When you need more control or validation and custom logic in the getter or setter.
* Read Only Properties
* Write Only Properties
* Auto Implemented Properties: Simple properties where you just need to store and retrieve a value without any custom logic in the getter or setter.

class Student

{

int num1; //by default private member variable

string name; //by default private member variable

private int \_stdid; //private member variable

private string \_name; //private member variable

private int \_totalmarks = 100; //private member variable, here value is already set

public string \_firstname { get; set; } //property, //Auto implemented property

public string \_lastname { get; set; } //property, //Auto implemented property

public int StdId //read write property

{

set {

if (value!= null)

this.\_stdid = value;

else

throw new ArgumentException("Name cannot be null.");

}

get { return this.\_stdid}

}

public string Name //write only properties

{ //set only value

set {this.\_name = value; } //No getter, so cannot read the value

}

public int Totalmarks //read only property

{ //we can only get the value,

get { return this.\_totalmarks; } //No setter, so cannot be modified

}

class Program

{

static void Main(string[] args)

{

Student student = new Student();

student.StdId = 24; //set the value //read write property

Console.WriteLine(student.StdId); //get the value

student.Name = "Sudarshan"; //we can only set the value //write only property

Console.WriteLine(student.Totalmarks); //we can only get the value //Read only property

student.\_firstname = "sudarshan"; //Auto implemented property

student.\_lastname= "sharma"; //Auto implemented property //auto-implemented property

Console.WriteLine($"Full Name: {student.FirstName} {student.LastName}");

Console.ReadLine();

}

}

**Local Variables and Global Variables**

**Local Variable:** Local Variable **declared in the method** accessed only within the function or block and **cannot be accessed outside the method**.

**Global Variable:** Global Variable **declared outside the method at the class level** and can be accessed globally.

https://www.youtube.com/watch?v=rIxC-vHEFOE

class Student

{

int id1 = 10; //private instance variable

public int id2 = 20; //public instance variable

private int id3 = 30; //private instance variable //Global variable

protected int id4 = 40; //protected instance variable

static int id5 = 50; //static variable

public static int id6 = 60; //public static variable

public void show() //instance method, non-static method

{

int x; //Local variable //they do not have access modifiers because they are scoped to the method.

string name; //Local variable

Console.WriteLine(id1);

Console.WriteLine(id2);

Console.WriteLine(id3);

Console.WriteLine(id4);

Console.WriteLine(x);

Console.WriteLine(name);

}

}

internal class Program

{

static void Main(string[] args)

{

Student student = new Student();

student.show();

}

**Synchronous and Asynchronous Method**

**Synchronous programming:** Synchronous programming is the programming model where **operations take place sequentially.**

**Asynchronous programming:** Asynchronous programming is the programming model where **operations do not depend on other operations and execute independently of each other.**

**Async and Await:** Async and await are the keywords **used to create a program asynchronously.**

public static void Task1()

{

Console.WriteLine("Task 1 Started");

Thread.Sleep(5000);

Console.WriteLine("Task 1 Completed ");

}

public static void Task2()

{

Console.WriteLine("Task 2 Started");

Thread.Sleep(2000);

Console.WriteLine("Task 2 Completed ");

}

public static async void Task5()

{

await Task.Run(() =>

{

Console.WriteLine("Task 1 Started");

Thread.Sleep(5000);

Console.WriteLine("Task 1 Completed ");

});

}

public static async void Task6()

{

await Task.Run(() =>

{

Console.WriteLine("Task 6 Started");

Thread.Sleep(2000);

Console.WriteLine("Task 6 Completed ");

});

}

static void Main(string[] args)

{

Task1();

Task2();

Task5();

Task6();

Console.ReadLine();

}

**Access Specifiers/Modifiers**

Access modifiers or specifiers are the **reserved keywords** that are **used to define accessibility of (Class, Interface, Structs, Delegates and Enum) as well as the scope of their member variables and member methods**.

