<https://dotnettutorials.net/course/csharp-dot-net-tutorials/>

<https://www.javatpoint.com/c-sharp-collections>

<https://www.geeksforgeeks.org/csharp-programming-language/>

implementation :- <https://www.freecodecamp.org/learn/foundational-c-sharp-with-microsoft/>

**C# Language Features**

To become a.NET developer, it’s important to have a strong knowledge of C# language features, including:

1. Basic Syntax,Variables, Data Types, and Operators
2. Conditional Statements
3. Arrays
4. Strings
5. Object-Oriented Programming
6. Exception Handling
7. Events, Delegates, and Lambda Expression
8. Generics
9. Multi-Threading
10. Asynchronous Programming
11. Collections in C#
12. File Handling
13. Parallel Programming
14. AutoMapper
15. ADO.NET
16. LINQ (Language Integrated Query)

C# is an elegant and type-safe object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework. You can use C# to create Windows client applications, XML Web services, distributed components, client-server applications, database applications, and much, much more.

C# syntax is highly expressive, yet it is also simple and easy to learn. The curly-brace syntax of C# will be instantly recognizable to anyone familiar with C, C++ or Java. Developers who know any of these languages are typically able to begin to work productively in C# within a very short time.

**C# was developed by Microsoft as part of its . NET initiativeto address certain limitations and complexities in C++.** C# was designed to be simpler, more modern, and more productive for developers, with features such as automatic garbage collection, simplified type declarations, and a unified type system.

# **1] Basic syntax, Variables, Data Types, and Operators**

## C# Syntax

CCCCCCCCCCCCCCCCCCC

#include <stdio.h> **header file library**  
  
int main() {  
  printf("Hello World!"); int var; scanf(“%d”, &var);  
  return 0;  
}

C+++++++++++++++++++

#include <iostream> **header file library**  
using namespace std;   
  
int main() {  
  cout << "Hello World!";  
  return 0;  
}

C####################

using System;

namespace HelloWorld

{

class Program

{

static void Main(string[] args) //Entry point

{

Console.WriteLine("Hello World!");

}

}

} //CaMel case used in c#

using System means that we can use classes from the System namespace.

namespace is used to organize your code, and it is a container for classes and other namespaces.

class is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.

“using namespace std” means we use the namespace named std. “std” is an abbreviation for standard. So that means we use all the things with in “std” namespace. If we don't want to use this line of code, we can use the things in this namespace like this. std::cout,

<https://www.programiz.com/csharp-programming/online-compiler/>

## C# Input & Output

To output values or print text in C#, you can use the WriteLine() method:

Console.WriteLine("Hello World!");

There is also a Write() method, which is similar to WriteLine().

The only difference is that it does not insert a new line at the end of the output:

The Console.ReadLine() method returns a string

For int Convert it.

Console.WriteLine("Enter a number:");

string numString = "123";

int num = Convert.ToInt32(numString);

More ways to convert int to string

int.Parse() int.TryParse()

## C# Variables & Data Types

* int - stores integers (whole numbers), without decimals, such as 123 or -123 4 bytes  
    
  and long is for large value to store. 8bytes
* double - stores floating point numbers, with decimals, such as 19.99 or -19.99 8bytes

float for shorter **Floating point types 4bytes**

* char
* string
* bool

If you don't want to overwrite existing values, you can add the const keyword in front of the variable type.

**const** int myNum = 15;

To combine both text and a variable, use the + character:

string name = "John";

Console.WriteLine("Hello " + name);

**Use the \" escape sequence**

\" -> “

\\ -> \

* like a tab \t, new line \n, or a double quotation mark \".
* Use an escape character for the backslash \\ when you need to use a backslash in all other scenarios.

String interpolation

**string message = greeting + " " + firstName + "!";**

**string message = $"{greeting} {firstName}!";**

**The @ symbol creates a verbatim string where it's unnecessary to escape the \.**

**The literal with the m or M suffix is of type decimal.**

**Otherwise result will be zero.**

**readonly int x; //Readonly Variable**

we can modify the read-only variable value only within a constructor.

**Read-Only Variables in C#**

When we declare a variable by using the readonly keyword, then it is known as a read-only variable and these variables can’t be modified like constants but after initialization. That means it is not mandatory to initialize a read-only variable at the time of its declaration, they can also be initialized under the constructor. That means we can modify the read-only variable value only within a constructor.

## C# Type Casting

Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

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

double myDouble = 9.78;

int myInt = (int) myDouble; // Manual casting: double to int

Console.WriteLine(myDouble); // Outputs 9.78

Console.WriteLine(myInt); // Outputs 9

## Type Conversion Methods

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

**Convert.ToString()** method handles null, while the **ToString()** method doesn’t handle the Null and throws an exception

Convert.ToString**(**Name**)**; right

Name.ToString**()**; wrong

Var keyword defines the data type statically i.e. not on run time.

## C# Operators

|  |  |  |  |
| --- | --- | --- | --- |
| % | Modulus | Returns the division remainder | x % y |

|  |  |  |  |
| --- | --- | --- | --- |
| && | Logical and | Returns True if both statements are true | x < 5 &&  x < 10 |
| || | Logical or | Returns True if one of the statements is true | x < 5 || x < 4 |
| ! | Logical not | Reverse the result, returns False if the result is true | !(x < 5 && x < 10) |

**Ternary operators**

*variable = (condition) ? expressionTrue : expressionFalse;*

int time = 20;

string result = (time < 18) ? "Good day." : "Good evening.";

Console.WriteLine(result);

# **2] Conditional Statements**

## C# Switch Statements

switch(expression)

{

case x:

// code block

break;

case y:

// code block

break;

default:

// code block

break;

}

int day = 4;

switch (day)

{

case 1:

Console.WriteLine("Monday");

break;

case 2:

Console.WriteLine("Tuesday");

break;

case 3:

Console.WriteLine("Wednesday");

break;

case 4:

Console.WriteLine("Thursday");

break;

default:

Console.WriteLine("IDK");

break;

}

// Outputs "Thursday" (day 4)

When C# reaches a break keyword, it breaks out of the switch block.

