**Understand Recursion in C#**

Function calling itself is called Recursion. Or, in simple words, we can say that recursion is a process in which a function calls itself repeatedly until some specified condition has been satisfied.

using System;

namespace RecursionDemo

{

class Program

{

static void Main(string[] args)

{

int x = 3;

fun(x);

Console.ReadKey();

}

static void fun(int n)

{

if (n > 0)

{

fun(n - 1); //Function calling itself

Console.WriteLine($"{n} ");

}

}

}

}



**Call by Value and Call by Reference in C#**

##### ****Call By Value With Value Type in C#****

##### In .NET Framework, by default, all the objects are called by value, not called by reference. So, whether it is a Value Type (Primitive data types like int, char, double, etc.) or Reference Data Type (class, interface, delegate, string, etc.), they will be called by value by default.

namespace FunctionsDemo

{

class Program

{

static void Main(string[] args)

{

int a = 15;

int b = a;

b = 30;

Console.WriteLine(a);

Console.ReadKey();

}

}

}



Answer is 15, then you are absolutely right because **int** is a value data type, and by default, it is passed by value, which means for the above code, the variable “a” has stored the value 15 in it. When we create the variable b and assign it a, the value of a is copied to b, and after that, if we change b, it will not affect a. This is because we have copied the value of a to b.

##### ****Call by Value with Reference Types in C#****

namespace FunctionsDemo

{

class Program

{

static void Main(string[] args)

{

Employee Emp1 = new Employee();

Emp1.EmployeeID = 1001;

Emp1.Name = "James";

Employee Emp2 = Emp1;

Emp1.Name = "Smith";

Console.WriteLine($"Emp1 Name = {Emp1.Name}");

Console.WriteLine($"Emp2 Name = {Emp2.Name}");

Console.ReadKey();

}

}

public class Employee

{

public int EmployeeID;

public string Name;

}

##### }

##### 

##### When we assign Emp1 to Emp2, the reference of the object memory location that Emp1 is holding is copied to Emp2. Now we have two copies of reference, but they both point to the same memory location. So, changing the Name property’s value will change the object’s value in memory, which we have references in Emp1 and Emp2. So, “Smith” will be printed on the console for both references.

##### ****Call By Reference with Value Type in C#****

namespace FunctionsDemo

{

class Program

{

static void Main(string[] args)

{

int a = 15;

UpdateValue(ref a);

Console.WriteLine(a);

Console.ReadKey();

}

static void UpdateValue(ref int b)

{

b = 30;

}

}

}



It will print 30 on the console. This is because of the ref keyword. In the above example, when the UpdateValue method is called, **the incoming parameter b has the same memory address as a, which is passed as the argument.** That’s why modifying the value of b would also reflect the change in a. In this case, a new memory location is not created for method parameter b. Here, you can say b just as an alias of a or an alternative name of a. So, if we make any changes using variable b, it will also reflect in variable a.

##### ****Call by Reference with Reference Types in C#****

using System;

namespace FunctionsDemo

{

class Program

{

static void Main(string[] args)

{

Employee Emp1 = new Employee();

Emp1.EmployeeID = 1001;

Emp1.Name = "James";

UpdateName(ref Emp1);

Console.WriteLine($"Emp1 Name = {Emp1.Name}");

Console.ReadKey();

}

static void UpdateName(ref Employee Emp2)

{

Emp2.Name = "Kamal";

}

}

public class Employee

{

public int EmployeeID;

public string Name;

}

}



Now, when the UpdateName method sets the Emp2 to Kamal, this also sets the Emp1 to Kamal. This is because of Call By Reference. **So, in this case, the reference of Emp1 is passed to the method, and hence a new reference is not created for Emp2**. So, here, you can consider Emp2 as an alias of Emp1 or an alternative name for Emp1.

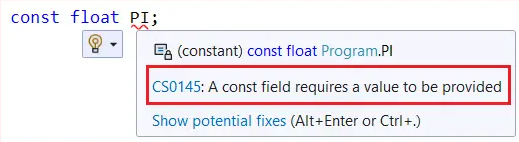
**Const, Read-Only and Static Variable in C#**

**Const Variables in c#**

When we declare a variable by using the const keyword, then it is a constant variable and the value of the constant variable can’t be modified once after its declaration. So, it is mandatory to initialize the constant variable at the time of its declaration only. Suppose, you want to declare a constant PI in your program, then you can declare the constant variable as follows:

const float PI = 3.14f;

If you are not initializing the const variable at the time of its declaration, then you will get a compiler error as shown in the below image.



**Read-Only Variables in C#**

When we declare a variable using the readonly keyword, it is known as a read-only variable. These variables can’t be modified after initialization, similar to constants. 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.

using System;

namespace ReadOnlyDemo

{

class Program

{

readonly int number = 5;

//You can also initialize through constructor

public Program()

{

number = 20;

}

static void Main(string[] args)

{

Program obj = new Program();

Console.WriteLine(obj.number);

//You cannot change the value of a readonly variable once it is initialized

//The following statement will give us compile time error

//obj.number = 20;

Console.ReadLine();

}

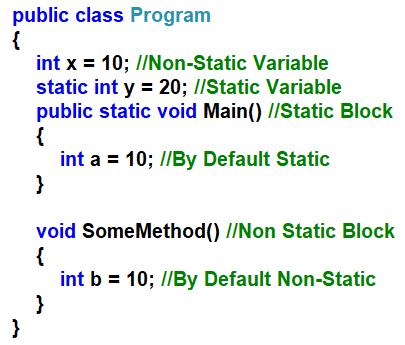
}

}



**Static Variables in C#**

Whenever we declare a variable by using the static modifier or when we declare a variable inside of any static block then those variables are considered static variables whereas the rest of the others are considered non-static variables.



If you want a **variable to have the same value throughout all instances of a class** then you need to declare that variable as a static variable. So, the static variables are going to hold the application-level data which is going to be the same for all the objects.

**Boxing and Unboxing in C#**

**Boxing:** Boxing is the process of converting a value type (like int, double, struct) to a reference type (object). When a value type is boxed, a new object is allocated to the heap, and the value is copied into it.

**Unboxing:** Unboxing is the reverse process of boxing, where a value is extracted from an object. It involves **explicitly converting a reference type (object) into a value type**. This operation also involves a copy operation, where the value is copied from the heap into the stack.

namespace BoxingUnboxingDemo

{

class Program

{

static void Main(string[] args)

{

int x = 10;

object y = x; //Boxing

int z = (int)y; //Unboxing

}

}

}

**Var, Object and Dynamic Keyword in C#**

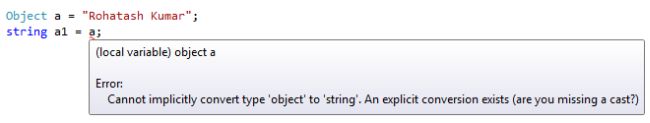
**Var**

* Var keywords are introduced in C# 3.0.
* You need to assign a value for the var keyword while declaring variables. Because its type is determined at compile time, it is a statically typed variable.
* You cannot change the data type of a variable once it has been declared using the var keywords.
* It provides intelligence once a value is assigned.