<https://www.youtube.com/watch?v=iWsDnm1vWro&list=PLX07l0qxoHFLZftsVKyj3k9kfMca2uaPR&index=37>

https://youtu.be/Xp5rlWTfyXM?si=bMruV3Bx138VKiZe

1. **Public:** The Public Access Specifier provides the most permissive access level. The public members can be accessed anywhere in the class as well as in another class.
2. **Private:** The Private Access Specifier provides the least permissive access level. The Private members can be accessible only within the class in which they are declared.
3. **Protected:** The Protected Access Specifier allows the class members to be accessible within the class as well as derived class.
4. **Internal:** The Internal Access Specifier allows the class members to be accessible only within the class of the same namespaces/assembly.

**Note: Access Specifiers will not use with class.**

**by default, member variable and member method is private.**

**There is no need to use specifier in class, which has main method.**

**Note: Access Specifiers only use with “member variables” and “member methods”**

**Always prefer “private” and “protected” specifier.**

**Avoid to use “Public”, because it has no privacy.**

**Program 2: Program of Access Specifier:**

**More than two class**

Class Demo //here it will not run in main method, due to protection level

//it is outer class.

//we can’t use Access Specifiers on class.

{

int num1 = 38; //it is member variable.

//by default it is private.

//it will not run because it is private member variable.

public int num2 = 80; //it is member variable.

//if we use public specifier, now it is public,

//it will be accessed anywhere.

private int num3 = 50; //it is member variable.

//if we use private specifier it will not run due to protection level.

//it will work only in own class

//it can’t be inherit.

protected int num4 = 68; //it is member variable.

//if we use protected specifier, it can be access,

//we have to inherit this class.

internal int num5 = 45; //it is member variable.

//if we use internal specifier,it can be access,

void show1() //member method.

//by default it is private.

//show1 is a signature of method.

{

Console.WriteLine("this is by default private method, show1:");

}

public void show2() //member method

//it will run.

//here there is no protection level.

{

Console.WriteLine("this is public method:");

}

private void show3() //member method

//declaring a method

//if we use private specifier it will not run due to protection.

//it will work only in own class

//it can’t be inherit.

{

Console.WriteLine("this is private method:");

}

protected void Show4() //member method

//declaring a method

//if we use protected specifier, it can be access,

//we have to inherit this class.

{

Console.WriteLine("this is protected method:");

}

internal void Show5() //internal specifier

//member method

//it will run.

{

Console.WriteLine("this is internal method:");

}

}

internal class Program : Demo

{

static void Main(string[] args) //main method , entry point of program

{

Program program = new Program(); //declaring instance of class

//invoke the object

//program is a local variable and object of Program class.

Demo demo = new Demo(); //declaring instance of class

//invoke the object

//demo is a local variable and object of Program class.

Console.WriteLine(demo.num1); //it will not run , by default it is private Console.WriteLine(demo.num2); //it will run , public specifier.

Console.WriteLine(demo.num3); //it will not run , private specifier.

Console.WriteLine(program.num4); //it will run , protected specifier. but inherit.

Console.WriteLine(demo.num5); //it will run , it is internal

demo.Show1(); //it will not run , by default it is private

demo.Show2(); //it will run , public specifier.

demo.Show3(); //it will not run, private specifier.

program.Show4(); //it will run , protected specifier. but inherit

demo.Show5(); //it will run, internal specifier.

Console.ReadLine();

}

**Type of Class:**

Partial, Static, Abstract, Sealed

Program: CLASS\_Types

**1: Partial Class:**

Partial class is a special feature of C#. A partial class created using ‘**partial’** keyword.

If there is more than one partial class, we can work with one class name. We can call from one object.

Example: Multiple users can work in one project

Use: We use partial class when a class contains so many code of lines.

partial class FirstApp

{

public void Add()

{

Console.WriteLine("ADD function");

}

}

partial class FirstApp // here replace SecondApp to FirstApp

{

public void Sub()

{

Console.WriteLine("Sub function");

}

}

partial class FirstApp // here replace ThirdApp to FirstApp

{

public void Mul()

{

Console.WriteLine("Mul function");

}

}

internal class Program /or public partial class Program

{

static void Main(string[] args)

{

FirstApp firstApp = new FirstApp();

firstApp.Add();

firstApp.Sub();

firstApp.Mul();

Console.ReadLine();

}

}

**2: Static Class:**

Static class declared by ‘**Static’** keyword.

Static Class cannot be instantiated.

Static Class cannot be inherit.