The default keyword is optional and specifies some code to run if there is no case match:

## The Do/While Loop

do

{

*// code block to be executed*

}

while (condition);

## C# For Loop

for (int i = 0; i < 5; i++)

{

Console.WriteLine(i);

}

## The foreach Loop

There is also a foreach loop, which is used exclusively to loop through elements in an **array**:

foreach (type variableName in arrayName)

{

*// code block to be executed*

}

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

foreach (string i in cars)

{

Console.WriteLine(i);

}

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

The break statement can also be used to jump out of a **loop**

# 3] Arrays

Arrays are mutable.

you should note that if you declare an array and initialize it later, you have to use the new keyword:

// Declare an array

string[] cars;

// Add values, using new

cars = new string[] {"Volvo", "BMW", "Ford"};

// Add values without using new (this will cause an error)

cars = {"Volvo", "BMW", "Ford"};

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Console.WriteLine(cars[0]);

// Outputs Volvo

We can edit arrays

// Create an array of four elements and add values right away

string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

**Array Length**

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Console.WriteLine(cars.Length); // Outputs 4

**1)Single Arrays**

Console.Write("Input {0} number of elements in the array :\n",n);

for(i=0;i<n;i++)

{

Console.Write("element - {0} : ",i);

a[i] = Convert.ToInt32(Console.ReadLine());

}

int[] num = new int[4]{1,2,3,4}

**2) Mutli- Array**

int[,] numbers = { {1, 4, 2}, {3, 6, 8} };

**Good to know:** The single comma [,] specifies that the array is two-dimensional. A three-dimensional array would have two commas: int[,,].

You can easily loop through the elements of a two-dimensional array with a foreach loop:

int[,] numbers = { {1, 4, 2}, {3, 6, 8} };

foreach (int i in numbers)

{

Console.WriteLine(i);

}

int[,] numbers = { {1, 4, 2}, {3, 6, 8} };

for (int i = 0; i < numbers.GetLength(0); i++)

{

for (int j = 0; j < numbers.GetLength(1); j++)

{

Console.WriteLine(numbers[i, j]);

}

}

**i = arr.Length & j = arr[i].Length**

**Types of Arrays**

There are four types of arrays in C#.

1. Single-dimensional arrays
2. Multi-dimensional arrays or rectangular arrays
3. Jagged arrays
4. Mixed arrays.

Initializing multi-dimensional arrays

int[,] numbers = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };

**int**[,] arr=**new** **int**[3,3]; //declaration of 2D array

**int**[,,] arr=**new** **int**[3,3,3]; //declaration of 3D array

**C# Jagged Arrays**

"array of arrays"

**int**[][] arr = **new** **int**[2][];

arr[0] = **new** **int**[4];

arr[1] = **new** **int**[6];

arr[0] = **new** **int**[4] { 11, 21, 56, 78 };

arr[1] = **new** **int**[6] { 42, 61, 37, 41, 59, 63 };

eg.

**int**[][] arr = **new** **int**[3][]{

**new** **int**[] { 11, 21, 56, 78 },

**new** **int**[] { 2, 5, 6, 7, 98, 5 },

**new** **int**[] { 2, 5 }

        };

Access same it as 2d arrays.

 // Traverse array elements

**for** (**int** i = 0; i < arr.Length; i++)

        {

**for** (**int** j = 0; j < arr[i].Length; j++)

            {

                System.Console.Write(arr[i][j]+" ");

            }

            System.Console.WriteLine();

        }

In C#, **params** is a keyword which is used to specify a parameter that takes variable number of arguments.

**public** **void** Show(**params** **int**[] val) // Params Paramater    
just pass this as argument  
and pass as many as arguments.

**C# Array class**

C# provides an Array class to deal with array related operations. It provides methods for creating, manipulating, searching, and sorting elements of an array. This class works as the base class for all arrays in the .NET programming environment.

 Array.Reverse(arr);

Array.Sort(arr);

 Array.Copy(arr, arr2, arr.Length);

# 4] Strings

String objects are immutable but StringBuilder is the mutable string type.

string txt = "Hello World";

Console.WriteLine(txt.ToUpper()); // Outputs "HELLO WORLD"

Console.WriteLine(txt.ToLower()); // Outputs "hello world"

Console.WriteLine(txt.Length); // Outputs 11

The + operator can be used between strings to combine them

You can also use the string.Concat() method to concatenate two strings

**Strings - Special Characters**

The backslash (\) escape character turns special characters into string characters:

|  |  |  |
| --- | --- | --- |
| **Escape character** | **Result** | **Description** |
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |

All other String Functions.

string Substring(int startIndex)

string Substring(int startIndex, int length)

string a = "bcfthgmc";

a.Substring(0,2) => bc

string authors = "Mahesh Chand, Henry He, Chris Love, Raj Beniwal, Praveen Kumar";

// Split authors separated by a comma followed by space

string[] authorsList = authors.Split(", ");

// Splitting the string into words based on spaces and storing them in an array

String line= “Hi guys”

string[] words = line.Split(new[] { " " }, StringSplitOptions.None);

// convert the letters of a string into alphabetical order

// Using LINQ's OrderBy method to sort characters in the string alphabetically

// Converting the sorted characters back to a string using new string() and ToArray()

return new string(str1.OrderBy(x => x).ToArray());

# 5] Object-Oriented Programming

## C# Classes and Objects

class Car

{

string color = "red";

static void Main(string[] args)

{

Car **myObj** = new Car();

Console.WriteLine(myObj.color);

}

}

* When calling a stateless method, you don't need to create a new instance of its class first.
* When calling a stateful method, you need to create an instance of the class, and access the method on the object.
* Use the new operator to create a new instance of a class.
* An instance of a class is called an *object*.

## C# Class Members

// The class

class MyClass

{

**// Class members**

string color = "red"; // **attributes**

int maxSpeed = 200; // **attributes**

public void fullThrottle() // **methods**

{

Console.WriteLine("The car is going as fast as it can!");

}

}

## C# Constructors & Destructor

A constructor is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

// Create a Car class

class Car

{

public string model; // Create a field

// Create a **class constructor** for the Car class

public Car()

{

model = "Mustang"; // Set the initial value for model

}

static void Main(string[] args)

{

Car Ford = new Car(); // Create an object of the Car Class (this will **call the constructor**)

Console.WriteLine(Ford.model); // Print the value of model

}

}

// Outputs "Mustang"

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void or int).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.