**Object**

* The object was introduced in C# 1.
* It can store any type of value because it is the base class for all types in C#.
* You need to typecast an object value to a specific type before you perform any manipulation on it.

Example



You need to do **explicit typecasting** every time you want to get the value back and forth.

object a = "Rohatash Kumar";

string a1 = a.ToString();

**Dynamic**

* It was introduced in C# 4.0.
* It can store any type of value, and the type of the variable is unknown until runtime, so it will not support IntelliSense.
* It's not mandatory to initialise at declaration time.

dynamic a = "Rohatash Kumar";

string a1 = a;

**What is an Exception in C#?**

An Exception is a class in C# which is responsible for abnormal termination of the program when runtime errors occur while running the program.  So, these errors (runtime) are very dangerous because whenever the runtime errors occur in the programs, the program gets terminated abnormally on the same line where the error gets occurred without executing the next line of code.

**Note:** Most people are saying Runtime Errors are Exceptions which is not true. Exceptions are classes that are responsible for abnormal termination of the program when runtime errors occur.

1. IndexOutOfRangeException
2. FormatException
3. NullReferenceException
4. DivideByZeroException
5. FileNotFoundException
6. SQLException,
7. OverFlowException, etc.

##### ****What happens if an Exception is Raised in the Program in C#?****

When an Exception is raised in C#, the program execution is terminated abnormally. That means the statements placed after the exception-causing statements are not executed but the statements placed before that exception-causing statement are executed by CLR.

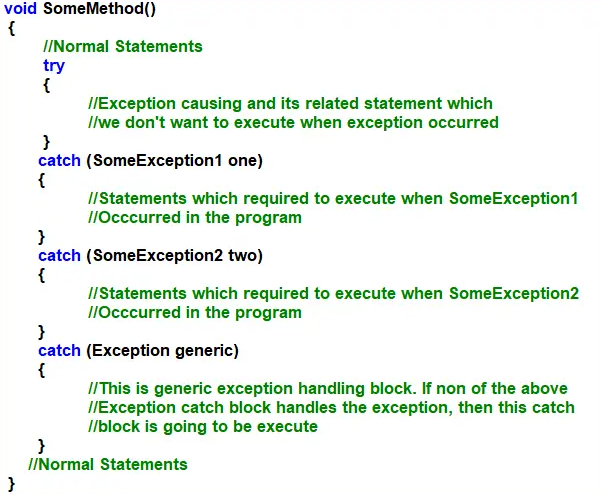
##### ****What CLR does when an Exception Occurred in the program?****

The CLR creates the exception class object that is associated with that logical mistake (exception) and terminates the program execution by throwing that exception object by using the throw keyword. So, we can say an exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction execution.

**What is Exception Handling in C#?**

The process of catching the exception for converting the CLR given exception message to an end-user understandable message and for stopping the abnormal termination of the program whenever runtime errors are occurring is called Exception Handling in C#. Once we handle an exception under a program we will be getting the following advantages.

1. We can stop the Abnormal Termination
2. We can perform any corrective action that may resolve the problem.
3. Displaying a user-friendly error message, so that the user can resolve the problem provided if it is under his control.



**Why do we need Exception Handling in C#?**

We need Exception Handling in C# because of the following two reasons.

1. To stop the Abnormal Termination of the program.
2. To provide users with understandable messages when an exception is raised. So that users can make a decision without the developer’s help.

**Multiple catch block example**

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

Console.WriteLine("Enter First Number");

Number1 = int.Parse(Console.ReadLine());

Console.WriteLine("Enter Second Number");

Number2 = int.Parse(Console.ReadLine());

Result = Number1 / Number2;

Console.WriteLine($"Result: {Result}");

}

catch (DivideByZeroException DBZE)

{

Console.WriteLine("Second Number Should Not Be Zero");

}

catch (FormatException FE)

{

Console.WriteLine("Enter Only Integer Numbers");

}

Console.ReadKey();

}

}

}

**Create user-defined exceptions**

.NET provides a hierarchy of exception classes ultimately derived from the [Exception](https://learn.microsoft.com/en-us/dotnet/api/system.exception) base class. However, if none of the predefined exceptions meet your needs, you can create your own exception classes by deriving from the [Exception](https://learn.microsoft.com/en-us/dotnet/api/system.exception) class.

Let's assume you want to create a StudentNotFoundException that contains a StudentName property. To create a custom exception, follow these steps.

1. Create a serializable class that inherits from [Exception](https://learn.microsoft.com/en-us/dotnet/api/system.exception). The class name should end in "Exception":

[Serializable]

public class StudentNotFoundException : Exception { }

1. Add the default constructors:

[Serializable]

public class StudentNotFoundException : Exception

{

public StudentNotFoundException() { }

public StudentNotFoundException(string message)

: base(message) { }

public StudentNotFoundException(string message, Exception inner)

: base(message, inner) { }

}

1. Define any additional properties and constructors:

[Serializable]

public class StudentNotFoundException : Exception

{

public string StudentName { get; }

public StudentNotFoundException() { }

public StudentNotFoundException(string message)

: base(message) { }

public StudentNotFoundException(string message, Exception inner)

: base(message, inner) { }

public StudentNotFoundException(string message, string studentName)

: this(message)

{

StudentName = studentName;

}

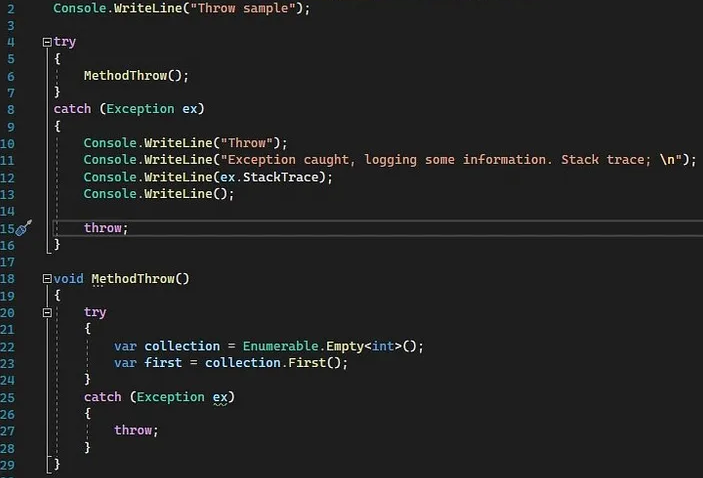
}

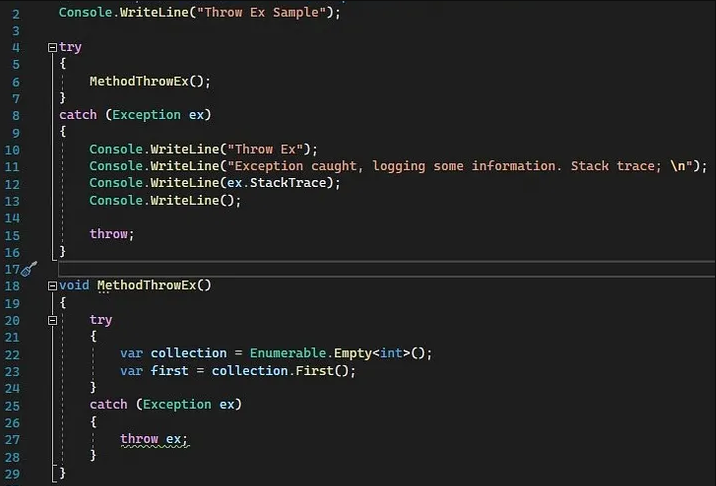
You have created a custom exception, and you can throw it anywhere with code like the following:

throw new StudentNotFoundException("The student cannot be found.", "John");