Static Class consists of only **Static members and Static methods**.

Use: When we want fix the value.

static class Program

{

static string Address = "Noida"; //static member variable

static void show1() //static member method

{

Console.WriteLine("Welcome the C# Programming");

Console.WriteLine("Hello world");

}

static void Main(string[] args)

{

Console.WriteLine(Address);

Program.show1();

Console.ReadLine();

}

Or

internal class Program

{

static string Address = "Noida";

static void show1() //static method

{

Console.WriteLine("Welcome the C# Programming");

Console.WriteLine("Hello world");

}

static void Main(string[] args)

{

Console.WriteLine(Address);

Program.show1();

Console.ReadLine();

}

**3: Sealed Class**

A sealed class prevents inheritance.

Sealed class declared by **‘Sealed’** keyword.

Purpose: consider a class of Systeminformation that consists a critical method that affect the working of OS.

sealed class Baseclass //sealed class

{

public void Show1()

{

Console.WriteLine("This is protected base class show1 method... ");

}

}

class Derivedclass: Baseclass //Base class cannot be inherit because of Sealed class

{

public void Show2()

{

Console.WriteLine("This is protected derviedclass show3 method... ");

}

}

internal class Program

{

static void Main(string[] args)

{

Derivedclass derivedclass = new Derivedclass();

derivedclass.Show1();

Console.ReadLine();

}

}

**Sealing Method:**

Sealing method is used to prevent the method from further overriding.

* Sealed method is always used on override method of child class.
* We cannot override sealed method.
* We cannot make normal method as sealed.
* Sealed method cannot use with method hiding.

class A

{

public virtual void Show1()

{

Console.WriteLine("This is Class A ... ");

}

}

class B:A

{

public sealed override void Show1()

{

Console.WriteLine("This is Class B ... ");

}

}

class C :B

{

public override void Show1() //here we can't inherit because of sealed method Show1

{

Console.WriteLine("This is Class B ... ");

}

}

internal class Program

{

static void Main(string[] args)

{

B b= new B();

b.Show1();

Console.ReadLine();

}

}

10-May

Login

namespace Class\_Types

{

internal class Logic: Property

{

protected void Input()

{

Console.WriteLine("Enter the username : ");

Username = Console.ReadLine();

Console.WriteLine("Enter the password : ");

Password = Console.ReadLine();

}

public void Login()

{

Input();

if(Username=="Guest" && Password=="123")

{

Console.WriteLine("Welcome to dashboard {0}:", Username);

}

else

{

Console.WriteLine("Check the credantials");

}

}

}

}

Main class

namespace Class\_Types

{

internal class Program

{

static void Main(string[] args)

{

Logic logic = new Logic();

logic.Login();

Console.ReadLine();

}

}

}

Another class

namespace Class\_Types

{

internal class Property

{

protected string Username { get; set; }

protected string Password { get; set; }

}

}

**CONSTRUCTOR:**

Constructor is a method of a class which is responsible for initializing the variables of that class.

* Constructor must have the same name as the class name.
* Constructors can’t be abstract.
* Constructor is not a return type.
* Constructors can be multiple.
* Every variable has default value (All numeric values will be 0, string will be null and bool will be false) declared inside a class

**Defining and Calling Constructor**

**Defining:** Defining a constructor means implementing a constructor in your class.

Defining can be two types i.e. Implicit and Explicit. Implicit means the compiler will define the constructor. Explicit means we as a programmer define the constructor.

**Calling:** Whenever we are creating the instance, we are calling the constructor. Calling is Explicit. We should only call. There is no implicit call to the constructor.