Constructors can also take parameters, which is used to initialize fields.

class Car

{

public string model;

// Create a class constructor with a parameter

public Car(string modelName)

{

model = modelName;

}

static void Main(string[] args)

{

Car Ford = new Car("Mustang");

Console.WriteLine(Ford.model);

}

}

// Outputs "Mustang"

**You can have as many parameters as you want:**

**A destructor** works opposite to constructor, It destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

**public** Employee()

        {

            Console.WriteLine("Constructor Invoked");

        }

        ~Employee()

        {

            Console.WriteLine("Destructor Invoked");

        }

## C# Access Modifiers

C# has the following access modifiers:

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| public | The code is accessible for all classes |
| private | The code is only accessible within the same class |
| protected | The code is accessible within the same class, or in a class that is inherited from that class. |
| internal | The code is only accessible within its own assembly, but not from another assembly. |

There's also two combinations: protected internal and private protected.

**Why Access Modifiers?**

To control the visibility of class members (the security level of each individual class and class member).

**C# this**

It can be used **to refer current class instance variable**. It is used if field names (instance variables) and parameter names are same, that is why both can be distinguish easily.

It can be used **to pass current object as a parameter to another method**.

It can be used **to declare indexers**.

## C# static class

In C#, static is a keyword or modifier In C#, static can be field, method, constructor, class, properties, operator and event.

It is used to refer the common property of all objects

whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

The C# static class is like the normal class but it cannot be instantiated. It can have only static members. **The advantage of static class is that it provides you guarantee that instance of static class cannot be created**

* C# static constructor cannot have any modifier or parameter.
* C# static constructor is invoked implicitly. It can't be called explicitly.

## OOPs

OOPs, provide 4 principles. They are

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**

**Abstraction**

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or **Interfaces**

The abstract keyword is used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

// Abstract class

abstract class Animal

{

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep()

{

Console.WriteLine("Zzz");

}

}

// Derived class (inherit from Animal)

class Pig : Animal

{

public override void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound(); // Call the abstract method

myPig.sleep(); // Call the regular method

}

}

The pig says: wee wee  
Zzz

The process of binding the data and functions together into a single unit (i.e. class) is called [**Encapsulation**](https://dotnettutorials.net/lesson/encapsulation-csharp/). In simple words, we can say that it is a process of defining a class by hiding its internal data members from outside the class and accessing those internal data members only through publicly exposed methods or propertie

To achieve "**Encapsulation**" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as private

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare fields/variables as private
* provide public get and set methods, through **properties**, to access and update the value of a private field

A property is like a combination of a variable and a method, and it has two methods: a get and a set method:

class Person

{

private string name; // field

public string Name // property

{

get { return name; }

set { name = value; }

}

}

class Program

{

static void Main(string[] args)

{

Person myObj = new Person();

myObj.Name = "Liam";

Console.WriteLine(myObj.Name);

}

}

**C# Inheritance**

class Vehicle // base class (parent)

{

public string brand = "Ford"; // Vehicle field

public void honk() // Vehicle method

{

Console.WriteLine("Tuut, tuut!");

}

}

class Car : Vehicle // derived class (child)

{

public string modelName = "Mustang"; // Car field

}

class Program

{

static void Main(string[] args)

{

// Create a myCar object

Car myCar = new Car();

// Call the honk() method (From the Vehicle class) on the myCar object

myCar.honk();

// Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class

Console.WriteLine(myCar.brand + " " + myCar.modelName);

}

}

Tuut, tuut!  
Ford Mustang

The process by which the members of one class are transferred to another class is called [**inheritance**](https://dotnettutorials.net/lesson/inheritance-c-sharp/).

**Polymorphism**

Inheritance lets us inherit fields and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

So, the word polymorphism means the ability to take more than one form.

we can say that when the same function/operator will show different behaviors by taking different types of values or with a different number of values called [**Polymorphism**](https://dotnettutorials.net/lesson/polymorphism-csharp/). There are two types of polymorphism

1. Static polymorphism/compile-time polymorphism/Early binding
2. Dynamic polymorphism/Run time polymorphism/Late binding

Static polymorphism is achieved by using **function overloading and operator overloading** whereas dynamic polymorphism is achieved by using **function overriding**.

class Animal // Base class (parent)

{

public void animalSound()

{

Console.WriteLine("The animal makes a sound");

}

}

class Pig : Animal // Derived class (child)

{

public void animalSound()

{

Console.WriteLine("The pig says: wee wee");

}

}

class Dog : Animal // Derived class (child)

{

public void animalSound()

{

Console.WriteLine("The dog says: bow wow");

}

}

class Program

{

static void Main(string[] args)

{

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

The output will be:

The animal makes a sound  
The animal makes a sound  
The animal makes a sound

That is because the base class method overrides the derived class method, when they share the same name.

However, C# provides an option to override the base class method, by adding the virtual keyword to the method inside the base class, and by using the override keyword for each derived class methods:

class Animal // Base class (parent)

{

public **virtual** void animalSound()

{

Console.WriteLine("The animal makes a sound");

}

}

class Pig : Animal // Derived class (child)

{

public **override** void animalSound()

{

Console.WriteLine("The pig says: wee wee");

}

}

class Dog : Animal // Derived class (child)

{

public **override** void animalSound()

{

Console.WriteLine("The dog says: bow wow");

}

}

class Program

{

static void Main(string[] args)

{

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

The output will be:

The animal makes a sound  
The pig says: wee wee  
The dog says: bow wow

**C# Method Overloading**

You can perform method overloading in C# by two ways:

1. By changing number of arguments add(**int** a,**int** b) & add(**int** a, **int** b, **int** c)
2. By changing data type of the arguments add(**float** a, **float** b)  & add(**int** a, **int** b)

To perform method overriding in C#, you need to use **virtual** keyword with base class method and **override** keyword with derived class method.

Because of = > Same class name and everything same

# 6] Exception Handling

## C# try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

try

{

// *Block of code to try*

}

catch (Exception e)

{

// *Block of code to handle errors*

}

Eg.

try

{

int[] myNumbers = {1, 2, 3};

Console.WriteLine(myNumbers[10]);

}

catch (Exception e)

{

Console.WriteLine("Something went wrong.");

}

## Finally

The finally statement lets you execute code, after try...catch, regardless of the result:

C# finally block is used to execute important code which is to be executed whether exception is handled or not. It must be preceded by catch or try block.

## throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception class**. There are many exception classes available in C#: ArithmeticException, FileNotFoundException, IndexOutOfRangeException, TimeOutException, etc:

static void checkAge(int age)

{

if (age < 18)

{

throw new ArithmeticException("Access denied - You must be at least 18 years old.");

}

else

{

Console.WriteLine("Access granted - You are old enough!");

}

}

static void Main(string[] args)

{

checkAge(15);

}

The error message displayed in the program will be:

System.ArithmeticException: 'Access denied - You must be at least 18 years old.'

Checked exceptions stopes excuation  
and Unchecked does not stops execution.

# 7] Events, Delegates, Interfaces, Enums, and Lambda Expression

## C# Enums

An enum is a special "class" that represents a group of **constants** (unchangeable/read-only variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma:

class Program

{

enum Level

{

Low,

Medium,

High

}

static void Main(string[] args)

{

Level myVar = Level.Medium;

Console.WriteLine(myVar);

}

}

The output will be:

Medium

By default, the first item positions of an enum has the value 0. The second positions has the value 1, and so on.

To get the integer value from an item, you must [explicitly convert](https://www.w3schools.com/cs/cs_type_casting.php) the item to an int:

March, // 2

March=6, // 6

#### **Why And When To Use Enums?**

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

## C# Delegates

In C#, delegate is a reference to the method. It works like function pointer in C and C++. But it is object-oriented, secured and type-safe than function pointer.

For static method, delegate encapsulates method only. But for instance method, it encapsulates method and instance both.

The best use of delegate is to use as event.

Internally a delegate declaration defines a class which is the derived class of **System.Delegate**.

// C# program to illustrate the use of Delegates

using System;

namespace GeeksForGeeks {

// declare class "Geeks"

class Geeks {

// Declaring the delegates

// Here return type and parameter type should

// be same as the return type and parameter type

// of the two methods

// "addnum" and "subnum" are two delegate names

public delegate void addnum(int a, int b);

public delegate void subnum(int a, int b);

// method "sum"

public void sum(int a, int b)

{ Console.WriteLine("(100 + 40) = {0}", a + b); }

// method "subtract"

public void subtract(int a, int b)

{ Console.WriteLine("(100 - 60) = {0}", a - b); }

// Main Method

public static void Main(String []args)

{

// creating object "obj" of class "Geeks"

Geeks obj = new Geeks();

// creating object of delegate, name as "del\_obj1"

// for method "sum" and "del\_obj2" for method "subtract" &

// pass the parameter as the two methods by class object "obj"

// instantiating the delegates

addnum del\_obj1 = new addnum(obj.sum);

subnum del\_obj2 = new subnum(obj.subtract);

// pass the values to the methods by delegate object

del\_obj1(100, 40);

del\_obj2(100, 60);

// These can be written as using

// "Invoke" method

// del\_obj1.Invoke(100, 40);

// del\_obj2.Invoke(100, 60);

} } }

(100 + 40) = 140

(100 - 60) = 40

## C# Interface

An interface is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies):

It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.

By default, members of an interface are abstract and public.

// Interface

interface IAnimal

{

void animalSound(); // interface method (does not have a body)

}

// Pig "implements" the IAnimal interface

class Pig : IAnimal

{

public void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

}

}

The pig says: wee wee

**Notes on Interfaces:**

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interfaces can contain properties and methods, but not fields/variables
* Interface members are by default abstract and public
* An interface cannot contain a constructor (as it cannot be used to create objects)

**Why And When To Use Interfaces?**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

**Multiple Interfaces**

To implement multiple interfaces, separate them with a comma:

interface IFirstInterface

{

void myMethod(); // interface method

}

interface ISecondInterface

{

void myOtherMethod(); // interface method

}

// Implement multiple interfaces

class DemoClass : IFirstInterface, ISecondInterface

{

public void myMethod()

{

Console.WriteLine("Some text..");

}

public void myOtherMethod()

{

Console.WriteLine("Some other text...");

}

}

class Program

{

static void Main(string[] args)

{

DemoClass myObj = new DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

Some text..  
Some other text...

# 8] Generics

## C# Generics

Generic is a concept that allows us to define classes and methods with placeholder. C# compiler replaces these placeholders with specified type at compile time. The concept of generics is used to create general purpose classes and methods.

o define generic class, we must use angle **<>** brackets. The angle brackets are used to declare a class or method as generic type. In the following example, we are creating generic class that can be used to deal with any type of data.

**using** System;

**namespace** CSharpProgram

{

**class** GenericClass<T>

    {

**public** GenericClass(T msg)

        {

            Console.WriteLine(msg);

        }

    }

**class** Program

    {

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

        {

            GenericClass<**string**> gen   = **new** GenericClass<**string**> ("This is generic class");

            GenericClass<**int**>    genI  = **new** GenericClass<**int**>(101);

            GenericClass<**char**>   getCh = **new** GenericClass<**char**>('I');

        }

    }

}

This is generic class

101

I

# 9] Multi-threading

Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time. To create multithreaded application in C#, we need to use **System.Threding** namespace.

A list of commonly used classes are given below:

* Thread
* Mutex
* Timer
* Monitor
* Semaphore
* ThreadLocal
* ThreadPool
* Volatile etc.

**mutex**

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for [Mutual Exclusion](https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/) Object. Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time.

**Semaphore**

A semaphore is a non-negative integer variable that is shared between various threads. Semaphore works upon signaling mechanism, in this a thread can be signaled by another thread. Semaphore uses two atomic operations for process synchronisation:

Wait (P)

Signal (V)

In C#, each thread has a life cycle. The life cycle of a thread is started when instance of *System.Threading.Thread class* is created. When the task execution of the thread is completed, its life cycle is ended.

There are following states in the life cycle of a Thread in C#.

* Unstarted
* Runnable (Ready to run)
* Running
* Not Runnable
* Dead (Terminated)

The first thread which is created inside a process is called Main thread. It starts first and ends at last.