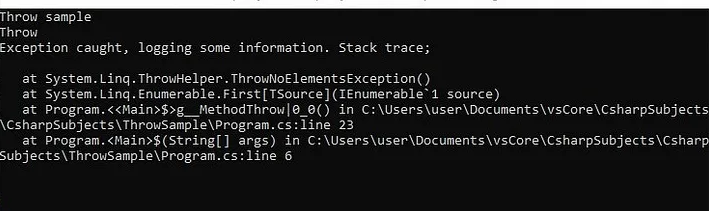
**throw” and “throw ex” in C#**

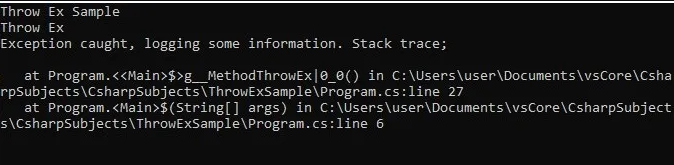
The difference between “throw” and “throw ex” is that “throw” preserves the stack trace while ”throw ex” does not preserve the stack trace.





Now. Let’s see the output of the programs.





**Delegates**

A delegate in C# is similar to a function pointer in C or C++.  It means they hold the reference of a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked. Unlike function pointers in C or C++, delegates are object-oriented, type-safe, and secure.

The syntax to create a delegate in C# is very much like the abstract method declaration. In abstract method declaration, we use the abstract keyword whereas, in delegate, we need to use the delegate keyword. The syntax for defining a delegate in C# is as follows:

**<Access Modifier> delegate <Return Type> <Delegate Name> (Parameter List);**

Example

**using System;**

public delegate void MyDelegate(string s);

class MyClass

{

public static void Hello(string s)

{

Console.WriteLine(" Hello, {0}!", s);

}

public static void Goodbye(string s)

{

Console.WriteLine(" Goodbye, {0}!", s);

}

public static void Main()

{

MyDelegate a, b, c, d;

// Create the delegate object a that references

// the method Hello:

a = new MyDelegate(Hello);

// Create the delegate object b that references

// the method Goodbye:

b = new MyDelegate(Goodbye);

// The two delegates, a and b, are composed to form c:

c = a + b;

// Remove a from the composed delegate, leaving d,

// which calls only the method Goodbye:

d = c - a;

Console.WriteLine("Invoking delegate a:");

a("A");

Console.WriteLine("Invoking delegate b:");

b("B");

Console.WriteLine("Invoking delegate c:");

c("C");

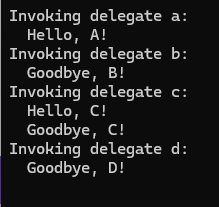
Console.WriteLine("Invoking delegate d:");

d("D");

Console.ReadKey();

}

}



 There are two types of Delegates in C#. They are as follows:

1. **Single Cast Delegate:** Delegate Refers to a single function or method.
2. **Multicast Delegate:** Delegate Refers to multiple functions or methods.

**What is Multicast Delegate in C#?**

A Multicast Delegate in C# is a delegate that holds the references of more than one handler function. When we invoke the multicast delegate, then all the functions which are referenced by the delegate are going to be invoked. If you want to call multiple methods using a delegate then all the method signatures should be the same.

**What are Generics in C#?**

Generics are the most powerful feature of C# 2.0. Generics allow you to define type-safe data structures, without committing to actual data types. This results in a significant performance boost and higher quality code, because you get to reuse data processing algorithms without duplicating type-specific code.

In other words, we can say that the Generics allow us to create classes using angular brackets specifying the data type of its members.

**Advantages of Generics in C#**

* **It Increases the Reusability of the Cod**e: For example, you can create a generic method to add two numbers. This method can be used to add two integers as well as two float numbers without any modification to the code.
* **Generics are Type-Safe:** Generics are type safety, especially in the case of collections. When using generics, we need to define the type of objects to be passed to a collection. This helps the compiler to ensure that only those object types that are defined in the definition can be passed to the collection. We will get the compile-time error if we try to use a different type of data rather than the one, we specified in the definition.
* **Generic Provides Better Performance**: It gives better performance compared to non-Generics **because they reduce the need for boxing, unboxing, and typecasting of variables or objects.**

using System;

namespace GenericsDemo

{

//MyGenericClass is a Generic Class

//Here T specifies the Data Types of the class Members

class MyGenericClass<T>

{

//Generic variable

//The data type is generic i.e. T

private T GenericMemberVariable;

//Generic Constructor

//Constructor accepts one parameter of Generic type i.e. T

public MyGenericClass(T value)

{

GenericMemberVariable = value;

}

//Generic Method

//Method accepts one Generic type Parameter i.e. T

//Method return type also Generic i.e. T

public T GenericMethod(T GenericParameter)

{

Console.WriteLine($"Parameter type: {typeof(T).ToString()}, Value: {GenericParameter}");

Console.WriteLine($"Return type: {typeof(T).ToString()}, Value: {GenericMemberVariable}");

return GenericMemberVariable;

}

}

}

using System;

namespace GenericsDemo

{

class Program

{

static void Main()

{

MyGenericClass<int> integerGenericClass = new MyGenericClass<int>(10);

int val1 = integerGenericClass.GenericMethod(200);

Console.WriteLine(val1);

MyGenericClass<string> stringrGenericClass = new MyGenericClass<string>("Kamal");

string val2 = stringrGenericClass.GenericMethod("Pratap");

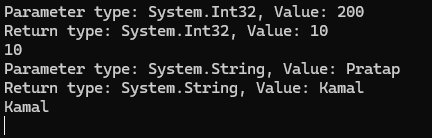
Console.WriteLine(val2);

Console.ReadKey();