Different Types of Constructor:

* Default Constructor
* Parameterized Constructor
* Static Constructor
* Copy Constructor
* Private Constructor

**1: Default Constructor**

* Without parameters is called the default constructor.
* All numeric values will be zero and string values will be null.
* Default Constructor is public.
* Default Constructor can be divided into two: **System Defined Default Constructor, User Defined Default Constructor**

**System Defined Implicit Default Constructor**: If we do not define any constructor explicitly in our program, then by default the **compiler will provide one constructor** at the time of compilation. That constructor is called a system defined default constructor.

public class Program

{

int age;

string name;

static void Main(string[] args)

{

Program program = new Program();

Console.WriteLine(program.age);

Console.WriteLine(program.name);

Console.ReadLine();

}

}

Output:

Id: 0

Address: null

**User Defined Default Constructor/ Explicit Default Constructor:** The **constructor which is defined by the user without any parameter** is called the user-defined default constructor.

public class Program

{

public int age;

public string name;

public Program()

{

Console.WriteLine(age = 18);

Console.WriteLine(name = "Hello");

}

static void Main(string[] args)

{

Program program = new Program();

Console.ReadLine();

}

}

Output:

10

Hello

**2: Parameterized Constructor:**

* A constructor that has **at least one parameter** is called a parameterized constructor.
* We can initialize the value.

Example 1:

internal class Program

{

public Program(int a, int b)

{

Console.WriteLine(a+b);

}

static void Main(string[] args)

{

Program program = new Program(10,20);

Console.ReadLine();

}

}

Output: 30

Example 2:

public class Program

{

public int age;

public string name;

public Program(int age, string name)

{

this.age = age;

this.name = name;

}

public void Display()

{

Console.WriteLine("Name: " + name + ", Age: " + age);

}

static void Main(string[] args)

{

Program program= new Program(25, "john");

program.Display();

Console.ReadLine();

}

}

Output: Name: John, Age: 25

**3: Static Constructor:**

* Static constructor is **used to initialize static fields.**
* Static constructor is created using a **static** keyword.
* It does not take access specifiers or parameters.
* It will be invoked only once for all the instances.
* Static constructor is invoked implicitly. It can't be called explicitly.before the default constructor.

Example 1:

public class Account

{

public int id;

public String name;

public static float rateOfInterest;

public Account(int id, String name)

{

this.id = id;

this.name = name;

}

static Account()

{

rateOfInterest = 9.5f;

}

public void display()

{

Console.WriteLine(id + " " + name + " " + rateOfInterest);

}

static void Main(string[] args)

{

Account account = new Account(100, "Vikas");

account.display();

Console.ReadLine();

}

}

Example 2::

internal class Program

{

static Program()

{

Console.WriteLine("Static Constructor...");

}

public Program()

{

Console.WriteLine("Default Constructor...");

}

static void Main(string[] args)

{

Program program = new Program();

Console.ReadLine();

}

}

Output:

Static Constructor...

Default Constructor...

4: **Copy Constructor:**

* A Constructor which creates an object by copying variables from another object is called a Copy Constructor.

5: **Private Constructor:**

* When a constructor create with private specifier.
* Private Constructor can’t be inherit.
* We can’t create instance and object.
* They are used in classes that contain static members only.

class Demo

{

private Demo()

{ }

}

internal class Program: Demo //we cannot inherit private constructor

{

static void Main(string[] args)

{

Program program = new Program();

Demo demo = new Demo(); //we cannot create instance of private constructor

Console.ReadLine();

}

}

**Constructor Overloading:**

* Having more than one constructor with different parameters called Constructor Overloading.

internal class Program

{

public Program()

{

Console.WriteLine("This is default constructor: ");

}

public Program(int a)

{

Console.WriteLine("This is constructor with one parameter:{0} " , a);

}

public Program(int a , int b)

{

Console.WriteLine("This is constructor with two parameter: {0}", (a+b));

}

public Program(string a , string b)

{

Console.WriteLine("This is constructor with two parameter: {0}", (a+b));

}

static void Main(string[] args)

{

Program program = new Program(12, 2.3344);

Console.ReadLine();

}

This Keyword:

* This keyword is used to refer the current instance of the class.
* This keyword is also used to differentiate between construtor parameter and class field if they have same name.
* This is a reference variable.