**using** System;

**using** System.Threading;

|  |  |
| --- | --- |
| Abort() | is used to terminate the thread. It raises ThreadAbortException. |
| Interrupt() | is used to interrupt a thread which is in *WaitSleepJoin* state. |
| Join() | is used to block all the calling threads until this thread terminates. |
| ResetAbort() | is used to cancel the Abort request for the current thread. |
| Resume() | is used to resume the suspended thread. It is obselete. |
| Sleep(Int32) | is used to suspend the current thread for the specified milliseconds. |
| Start() | changes the current state of the thread to Runnable. |
| Suspend() | suspends the current thread if it is not suspended. It is obselete. |
| Yield() | is used to yield the execution of current thread to another thread |

# 10] Asynchronous Programming

**Asynchronous Programming allows us to use threads efficiently and threads are prevented from being unnecessarily blocked.**

<https://dotnettutorials.net/lesson/introduction-to-parallel-and-asynchronous-programming-in-csharp/>

Parallel Programming helps us to divide a task into different parts and execute those parts simultaneously

if we have a web application, it will be able to serve more HTTP requests at the same time by using Asynchronous Programming. This is because each HTTP request is handled by a thread, and if we avoid blocking threads, then there will be more threads available to process HTTP requests.

C# we use **async** and **await**keywords

**async** keyword to mark a method as asynchronous and with **await**, we can wait for an asynchronous operation in such a way that the original thread is not blocked.

The method which is marked with the **async** keyword must return a **Task** or **Task<T>**. The idea of a **Task** is that it represents an asynchronous operation and does not return anything. In the case of **Task<T>,** it is like a promise that in the future this method will return a value of the data type T.

A screenshot of a computer program

Description automatically generated

when we use await operator, what we are doing is, we are freeing the current thread from having to wait for the execution of the task. In this way, we are avoiding blocking the current thread that we’re using and then that thread can be used in another task.

**public** **async** **static** **void** SomeMethod**()**

**{**

Console.WriteLine**(**"Some Method Started......"**)**;

//Thread.Sleep(TimeSpan.FromSeconds(10));

**await** Task.Delay**(**TimeSpan.FromSeconds**(**10**))**;

Console.WriteLine**(**"\n"**)**;

Console.WriteLine**(**"Some Method End"**)**;

**}**

# 11] Collections in C#

In C#, collection represents group of objects.

We can store objects in array or collection. Collection has advantage over array. Array has size limit but objects stored in collection can grow or shrink dynamically.

Loosely typed means you can store any type of value in the collection. Because of this loosely typed nature, we may get runtime errors.

So the solution is Generic Collections in C#.

1. [**Array**](https://dotnettutorials.net/lesson/arrays-csharp/): Type-Safe but Fixed Length
2. [**Collections**](https://dotnettutorials.net/lesson/collections-csharp/): Auto Resizing but not Type-Safe
3. [**Generic Collections**](https://dotnettutorials.net/lesson/generic-collections-csharp/): Type-Safe and Auto-Resizing

**Types of Collections in C#**

There are 3 ways to work with collections. The three namespaces are given below:

* **System.Collections.Generic** classes
* **System.Collections** classes (Now deprecated)
* **System.Collections.Concurrent** classes

## 1) System.Collections.Generic classes

The System.Collections.Generic namespace has following classes:

* List
* Stack
* Queue
* LinkedList
* HashSet
* SortedSet
* Dictionary
* SortedDictionary
* SortedList

### **C# List<T>**

C# List<T> class is used to store and fetch elements. It can have duplicate elements. It is found in System.Collections.Generic namespace.

##### **Generic List<T> Collection in C#**

 This Generic List<T> Collection Class represents a strongly typed list of objects which can be accessed by using the integer index which is starting from 0

 var names = **new** List<**string**>();

        names.Add("Sonoo Jaiswal");

 // Create a list of strings using collection initializer

        var names = **new** List<**string**>() {"Sonoo", "Vimal", "Ratan", "Love" };

  // Iterate through the list.

**foreach** (var name **in** names)

        {

            Console.WriteLine(name);

        }

**A close-up of text

Description automatically generated**

A white background with blue text

Description automatically generated

##### **How to Insert Elements at a Specific Position in a Generic List Collection in C#?**

**//Insert Element at Index Position 1**

**countries.Insert(1, "China");**

countries.Contains**(**"NZ"**))**;  
returns true or false,

//Removing Element using Index Position from the List

countries.RemoveAt**(**2**)**;

countries.Clear**()**;

countries.Remove("SRILANKA")

##### **How to Copy an Array to a List in C#?**

// Create new array with 3 elements.

**string[]** array = new **string[]** **{** "INDIA", "USA", "UK" **}**;

// Copy the array to a List.

List**<string>** copiedList = new List**<string>(**array**)**;

List**<int>** numbersList = new List**<int>** **{** 1, 8, 7, 5, 2 **}**;

numbersList.Sort**()**;

numbersList.Reverse**()**;

<https://dotnettutorials.net/lesson/list-collection-csharp/> More In Detail.

### **C# Stack<T>**

C# Stack<T> class is used to push and pop elements. It uses the concept of Stack that arranges elements in LIFO (Last In First Out) order. It can have duplicate elements. It is found in System.Collections.Generic namespace.

**Stack(IEnumerable<T> collection):** It is used to initialize a new instance of the Generic Stack class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied.

 Here, the parameters collection specifies the collection to copy elements from. If the collection is null, then it will throw ArgumentNullException

**using System.Collections.Generic;**

**Stack<type> stack = new Stack<type>();**

Stack**<int>** stack = new Stack**<int>()**;

stack.Push**(**10**)**;

 Stack<**string**> names = **new** Stack<**string**>();

        names.Push("Sonoo");

 Console.WriteLine("Pop: "+ names.Pop());

        Console.WriteLine("After Pop, Peek element: " + names.Peek());

genericStack.Contains(50);

returns true or false

//Copying the stack to an object array

**int[]** stackCopy = new **int[**5**]**;

genericStack.CopyTo**(**stackCopy, 0**)**;

### **C# Queue<T>**

C# Queue<T> class is used to Enqueue and Dequeue elements. It uses the concept of Queue that arranges elements in FIFO (First In First Out) order. It can have duplicate elements. It is found in System.Collections.Generic namespace.

**using System.Collections.Generic;**

**Queue<type> Queue\_Name = new Queue<type>();**

Queue**<int>** queue = new Queue**<int>()**;

//Adding Elements to the Queue using Enqueue Method

queue.Enqueue**(**10**)**;

queue.Dequeue()

queue.Peek()

The Dequeue() method removes and returns the item at the beginning of the queue, whereas the Peek() method returns the item at the beginning of the queue, without removing it.