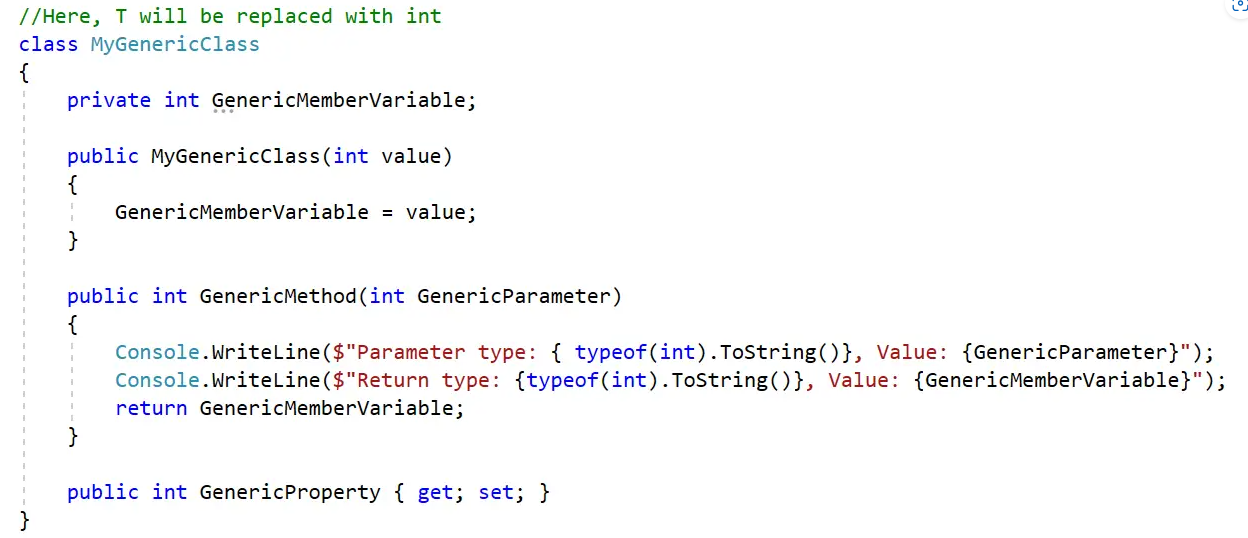
}

}

}



So, while creating the instance of this MyGenericClass class, we need to specify the type and the compiler will assign that type to T. In the following example, we use int as the data type. Once we create an instance of the MyGenericClass, then we are invoking the GenericMethod method. As we have specified the T as int while creating the instance, we do not need to specify the same while invoking the class members.



**What are Lambda Expressions in C#?**

The Lambda Expression in C# is the shorthand for writing the Anonymous Function. So, we can say that the Lambda Expression is nothing but to simplify the anonymous function in C#.

Anonymous Functions is a function without a name. Anonymous Functions are related to delegate and they are created by using the delegate keyword.

##### ****Understand Delegate using Method in C#****

using System;

namespace LambdaExpressionDemo

{

public class LambdaExpression

{

public delegate string GreetingsDelegate(string name);

public static string Greetings(string name)

{

return "Hello " + name + " welcome to Dotnet Tutorials";

}

static void Main(string[] args)

{

GreetingsDelegate obj = new GreetingsDelegate(Greetings);

string GreetingsMessage = obj.Invoke("Pranaya");

Console.WriteLine(GreetingsMessage);

Console.ReadKey();

}

}

}



##### ****Delegate using Anonymous Method in C#****

In the previous example, we used a named block while creating the delegate instance. Instead of a named block, we can also give an unnamed block which is called Anonymous Method. The anonymous methods are created using the delegate keyword and when we invoke the delegate, the anonymous method is going to be executed.

using System;

namespace LambdaExpressionDemo

{

public class LambdaExpression

{

public delegate string GreetingsDelegate(string name);

static void Main(string[] args)

{

GreetingsDelegate obj = delegate (string name)

{

return "Hello @" + name + " welcome to Dotnet Tutorials";

};

string GreetingsMessage = obj.Invoke("Pranaya");

Console.WriteLine(GreetingsMessage);

Console.ReadKey();

}

}

}

##### ****Create Lambda Expressions in C#****

##### Let us rewrite the previous example using the Lambda Expression in C#. And this time also, you will get the same output.

using System;

namespace LambdaExpressionDemo

{

public class LambdaExpression

{

public delegate string GreetingsDelegate(string name);

static void Main(string[] args)

{

GreetingsDelegate obj = (name) =>

{

return "Hello @" + name + " welcome to Dotnet Tutorials";

};

string GreetingsMessage = obj.Invoke("Pranaya");

Console.WriteLine(GreetingsMessage);

Console.ReadKey();

}

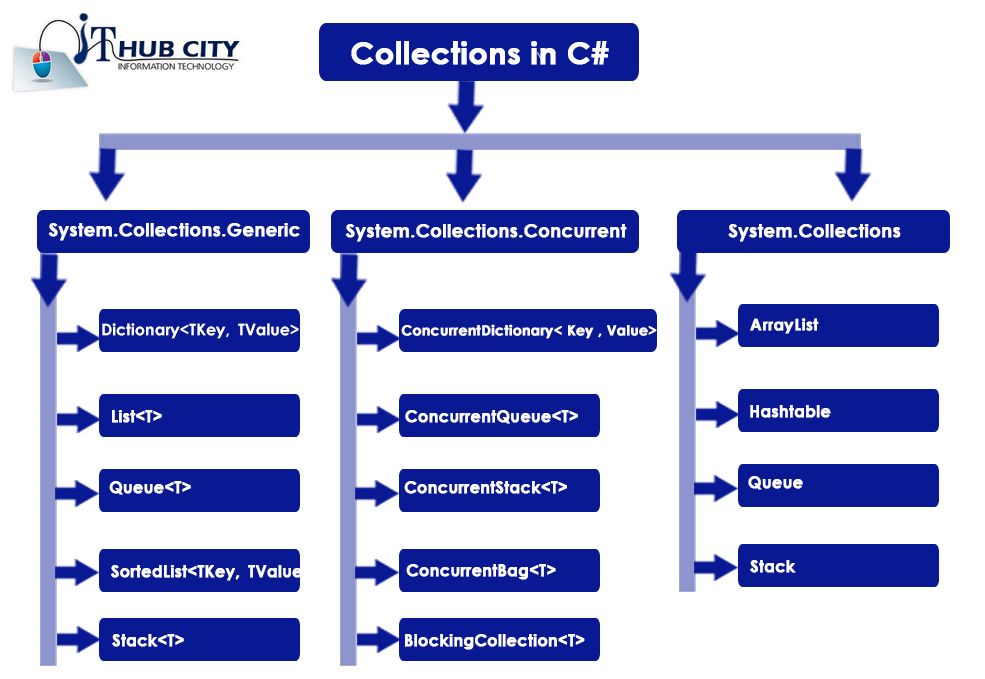
}

}

**What is a Collection in C#**

The **Collections in C#** are a set of predefined classes that are present in the **System.Collections** namespace that provides greater capabilities and functionalities than the traditional arrays. The collections in C# are reusable, more powerful, and more efficient and most importantly they have been designed and tested to ensure quality and performance.