Public class show()

{

int x;

int y;

public void input(int x, int y)

{

this.x=x;

this.y=y;

}

public void Output()

{

Console.Writline(x);

Console.Writline(y);

}

}

Public static void main

{

Show show =new Show();

show.Input(2,3);

show.Output();

}

**DESTRUCTOR:**

* A destructor is a special method which has same name as class name start with tilde sign(~) it is used to de-allocates memory of objects .
* It is invoked automatically in the end.
* Destructor cannot be overloaded and inherited.
* We cannot use specifier and cannot take parameters.

internal class Program

{

public void Add()

{

Console.WriteLine("Hello..");

}

~Program()

{

Console.WriteLine("Destructor has been invoked");

}

static void Main(string[] args)

{

Program program = new Program();

program.Add();

Console.ReadLine();

}

}

Output:

Hello

Destructor has been invoked

**Indexers:**

* Indexers allow to object to be used as just like array.
* Indexers are special types of properties that adds logic that how can array store the values.
* Syntax of indexer resembles to properties.
* We can be use Access Modifiers.
* Indexers create with ‘this’ keyword.
* More than one indexer can be used with different parameters.
* Use: Indexers **concept is object acts as an array.**

class Employee

{

private int[] Age = new int[3];

public int this[int index] ///short key indexer

{

set

{

Age[index] = value;

}

get

{

return Age[index];

}

}

}

internal class Program

{

static void Main(string[] args)

{

Employee employee = new Employee(); //object will work as a indexer

employee[0] = 5;

Console.WriteLine(employee[0]);

Console.ReadLine();

}

}

Or

class Employee

{

private int[] Age = new int[3];

public int this[int index]

{

set

{

if (index >= 0 && index < Age.Length)

{

if (value > 0)

{

Age[index] = value;

}

else

{

Console.WriteLine("value is invalid...");

}

}

else

{

Console.WriteLine("Invalid index...");

}

}

get

{

return Age[index];

}

}

}

internal class Program

{

static void Main(string[] args)

{

Employee employee = new Employee();

employee[1] = 5;

Console.WriteLine(employee[0]);

Console.ReadLine();

}

}

**Delegates**

A delegate is a type which holds **reference to a method in a delegate’s object**.

* It is also called **function** **Pointer**.
* Invoke method use with delegate method.
* Delegate is a reference type.
* Delegate signatures should be the same as method signatures.
* Delegates can be parameterized or non-parameterized.
* Delegates do not have a body.
* Delegates are used to encapsulate methods.

There are 3 types of Delegate:

* Single cast Delegate: Single cast delegate points to single method at a time.
* Multicast cast Delegate: Multicast delegate points to multiple methods at a time.
* Multiple Delegate: multiple Delegates invoke multiple delegates with a program.

public delegate void Calculation(int a, int b); //out of class

public static void Add(int a, int b)

{

int res=a+b;

Console.WriteLine(res);

}

public static void Sub(int a, int b)

{

int res = a - b;

Console.WriteLine(res);

}

public static void Mul(int a, int b)

{

int res = a \* b;

Console.WriteLine(res);

}

public static void Div(int a, int b)

{

int res = a / b;

Console.WriteLine(res);

}

static void Main(string[] args)

{

Calculation obj1 = new Calculation(Program.Add);

obj1.Invoke(2, 4);

Calculation obj2 = new Calculation(Program.Sub);

obj2.Invoke(4, 2);

Calculation obj3 = new Calculation(Program.Mul);

obj3.Invoke(2, 4);

Calculation obj4 = new Calculation(Program.Div);

obj4.Invoke(8,4);

Console.ReadLine();

}

**Anonymous Function**

An anonymous method is a **method without a name**, only just a body. An anonymous function can be defined using a delegate **keyword**.

We can’t use jump statements.

public delegate void Mydelegate(int num);

static void Main(string[] args)

{

Mydelegate obj = delegate (int a)

{

a += 10;

Console.WriteLine(a);

};

obj.Invoke(5);

Console.ReadLine();

}

Lambda Function:

Lambda Expression is used to simplifies the anonymous function.

It is like shorthand function for anonymous function.

It is also works like as anonymous function.

Introduced in 3.0, (=>) Lambda operator.

There are two types of Lambda Function:

* Statement Lambda

{

a += 10;

Console.WriteLine(a);

};

* Expression Lambda

Mydelegate obj = a => a\*a;

Console.WriteLine(obj.Invoke(5));

public delegate void Mydelegate(int num);

static void Main(string[] args)

{

Mydelegate obj = a =>

{

a += 10;

Console.WriteLine(a);

};

obj.Invoke(5);

Console.ReadLine();

}

There are 2 types of Lambda function:

* Lambda Statement

Input => {statments};

* Expression Lambda
* Input => expression;

**Extension Methods:**

Extension methods are allow to inject additional methods without modifying, deriving or

recompiling the original class.