 Queue<**string**> names = **new** Queue<**string**>();

        names.Enqueue("Sonoo");

 Console.WriteLine("Dequeue: "+ names.Dequeue());

        Console.WriteLine("After Dequeue, Peek element: " + names.Peek());

queue.Contains(50) returns T or F

//Copying the queue to an object array

**int[]** queueCopy = new **int[**5**]**;

queue.CopyTo**(**queueCopy, 0**)**;

### **C# LinkedList<T>**

C# LinkedList<T> class uses the concept of linked list. It allows us to insert and delete elements fastly. It can have duplicate elements. It is found in System.Collections.Generic namespace.

It allows us to add and remove element at before or last index.

**using System.Collections.Generic;**

 // Create a list of strings

        var names = **new** LinkedList<**string**>();

        names.AddLast("Sonoo Jaiswal");

LinkedList**<string>** linkedList = new LinkedList**<string>()**;

linkedList.AddLast**(**"India"**)**;

linkedList.Remove**(**USANode**)**;

linkedList.AddFirst**(**"UK"**)**;

linkedList.Contains**(**"India"**)**

linkedList.AddAfter**(**USANode, "UK"**)**;

linkedList.AddBefore**(**USANode, "UK"**)**;

### **C# HashSet<T>**

C# HashSet class can be used to store, remove or view elements. It does not store duplicate elements. It is suggested to use HashSet class if you have to store only unique elements. It is found in System.Collections.Generic namespace.

// Create a set of strings

        var names = **new** HashSet<**string**>();

        names.Add("Sonoo");

   // Create a set of strings

        var names = **new** HashSet<**string**>(){"Sonoo", "Ankit", "Peter", "Irfan"};

        // Iterate HashSet elements using foreach loop

**foreach** (var name **in** names)

        {

            Console.WriteLine(name);

        }

HashSet**<string>** hashSetCountries = new HashSet**<string>()**;

//Adding Elements to HashSet using Add Method

hashSetCountries.Add**(**"INDIA"**)**

hashSetCountries.Remove**(**"Bangladesh"**)**;

hashSetCountries.Contains**(**"INDIA"**)**

##### **Set Operations on Generic HashSet<T> Collection Class in C#**

Of two hashset

**UnionWith(IEnumerable<T> other):**

// Using UnionWith method

hashSetCountries1.UnionWith**(**hashSetCountries2**)**;

**IntersectWith(IEnumerable<T> other):**

**ExceptWith(IEnumerable<T> other):**

**SymmetricExceptWith(IEnumerable<T> other):**

### **C# SortedSet<T>**

C# SortedSet class can be used to store, remove or view elements. It maintains ascending order and does not store duplicate elements. It is suggested to use SortedSet class if you have to store unique elements and maintain ascending order. It is found in System.Collections.Generic namespace.

       // Create a set of strings

        var names = **new** SortedSet<**string**>();

        names.Add("Sonoo");

   // Create a set of strings

        var names = **new** SortedSet<**string**>(){"Sonoo", "Ankit", "Peter", "Irfan"};

        // Iterate SortedSet elements using foreach loop

**foreach** (var name **in** names)

        {

            Console.WriteLine(name);

        }

SortedSet**<int>** sortedSetNumbers = new SortedSet**<int>()**;

//Adding Elements to SortedSet using Add Method

sortedSetNumbers.Add**(**10**)**;

sortedSetCountries.Remove**(**"Bangladesh"**)**;

sortedSetCountries.Contains**(**"INDIA"**))**;

all same as hashset due to set origin

### **C# Dictionary<TKey, TValue>**

C# Dictionary<TKey, TValue> class uses the concept of hashtable. It stores values on the basis of key. It contains unique keys only. By the help of key, we can easily search or remove elements. It is found in System.Collections.Generic namespace.

 Dictionary<**string**, **string**> names = **new** Dictionary<**string**, **string**>();

        names.Add("1","Sonoo");

**using System.Collections.Generic;**

//Creating a Dictionary with Key and value both are string type using Collection Initializer

Dictionary**<string**, **string>** dictionaryCountries = new Dictionary**<string**, **string>**

**{**

**{** "UK", "United Kingdom" **}**,

**{** "USA", "United State of America" **}**,

**{** "IND", "India" **}**

**}**;

dictionaryCountries.ContainsValue**(**"India"**)**

dictionaryCountries.ContainsKey**(**"USA"**)**

**if** **(**dictionaryCountries.ContainsKey**(**"PAK"**))**

**{**

dictionaryCountries.Remove**(**"PAK"**)**;

##### **How to Assign Values to a Dictionary with Indexer in C#?**

//Assign Values to a Dictionary with Indexer

dictionaryCountries**[**"IND"**]** = "India";

##### **How to Update a Dictionary in C# using Indexer?**

//Updating the key UK and USA using Indexer

dictionaryCountries**[**"UK"**]** = "United Kingdom Updated";

dictionaryCountries**[**"IND"**]** = "India Updated";

### **C# SortedList<TKey, TValue>**

C# SortedList<TKey, TValue> . It is found in System.Collections.Generic namespace.

The Generic SortedList<TKey, TValue> Collection Class in C# represents a collection of key/value pairs that are sorted by the keys based on the associated IComparer implementation. By default, it sorts the key/value pairs in ascending order

In C#, the SortedList<TKey, TValue> Collection allows us to store duplicate values, but the keys must be unique

/Creating Generic SortedList Collection

SortedList**<int**, **string>** genericSortedList = new SortedList**<int**, **string>()**;

//Adding Elements to SortedList Collection using Add Method in Random Order

genericSortedList.Add**(**1, "One"**)**;

genericSortedList.Add**(**5, "Five"**)**;

genericSortedList.ContainsKey**(**"Ind"**)**

genericSortedList.ContainsValue**(**"India"**)**

### **C# SortedDictionary<TKey, TValue>**

The SortedDictionary<TKey, TValue> is a Generic Collection class in C# which is used to store the key/value pairs in the sorted form and the sorting is done based on the key. The SortedDictionary<TKey, TValue> class uses the concept of the hashtable.. It is found in System.Collections.Generic namespace.