##### **Types of Collections in C#**



Array List Example

using System;

using System.Collections;

namespace ColletionDemo

{

public class ColletionDemo

{

static void Main(string[] args)

{

//Adding elements to ArrayList using Add() method

ArrayList arrayList1 = new ArrayList();

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

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

arrayList1.Add("James"); //Adding Duplicate Value

arrayList1.Add(" "); //Adding Empty

arrayList1.Add(true); //Adding Boolean

arrayList1.Add(4.5); //Adding double

arrayList1.Add(null); //Adding null

for (int i = 0; i < arrayList1.Count; i++)

{

Console.WriteLine(arrayList1[i]);

}

Console.WriteLine("Other Method");

foreach (var item in arrayList1)

{

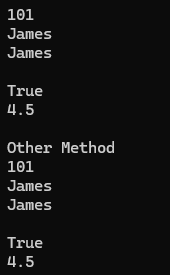
Console.WriteLine(item);

}

}

}

}



Hash Table Example

using System;

using System.Collections;

namespace ColletionDemo

{

public class ColletionDemo

{

static void Main(string[] args)

{

Hashtable hashtable = new Hashtable();

//Adding elements to the Hash table using key value pair

hashtable.Add("EId", 1001); //Here key is Eid and value is 1001

hashtable.Add("Name", "James"); //Here key is Name and value is James

hashtable.Add("Salary", 3500); //Here key is Salary and value is 3500

hashtable.Add("Location", "Mumbai"); //Here key is Location and value is Mumbai

hashtable.Add("EmailID", "a@a.com"); //Here key is EmailID and value is a@a.com

//Printing the keys and their values using foreach loop

Console.WriteLine("Printing Hashtable using Foreach Loop");

foreach (object obj in hashtable.Keys)

{

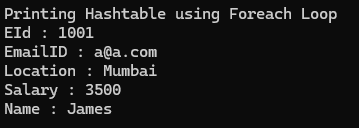
Console.WriteLine(obj + " : " + hashtable[obj]);

}

}

}

}



##### **Dictionary Collection in C#**

using System;

using System.Collections.Generic;

using System.Linq;

namespace GenericDictionaryDemo

{

class Program

{

static void Main()

{

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

Dictionary<string, string> dictionaryCountries = new Dictionary<string, string>();

//Adding Elements to the Dictionary using Add Method of Dictionary class

dictionaryCountries.Add("UK", "London, Manchester, Birmingham");

dictionaryCountries.Add("USA", "Chicago, New York, Washington");

dictionaryCountries.Add("IND", "Mumbai, Delhi, Bhubaneswar");

//Accessing Dictionary Elements using For Each Loop

Console.WriteLine("Accessing Dictionary Elements using For Each Loop");

foreach (KeyValuePair<string, string> KVP in dictionaryCountries)

{

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

}

Console.ReadKey();

}

}

}

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

using System;

using System.Collections.Generic;

namespace GenericListCollectionDemo

{

class Program

{

static void Main()

{

//Creating a Generic List of string type to store string elements

List<string> countries = new List<string>();

//Adding String Elements to the Generic List using the Add Method

countries.Add("INDIA");

countries.Add("USA");

//The following Two Statements will give compile time error as element is not string type

//countries.Add(100);

//countries.Add(true);

//Creating Another Generic List Collection of String Type

List<string> newCountries = new List<string>();

//Adding Elements using Add Method

newCountries.Add("JAPAN");

newCountries.Add("UK");

//Adding the newCountries collection into countries collection using AddRange Method

countries.AddRange(newCountries);

//Accessing Generic List Elements using ForEach Loop

Console.WriteLine("Accessing Generic List using For Each Loop");

foreach (var item in countries)

{

Console.WriteLine(item);

}

//Accessing Generic List Elements using For Loop

Console.WriteLine("\nAccessing Generic List using For Loop");

for (int i = 0; i < countries.Count; i++)

{

var element = countries[i];

Console.WriteLine(element);

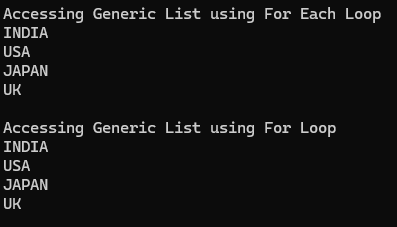
}

Console.ReadKey();

}

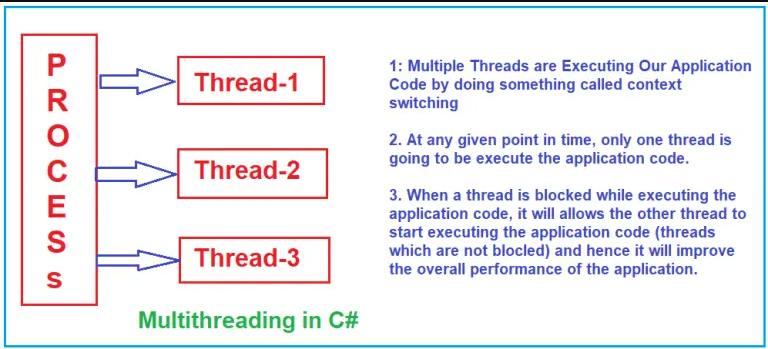
}

}

****

**Multithreading in C#**

Multithreading in C# refers to the ability of the C# programming language and the .NET Framework to create and manage multiple threads of execution within a single process. Threads are lightweight, independent sequences of instructions that can run concurrently, allowing you to perform multiple tasks simultaneously. Multithreading is a powerful concept in C# and is used to achieve various goals, such as improving application responsiveness, parallelizing tasks, and efficiently utilizing multi-core processors.

****

using System.Threading;

using System;

namespace ThreadingDemo

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Main Thread Started");

//Creating Threads

Thread t1 = new Thread(Method1)

{

Name = "Thread1"

};

Thread t2 = new Thread(Method2)

{

Name = "Thread2"

};

Thread t3 = new Thread(Method3)

{

Name = "Thread3"

};

//Executing the methods

t1.Start();

t2.Start();

t3.Start();

Console.WriteLine("Main Thread Ended");

Console.Read();

}

static void Method1()

{

Console.WriteLine("Method1 Started using " + Thread.CurrentThread.Name);

for (int i = 1; i <= 5; i++)

{

Console.WriteLine("Method1 :" + i);

}

Console.WriteLine("Method1 Ended using " + Thread.CurrentThread.Name);

}

static void Method2()

{

Console.WriteLine("Method2 Started using " + Thread.CurrentThread.Name);

for (int i = 1; i <= 5; i++)

{

Console.WriteLine("Method2 :" + i);

if (i == 3)

{

Console.WriteLine("Performing the Database Operation Started");

//Sleep for 10 seconds

Thread.Sleep(10000);

Console.WriteLine("Performing the Database Operation Completed");

}

}

Console.WriteLine("Method2 Ended using " + Thread.CurrentThread.Name);

}

static void Method3()

{

Console.WriteLine("Method3 Started using " + Thread.CurrentThread.Name);

for (int i = 1; i <= 5; i++)

{

Console.WriteLine("Method3 :" + i);

}

Console.WriteLine("Method3 Ended using " + Thread.CurrentThread.Name);