* Extension methods are defined as static methods but once they bind with another class,

then they convert into non-static or instance methods.

* If an Extension method is defined as same name or same signature of an existing method in the class, then Extension methods will not be called.
* Extension methods are static method.
* Introduced in version 3.0

Note: Inheritance is the process in which we can inherit the properties of another class.

We cannot apply inheritance on sealed class.

this Program p 🡪 binding parameter

static class Extensionmethod

{

public static void Func3(this Program p) //(this Program p) is a binding parameter

{

Console.WriteLine("This is third function...");

}

}

sealed class Program

{

public void Func1()

{

Console.WriteLine("This is first function...");

}

public void Func2()

{

Console.WriteLine("This is second function...");

}

static void Main(string[] args)

{

Program program = new Program();

program.Func1();

program.Func2();

program.Func3();

Console.ReadLine();

}

}

Or

With parameterize

static class Extensionmethod

{

public static void Func3(this Program p, int a) //(this Program p) is a binding parameter

{

Console.WriteLine("This is third function..." + a);

}

}

sealed class Program

{

public void Func1()

{

Console.WriteLine("This is first function...");

}

public void Func2()

{

Console.WriteLine("This is second function...");

}

static void Main(string[] args)

{

Program program = new Program();

program.Func1();

program.Func2();

program.Func3(10);

Console.ReadLine();

}

Multithreading:

A thread is defined as the execution path of a program. Each thread defines a unique flow of control. Threads are the executed by the OS using Time-Sharing.

Class: System.Threading.Thread

CurrentThread()

Thread.Sleep()

When c# program starts execution the main thread automatically created.

The threads are created using the Thread class called child threads of the main thread.

Multithreading is a process provided by the OS enables execution multiple application at the same time. It is a process to achieve multitasking.

public static void Func1()

{

for(int i=1;i<=40;i++)

{

Console.WriteLine("Function 1: "+ i);

}

}

public static void Func2()

{

for (int i = 1; i <= 40; i++)

{

Console.WriteLine("Function 2: " + i);

}

}

public static void Func3()

{

for (int i = 1; i <= 40; i++)

{

Console.WriteLine("Function 3: " + i);

}

}

static void Main(string[] args)

{

Thread t1 = new Thread(Func1);

t1.Start();

Thread t2 = new Thread(Func2);

t2.Start();

Thread t3 = new Thread(Func3);

t3.Start();

Console.ReadLine();

}

**Exception and Exception Handling**

An Exception is an error, which occurs during the execution of program that disrupts the normal flow of program.

All exceptions are derived from *System.Exception* class.

1: Program terminates or crash.

2: Error message will show

3: Statements after exception will not be execute.

Note: if error occurs , so C# default catch mechanism will handle, but we want handle from own side.

**Exception Handling:**

A process of handling runtime error is called **exception handling.**

|  |  |
| --- | --- |
| System.DivideByZeroException | handles the error generated by dividing a number with zero. |
| System.NullReferenceException | handles the error generated by referencing the null object. |
| System.InvalidCastException | handles the error generated by invalid typecasting. |
| System.IO.IOException | handles the Input Output errors. |
| System.FieldAccessException | handles the error generated by invalid private or protected field access. |

**Type of error:**

1: Compile time Error

2: Logical Error,

3: Runtime Error:

**Try function:** It can be use only one time.

It contains only logic.

**Catch function:** It can use multiple times.

It is used to show the errors.

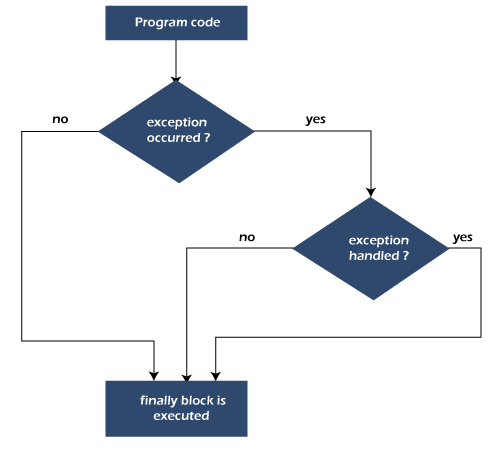
**Finally:** Finally block is always executed whether exception occur or not.