The keys of the SortedDictionary collection must be unique and maintains ascending order based on the key. With the help of a key, we can easily search or remove elements from the generic SortedDictionary<TKey, TValue> collection.

 Dictionary<**string**, **string**> names = **new** Dictionary<**string**, **string**>();

        names.Add("1","Sonoo");

SortedDictionary**<int**, **string>** genericSortedDictionary = new SortedDictionary**<int**, **string>()**;

//Adding Elements to SortedDictionary Collection using Add Method in Random Order

genericSortedDictionary.Add**(**1, "One"**)**;

genericSortedDictionary.Add**(**3, "Three"**)**;

/ Remove value having key SL Using Remove() method

genericSortedDictionary.Remove**(**"SL"**)**;

genericSortedDictionary.ContainsKey**(**"Ind"**)**

genericSortedDictionary.ContainsKey**(**"NZ"**)**

and same indexers as dictionary to add and update values.

## 2) System.Collections classes

These classes are legacy. It is suggested now to use System.Collections.Generic classes. The System.Collections namespace has following classes:

* ArrayList
* Stack
* Queue
* Hashtable

### C# ArrayList Non-Generic

Generic is a concept that allows us to define classes and methods with placeholder. C# compiler replaces these placeholders with specified type at compile time. The concept of generics is used to create general purpose classes and methods.

/Adding elements to ArrayList using Add() method

ArrayList arrayList1 = new ArrayList**()**;

arrayList1.Add**(**101**)**; //Adding Integer Value

arrayList1.Add**(**"James"**)**; //Adding String Value

//Insert "aalu" at First Position i.e. Index 0

arrayList.Insert**(**0, "aalu"**)**;

ArrayList arrayList2 = new ArrayList**()**

**{**

"Srilanka",

"Japan",

"Britem"

**}**;

arrayList1.InsertRange**(**2, arrayList2**)**;

arrayList.Remove**(**"HongKong"**)**; //Removes first occurance of null

arrayList.RemoveRange**(**0, 2**)**;

arrayList.RemoveAt**(**3**)**;

//Remove all items from the Array list

arrayList.Clear**()**;

arrayList.Contains("India")

// Sorting the elements of ArrayList Using sort() method

arrayList.Sort**()**;

loosely typed means contains any type of values.

## 3) System.Collections.Concurrent classes

The System.Collections.Concurrent namespace provides classes for thread-safe operations. Now multiple threads will not create problem for accessing the collection items.

System.Collections.Generic namespace classes are Type-Safe but not Thread Safe.

The System.Collections.Concurrent namespace has following classes:

* BlockingCollection
* ConcurrentBag
* ConcurrentStack
* ConcurrentQueue
* ConcurrentDictionary
* Partitioner
* Partitioner
* OrderablePartitioner

### **C# ConcurrentDictionary**

**System.Collections.Concurrent** Collection Classes

**TryAdd** method instead of the Add method.

using System;

using System.Threading;

using System.Collections.Concurrent;

using System.Collections.Generic;

namespace ConcurrentCollections

{

class Program

{

static ConcurrentDictionary<int, string> dictionary = new ConcurrentDictionary<int, string>();

static void Main(string[] args)

{

Thread t1 = new Thread(Method1);

Thread t2 = new Thread(Method2);

t1.Start();

t2.Start();

t1.Join();

t2.Join();

foreach (KeyValuePair<int, string> item in dictionary)

{

Console.WriteLine($"Key:{item.Key}, Value:{item.Value}");

}

Console.ReadKey();

}

public static void Method1()

{

for (int i = 0; i < 10; i++)

{

dictionary.TryAdd(i, "Added By Method1 " + i);

Thread.Sleep(100);

}

}

public static void Method2()

{

for (int i = 0; i < 10; i++)

{

dictionary.TryAdd(i, "Added By Method2 " + i);

Thread.Sleep(100);

}

}

}

}

1. **AddOrUpdate**: Adds a new entry if does not exist else updates the existing one
2. **GetOrAdd**: Retrieves an item if exists, else first adds it then retrieve it
3. **TryAdd**, **TrygetValue**, **TryUpdate**, **TryRemove**: Try to do the specified operation like Add/Get/Update/Remove. It returns true if performing the operation and false if failing to do the operation.

### **C# ConcurrentQueue**

//Creating a ConcurrentQueue to Store Integer Values

ConcurrentQueue**<int>** concurrentQueue = new ConcurrentQueue**<int>()**;

//Adding Elements to ConcurrentQueue using Enqueue Method

concurrentQueue.Enqueue**(**10**)**;

**bool** IsRemoved = concurrentQueue.TryDequeue**(**out **int** Result**)**;

# 11] File Handling

# 12] Parallel Programming

**C# Supports Two Types of Parallelism:**

**Data Parallelism:**In Data Parallelism, we have a collection of values and we want to use the same operation on each of the elements in the collection

1. [**Parallel.For**](https://dotnettutorials.net/lesson/parallel-for-method-csharp/)
2. [**Parallel.ForEach**](https://dotnettutorials.net/lesson/parallel-foreach-method-csharp/)

**Task Parallelism:**Task Parallelism occurs when we have a set of independent tasks that we want to perform in parallel.

1. [**Parallel.Invoke**](https://dotnettutorials.net/lesson/parallel-invoke-method-csharp/)

## ****Parallel For Loop in C#:****

Try to run the code multiple times and you might get different orders of the numbers in the console.

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

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

**{**

Console.WriteLine**(**"C# Parallel For Loop"**)**;

//It will start from 1 until 10

//Here 1 is the start index which is Inclusive

//Here 11 us the end index which is Exclusive

//Here number is similar to i of our standard for loop

//The value will be store in the variable number

Parallel.For**(**1, 11, number =**>** **{**

Console.WriteLine**(**number**)**;