}

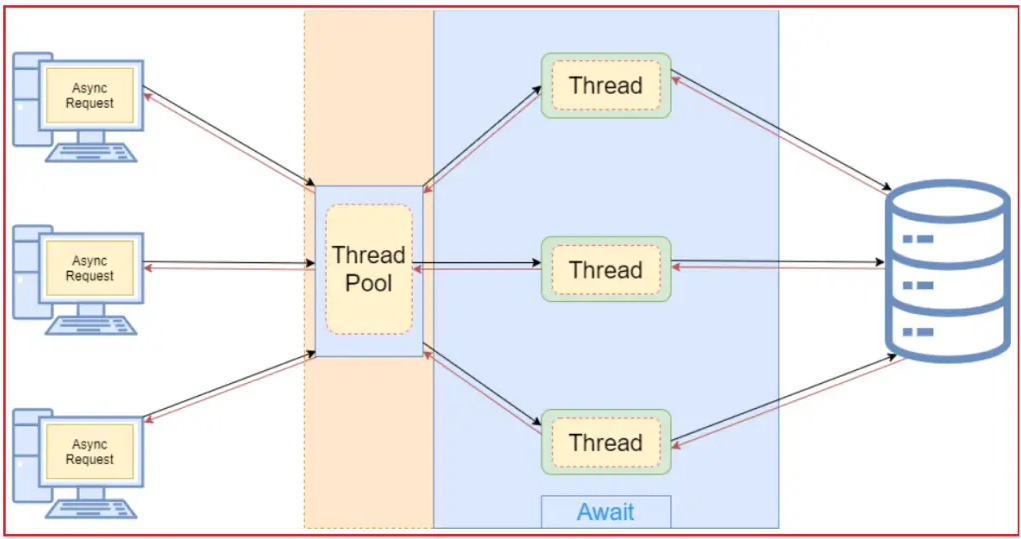
}

}

****

**Asynchronous Programming in C#**

Asynchronous programming in C# is a method of performing tasks without blocking the main or calling thread. This is especially beneficial for I/O-bound operations (like reading from a file, fetching data from the web, or querying a database), where waiting for the task to be completed might waste valuable CPU time that could be better spent doing other work. For a better understanding, please have a look at the following diagram.

****

using System;

using System.Threading.Tasks;

namespace AsynchronousProgramming

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Main Method Started......");

SomeMethod();

Console.WriteLine("Main Method End");

Console.ReadKey();

}

public async static void SomeMethod()

{

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

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

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

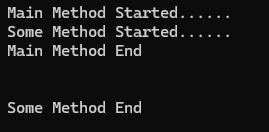
Console.WriteLine("\n");

Console.WriteLine("Some Method End");

}

}

}



**Difference Between Asynchronous Programming VS Multithreading**

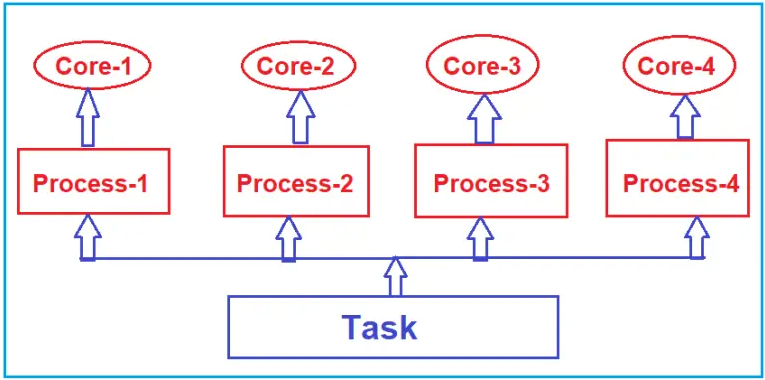
We use asynchronous programming when we have a blocking operation in the program and we want to continue with the execution of the program without waiting for the result. This allows us to implement tasks that can run at the same time.

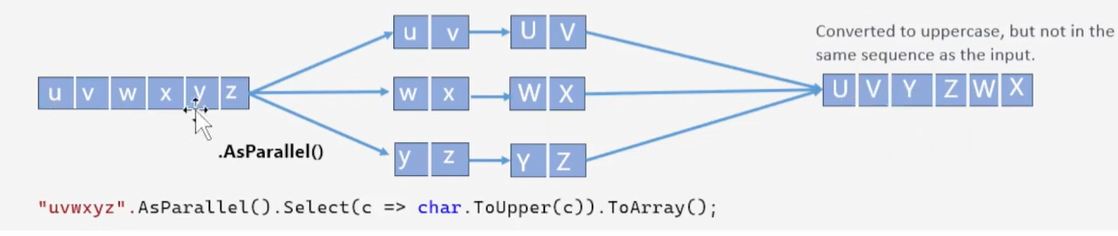
We mainly use multithreading when we want to maximize the multi-core processors to have multiple workers working independently.

**Parallel Programming in C#**

Parallel Programming in C# helps us divide a task into different parts and work those parts simultaneously. An example might be that we have a set of credit cards and we want to process them simultaneously, we can do this by taking the advantage of parallelism.

For a better understanding, please have a look at the below diagram. As you can see, the same task will be executed by multiple processes, multiple cores where each process has multiple threads to execute the application code.

****



In C#, we mainly use two tools to work with parallelism. They are as follows:

* The Task Parallel Library (TPL)
* Parallel LINQ (PLINQ)

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

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

Console.WriteLine(number);

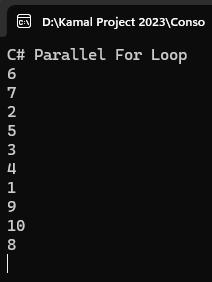
});

Console.ReadLine();

}

}

}

****

**Parallel Invoke in c#**

The Parallel Invoke method in C# is used to launch multiple tasks that are going to be executed in parallel.

using System;

using System.Diagnostics;

using System.Threading;

using System.Threading.Tasks;

namespace ParallelProgrammingDemo

{

public class Program

{

static void Main()

{

Stopwatch stopWatch = new Stopwatch();

stopWatch.Start();

//Calling Three methods sequentially

Method1();

Method2();

Method3();

stopWatch.Stop();

Console.WriteLine($"Sequential Execution Took {stopWatch.ElapsedMilliseconds} Milliseconds");

Console.ReadKey();

}

static void Method1()

{

Thread.Sleep(200);

Console.WriteLine($"Method 1 Completed by Thread={Thread.CurrentThread.ManagedThreadId}");

}

static void Method2()

{

Thread.Sleep(200);

Console.WriteLine($"Method 2 Completed by Thread={Thread.CurrentThread.ManagedThreadId}");

}

static void Method3()

{

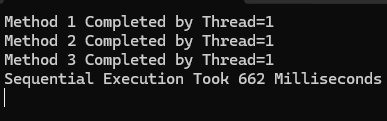
Thread.Sleep(200);

Console.WriteLine($"Method 3 Completed by Thread={Thread.CurrentThread.ManagedThreadId}");

}

}

}

****

using System;

using System.Diagnostics;

using System.Threading;

using System.Threading.Tasks;

namespace ParallelProgrammingDemo

{

public class Program

{

static void Main()

{

Stopwatch stopWatch = new Stopwatch();

stopWatch.Start();

//Calling Three methods Parallely

Parallel.Invoke(

Method1, Method2, Method3

);

stopWatch.Stop();

Console.WriteLine($"Parallel Execution Took {stopWatch.ElapsedMilliseconds} Milliseconds");

Console.ReadKey();

}

static void Method1()

{

Thread.Sleep(200);

Console.WriteLine($"Method 1 Completed by Thread={Thread.CurrentThread.ManagedThreadId}");

}

static void Method2()

{

Thread.Sleep(200);

Console.WriteLine($"Method 2 Completed by Thread={Thread.CurrentThread.ManagedThreadId}");

}

static void Method3()

{

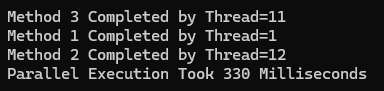
Thread.Sleep(200);

Console.WriteLine($"Method 3 Completed by Thread={Thread.CurrentThread.ManagedThreadId}");

}

}

}

****

**Ref vs Out Keywords in C#**

The REF and OUT keywords in C# are used for passing the arguments to a method as a reference type. By default, arguments are passed to a method by value. By using these REF and OUT keywords in C#, we can pass arguments by reference. In this case, any changes made to these arguments in the method body will be reflected in those variable when the control returns to the calling method.