Finally block is important block that is used to closing the connection.

Finally block follows try or catch block.

Finally blocks can only be one time.

**Throw:** User defined exception method



**Example: 1**

using System;

namespace Exception\_Prog

{

internal class Program

{

public int num1 { get; set; }

public int num2 { get; set; }

public int res { get; set; }

void Input()

{

Console.WriteLine("Enter the first number");

num1=Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter the first number");

num2 = Convert.ToInt32(Console.ReadLine());

}

void divide()

{

Input();

res = num1 / num2;

Console.WriteLine("answer is :"+res);

}

static void Main(string[] args)

{

Program program = new Program();

program.divide();

Console.ReadLine();

}

}

}

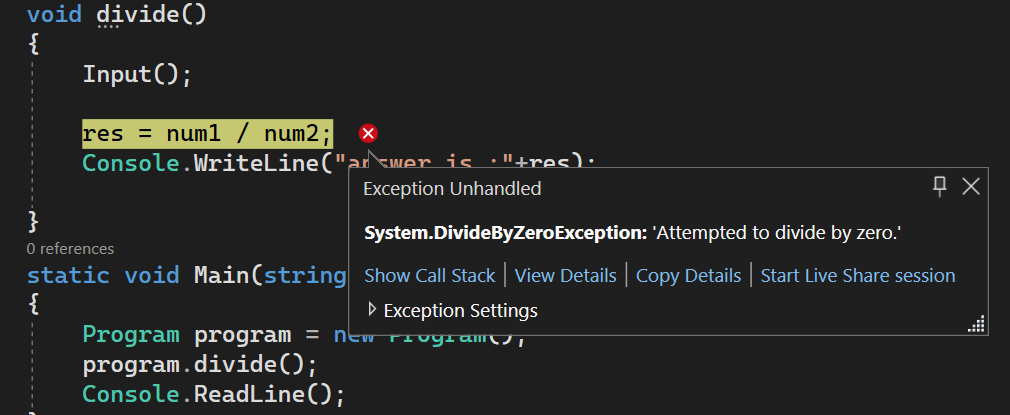
Output:

Enter the first number

10

Enter the first number

0



It is Runtime error.

**Example: 2**

using System;

namespace Exception\_Prog

{

internal class Program

{

public decimal num1 { get; set; }

public decimal num2 { get; set; }

public decimal res { get; set; }

void Input()

{

Console.WriteLine("Enter the first number");

num1=Convert.ToDecimal(Console.ReadLine());

Console.WriteLine("Enter the first number");

num2 = Convert.ToDecimal(Console.ReadLine());

}

void divide()

{

Input();

res = num1 / num2;

Console.WriteLine("answer is :"+res);

}

static void Main(string[] args)

{

Program program = new Program();

program.divide();

Console.ReadLine();

}

}

}

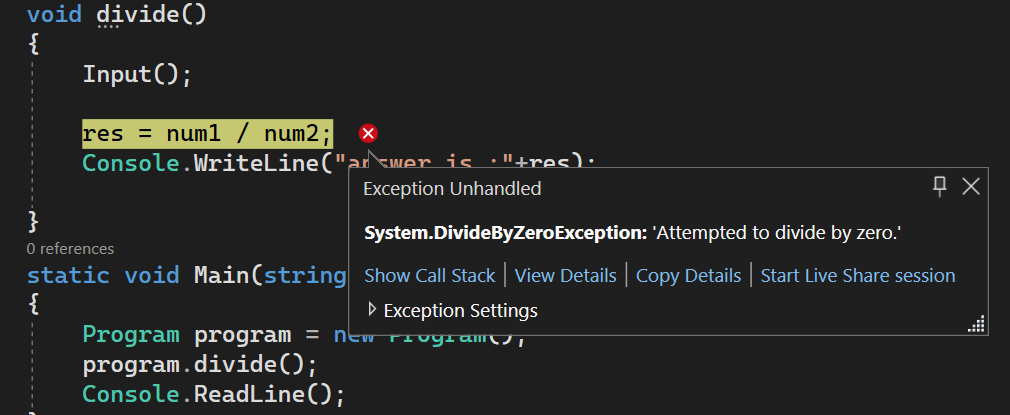
Output:

Enter the first number

10

Enter the first number

0



It is Runtime error.