**})**;

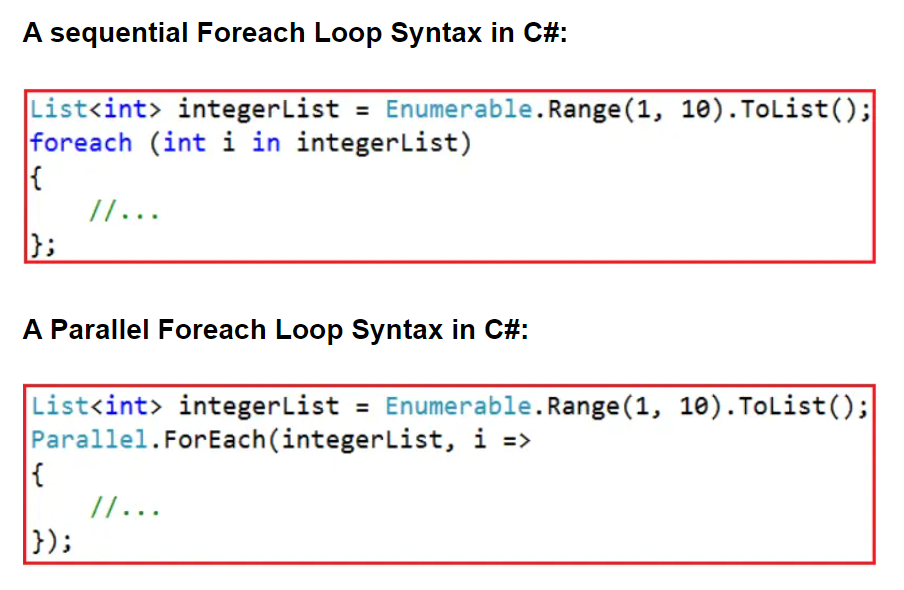
Console.ReadLine**()**;

**}**

**}**

**}**

## ****Parallel Foreach Loop in C#****



# 13] Auto Mapper

**How do we avoid Inheritance, how do we avoid abstract keywords, or how do we avoid the OOPs principle in a class?**

The answer is by using the **static** keyword. So, you need to mark the **CommonTask** class as **static** by using the static keyword. When you mark a class as **static**, everything inside the class should be static

* Use Static Class
* In Real-Time Projects, many times we need to map the objects between the UI Layer or Presentation Layer and Business Logic layers.
* solution is **AutoMapper**.
* The AutoMapper in C# is a mapper between two objects. That is AutoMapper is an Object-Object Mapper. It maps the properties of two different objects by transforming the input object of one type to the output object of another type
* **Step 1: Installing AutoMapper Library in Your Project**

Step2: Configuring and Initializing AutoMapper in C#

* A close-up of a logo

  Description automatically generated
* A white background with text

  Description automatically generated
* what we are going to do is, we will create a separate class where we will do all our mapping configurations. So, create a class file with the name MapperConfig.cs
* using AutoMapper;
* namespace AutoMapperDemo
* {
* public class MapperConfig
* {
* public static Mapper InitializeAutomapper()
* {
* //Provide all the Mapping Configuration
* var config = new MapperConfiguration(cfg =>
* {
* //Configuring Employee and EmployeeDTO
* cfg.CreateMap<Employee, EmployeeDTO>();
* //Any Other Mapping Configuration ....
* });
* //Create an Instance of Mapper and return that Instance
* var mapper = new Mapper(config);
* return mapper;
* }
* }
* }
* **Step 3: Using AutoMapper in C#:**
* //Create and Populate Employee Object i.e. Source Object
* Employee emp = new Employee
* **{**
* Name = "James",
* Salary = 20000,
* Address = "London",
* Department = "IT"
* **}**;
* //Initializing AutoMapper
* var mapper = MapperConfig.InitializeAutomapper**()**;
* //Way1: Specify the Destination Type and to The Map Method pass the Source Object
* //Now, empDTO1 object will having the same values as emp object
* var empDTO1 = mapper.Map**<**EmployeeDTO**>(**emp**)**;
* //Way2: Specify the both Source and Destination Type
* //and to The Map Method pass the Source Object
* //Now, empDTO2 object will having the same values as emp object
* var empDTO2 = mapper.Map**<**Employee, EmployeeDTO**>(**emp**)**;

**How do Map Properties when the Property Names are Different using AutoMapper?**

* //Provide all the Mapping Configuration
* var config = new MapperConfiguration**(**cfg =**>**
* **{**
* //Configuring Employee and EmployeeDTO
* cfg.CreateMap**<**Employee, EmployeeDTO**>()**
* //Provide Mapping Configuration of FullName and Name Property
* .ForMember**(**dest =**>** dest.FullName, act =**>** act.MapFrom**(**src =**>** src.Name**))**
* //Provide Mapping Dept of FullName and Department Property
* .ForMember**(**dest =**>** dest.Dept, act =**>** act.MapFrom**(**src =**>** src.Department**))**;
* //Any Other Mapping Configuration ....
* **})**;
* //Create an Instance of Mapper and return that Instance
* var mapper = new Mapper**(**config**)**;
* **return** mapper;

# 14] ADO.NET

# 15] LINQ (Language Integrated Query)

# 16] Extras

# Task in C#

In C#, when we have an asynchronous method, in general, we want to return one of the following data types.

1. **Task and Task<T>**
2. **ValueTask and ValueTask<T>**

The Task data type represents an asynchronous operation. A task is basically a “promise” that the operation to be performed will not necessarily be completed immediately, but that it will be completed in the future.

In asynchronous programming when your method does not return anything, then instead of using void you can use Task.

# **How to Return a Value from a Task in C#?**

generic version of the Task class i.e. Task<T>. Using this Task<T> class we can return data or values from a task. In Task<T>, T represents the data type that you want to return as a result of the task. With Task<T>, we have the representation of an asynchronous method that is going to return something in the future. That something could be a string, a number, a class, etc.

# Example to Understand How to Return Complex Type Value from a Task in C#:

In the below example, we are returning a Complex type.

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

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

**{**

Console.WriteLine**(**$"Main Thread Started"**)**;

SomeMethod**()**;

Console.WriteLine**(**$"Main Thread Completed"**)**;

Console.ReadKey**()**;

**}**

**private** **async** **static** **void** SomeMethod**()**

**{**

Employee emp = **await** GetEmployeeDetails**()**;

Console.WriteLine**(**$"ID: {emp.ID}, Name : {emp.Name}, Salary : {emp.Salary}"**)**;

**}**

**static** **async** Task**<**Employee**>** GetEmployeeDetails**()**

**{**

Employee employee = new Employee**()**

**{**

ID = 101,

Name = "James",

Salary = 10000

**}**;

**return** employee;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **double** Salary **{** **get**; **set**; **}**

**}**

**}**

A black and white screen with white text

Description automatically generated