##### **Differences Between OUT and REF Keyword in C#**

##### So, the first point that you need to remember is when you want multiple outputs from a function, then you need to use the ref and out parameters in C#. If you look out and ref, both are closely doing the same thing. Then what are the differences between them? Let us understand the differences with an example.

using System;

namespace RefvsOutDemo

{

class Program

{

static void Main(string[] args)

{

//Calling the Method with the REF arguments

int AdditionRef = 0;

int SubtractionRef = 0;

MathRef(200, 100, ref AdditionRef, ref SubtractionRef);

Console.WriteLine($"AdditionRef: {AdditionRef}");

Console.WriteLine($"SubtractionRef: {SubtractionRef}");

//Call the Method with the OUT arguments

int AdditionOut = 0;

int SubtractionOut = 0;

MathOut(200, 100, out AdditionOut, out SubtractionOut);

Console.WriteLine($"AdditionOut: {AdditionOut}");

Console.WriteLine($"SubtractionOut: {SubtractionOut}");

Console.ReadKey();

}

//Creating Method with Ref Parameters

public static void MathRef(int number1, int number2, ref int Addition, ref int Subtraction)

{

Addition = number1 + number2; //This will Update the Addition variable inside the Main method

Subtraction = number1 - number2; //This will Update the Subtraction variable inside the Main method

}

//Creating Method with out Parameters

public static void MathOut(int number1, int number2, out int Addition, out int Subtraction)

{

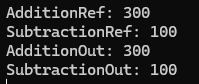
Addition = number1 + number2; //This will Update the Addition variable inside the Main method

Subtraction = number1 - number2; //This will Update the Subtraction variable inside the Main method

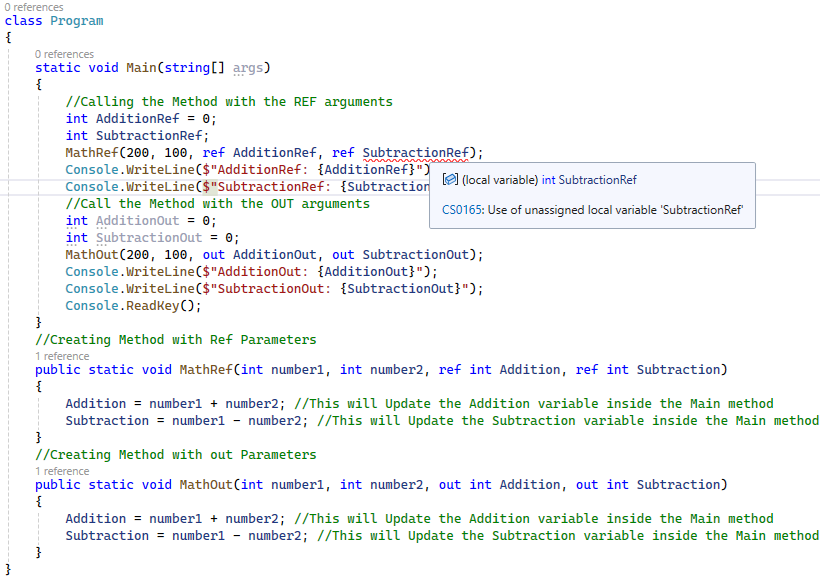
}

}

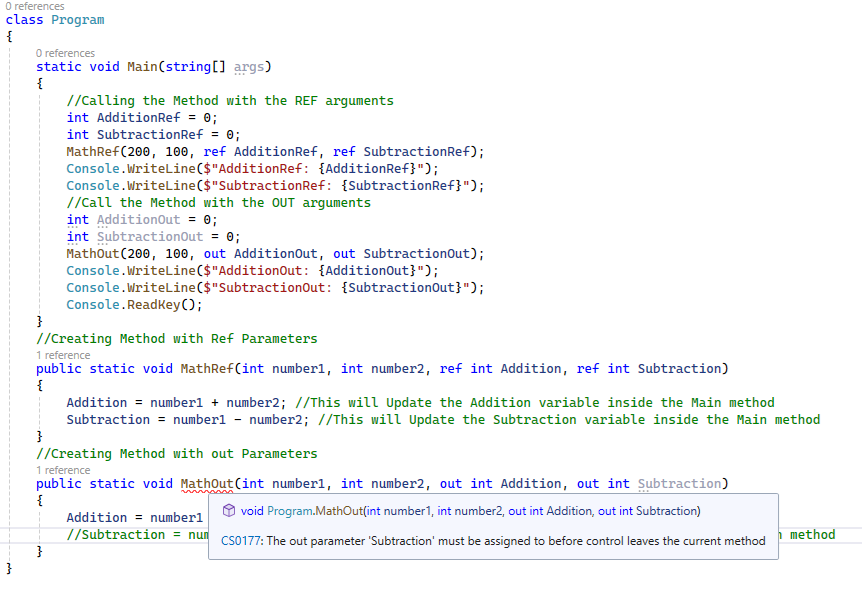
}

****

When we are passing the ref parameter as arguments, it is mandatory to initialize the ref parameter before passing it to the method else we will get compile time error. This is because with the ref parameter, updating the value inside the method is optional. So, before passing the ref parameter, it should be initialized.

****

When we call a method with the “out” variable, the method has to update the out variable inside the function and it is mandatory. But this is not mandatory if you are using the ref variable

****

**Why do we need Tuples in C#?**

If you want to return more than one value from a method then you need to use **Tuples in C#**. Tuples in C# 7, provide a better mechanism to return multiple values from a method.

Tuples Before C# 7:

using System;

using System.Collections.Generic;

namespace TuplesDemo

{

class Program

{

static void Main()

{

var values = new List<double>() { 10, 20, 30, 40, 50 };

//Store the Result of Calulate Method in a variable of Tuple type

Tuple<int, double> t = Calulate(values);

//Access the First value using Item1 and second value using Item2 properties

Console.WriteLine($"There are {t.Item1} values and their sum is {t.Item2}");

Console.ReadKey();

}

//Declaring the return type as Tuple<int, double>

private static Tuple<int, double> Calulate(IEnumerable<double> values)

{

int count = 0;

double sum = 0.0;

foreach (var value in values)

{

count++;

sum += value;

}

//Creating an object of Tuple class by calling the static Create method

Tuple<int, double> t = Tuple.Create(count, sum);

//Returning the tuple instance

return t;

}

}

}



Tuples in C# 7

using System;

using System.Collections.Generic;

namespace TulesDemo

{

class Program

{

static void Main()

{

var values = new List<double>() { 10, 20, 30, 40, 50 };

var result = Calulate(values);

Console.WriteLine($"There are {result.count} values and their sum is {result.sum}");

Console.ReadKey();

}

private static (int count, double sum) Calulate(IEnumerable<double> values)

{

int count = 0;

double sum = 0.0;

foreach (var value in values)

{

count++;

sum += value;

}

return (count, sum);

}

}

}



**Covariance and Contravariance**

Covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

// Assignment compatibility.

string str = "test";

// An object of a more derived type is assigned to an object of a less derived type.

object obj = str;

// Covariance.