**Only Try**

public decimal num1 { get; set; }

public decimal num2 { get; set; }

public decimal res { get; set; }

void Input()

{

try

{

Console.WriteLine("Enter the first number");

num1 = Convert.ToDecimal(Console.ReadLine());

Console.WriteLine("Enter the first number");

num2 = Convert.ToDecimal(Console.ReadLine());

}

catch

{

}

void divide()

{

try

{

Input();

res = num1 / num2;

Console.WriteLine("answer is :" + res);

}

catch

{

}

}

static void Main(string[] args)

{

Program program = new Program();

program.divide();

Console.ReadLine();

}

Output:

Enter the first number

10

Enter the first number

0

Output

Enter the first number

10

Enter the first number

S

Remarks: it will not break the program

**Try, Catch**

public decimal num1 { get; set; }

public decimal num2 { get; set; }

public decimal res { get; set; }

void Input()

{

try

{

Console.WriteLine("Enter the first number");

num1 = Convert.ToDecimal(Console.ReadLine());

Console.WriteLine("Enter the first number");

num2 = Convert.ToDecimal(Console.ReadLine());

}

catch(Exception excp) //excp is a object

{

Console.WriteLine(excp.Message);

Console.WriteLine("Check Credentials");

}

}

void divide()

{

try

{

Input();

res = num1 / num2;

Console.WriteLine("answer is :" + res);

}

catch (Exception excp) //exception is class, here we are calling class directly

{

Console.WriteLine(excp.Message);

Console.WriteLine("Check Credentials");

}

}

static void Main(string[] args)

{

Program program = new Program();

program.divide();

Console.WriteLine("Hello World");

Console.ReadLine();

}

Output: exception message will show if inputs are wrong.

**Try, Catch, Finally**

internal class Program

{

public decimal num1 { get; set; }

public decimal num2 { get; set; }

public decimal res { get; set; }

void Input()

{

try

{

Console.WriteLine("Enter the first number");

num1 = Convert.ToDecimal(Console.ReadLine());

Console.WriteLine("Enter the first number");

num2 = Convert.ToDecimal(Console.ReadLine());

}

catch(Exception excp) //exception is class

{

Console.WriteLine(excp.Message);

Console.WriteLine("Check Credentials");

}

}

void divide()

{

try

{

Input();

res = num1 / num2;

Console.WriteLine("answer is :" + res);

}

catch (Exception excp) //exception is class

{

Console.WriteLine(excp.Message);

Console.WriteLine("Check Credentials");

}

finally

{

Num1=0; /string.empty; //memory will clear/free

Console.WriteLine("Finish the program" + num1);

}

}

static void Main(string[] args)

{

Program program = new Program();

program.divide();

Console.ReadLine();

}

Output: exception message will show if inputs are wrong.

And also will run finally (for close the program)

**Try, Catch, Finally, Throw**

void throw\_exception()

{

Console.WriteLine("Enter your Age:");

int age = Convert.ToInt16(Console.ReadLine());

try

{

if (age >= 18)

{

Console.WriteLine("You are eligible to vote: ");

}

else

{

throw new Exception("you are not eligible to vote :");

}

}

catch(Exception ex)

{

Console.WriteLine(ex.Message);

}

finally

{

age = 0; //memory will clear/free

Console.WriteLine("Finish the program and memery is : {0}", age);

}

}

static void Main(string[] args)

{

Program program = new Program();

program.throw\_exception();

Console.ReadLine();

}

**Class and Structure:**

* Class is a collection of methods. Structure is a collection of variable of different data types.
* Class is user-defined data type. Structure is also user-defined data type.
* Class’s objects are created on the **heap** memory. Structure’s objects are created on the **stack** memory.
* Class is a reference type. Structure is value type.
* Class can be **inherit**. Structure can’t be **inherit**.
* Class has all members **private** by default. Structure has all members **public** by default.