IEnumerable<string> strings = new List<string>();

// An object that is instantiated with a more derived type argument

// is assigned to an object instantiated with a less derived type argument.

// Assignment compatibility is preserved.

IEnumerable<object> objects = strings;

// Contravariance.

// Assume that the following method is in the class:

static void SetObject(object o) { }

Action<object> actObject = SetObject;

// An object that is instantiated with a less derived type argument

// is assigned to an object instantiated with a more derived type argument.

// Assignment compatibility is reversed.

Action<string> actString = actObject;

**Preprocessor directives**

C# contains two preprocessor directives, #region and #endregion, that are useful only within the context of visual code editors. Code editors, such as Microsoft Visual Studio, can search through source code and find these directives to provide editor features when writing code.

**What’s meaning of Mutable and Immutable in English?**

Answers are.

* **Mutable:** Can change
* **Immutable:** Cannot Change

In simple words we can say, immutable means object cannot be changed once it is created. If you want to change, you have to create new object/ assign new memory. Example of immutable type is string in C#.

**Record types in C#**

How to create Record Type?

Let’s try to implement Record Type in the Member class. See the below code.

public record Member

{

public int Id { get; init; }

public string FirstName { get; init; }

public string LastName { get; init; }

public string Address { get; init; }

}

The only change is we have replaced the “Class” keyword with “Record”. Now Member record type is treated as an immutable data value.

**With expression**

C# 9.0 introduces the “**With**” expression with RecordType. It is mainly used to create new objects more effectively.

using System;

using System.Collections.Generic;

namespace TulesDemo

{

public record Member

{

public int Id { get; init; }

public string FirstName { get; init; }

public string LastName { get; init; }

public string Address { get; init; }

}

class Program

{

static void Main()

{

var member = new Member

{

Id = 1,

FirstName = "Kirtesh",

LastName = "Shah",

Address = "Vadodara"

};

var newMember = member with { Address = "Mumbai" };

//The “With” expression is used as a {} syntax that allows you to define new values for a specific property. It will make development faster, easier, and cleaner.

Console.WriteLine("ID : -" + newMember.Id);

Console.WriteLine("First Name :- " + newMember.FirstName);

Console.WriteLine("Last Name :- " + newMember.LastName);

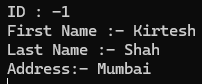
Console.WriteLine("Address:- " + newMember.Address);

Console.ReadLine();

}

}

}

****

**ADO.Net**

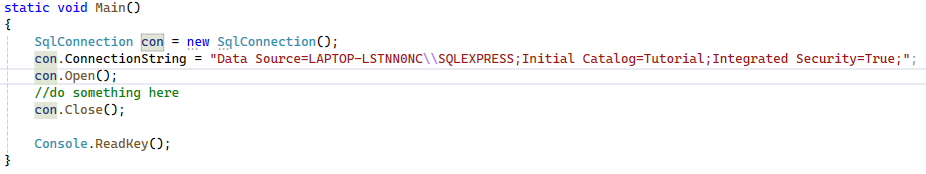
**ADO stands for Microsoft ActiveX Data Objects.** ADO.NET is one of Microsoft’s Data Access Technologies, which we can use to communicate with different data sources. It is a part of the .NET Framework, which is used to establish a connection between the .NET Application and different data sources. The **Data** Sources can be SQL Server, Oracle, MySQL, XML, etc. ADO.NET mainly uses **System.Data.dll** and **System.Xml.dll.**

###### **Key Components and Concepts in ADO.NET Include:**

* **Connection**
* **Command**
* **DataReader**
* **DataAdapter**
* **DataSet**
* **DataTable**
* **DataView**
* **Transaction**

**SqlConnection Class in C#?**

The ADO.NET SqlConnection class belongs to System.Data.SqlClient namespace is used to establish an open connection to the SQL Server database. The most important point you must remember is that the connection does not close implicitly, even if it goes out of scope. Therefore, it is always recommended and always a good programming practice to close the connection object explicitly by calling the Close() method of the connection object.



**SqlCommand Class in C#?**

The ADO.NET SqlCommand class in C# stores and executes the SQL statement against the SQL Server database.

**Methods of SqlCommand Class in C#**

The SqlCommand class in C# provides the following methods.

* **BeginExecuteNonQuery():** This method initiates the asynchronous execution of the Transact-SQL statement or stored procedure.
* **Cancel():** This method tries to cancel the execution of a Command.
* **Clone():** This method creates a new Command object is a copy of the current instance.
* **CreateParameter():** This method creates a new instance of a System.Data.SqlClient.SqlParameter object.
* **ExecuteReader():** Sends the [CommandText](https://learn.microsoft.com/en-us/dotnet/api/microsoft.data.sqlclient.sqlcommand.commandtext?view=sqlclient-dotnet-standard-5.1" \l "microsoft-data-sqlclient-sqlcommand-commandtext) to the [Connection](https://learn.microsoft.com/en-us/dotnet/api/microsoft.data.sqlclient.sqlcommand.connection?view=sqlclient-dotnet-standard-5.1#microsoft-data-sqlclient-sqlcommand-connection) and builds a [SqlDataReader](https://learn.microsoft.com/en-us/dotnet/api/microsoft.data.sqlclient.sqldatareader?view=sqlclient-dotnet-standard-5.1).
* **ExecuteScalar():** This method Executes the query and returns the first column of the first row in the result set returned by the query. Additional columns or rows are ignored.
* **ExecuteNonQuery():** This method executes a Transact-SQL statement against the connection and returns the number of rows affected.
* **Prepare():** This method creates a prepared version of the command on an instance of SQL Server.
* **ResetCommandTimeout():** This method resets the CommandTimeout property to its default value.



##### ****SqlDataReader Class in C#?****

The ADO.NET SqlDataReader class in C# is used to read data from the SQL Server database in the most efficient manner. It reads data in the forward-only direction. It means once it reads a record, it will then read the next record; there is no way to go back and read the previous record.

##### ****ADO.NET SqlDataAdapter in C#?****

The ADO.NET SqlDataAdapter in C# bridges a DataSet or DataTable and a Data Source (SQL Server Database) to retrieve data. The SqlDataAdapter is a class that represents a set of SQL commands and a database connection. It is used to fill the DataSet or DataTable and update the data source as well.



**Pass method name as a parameter in c#**

Func<int, string> is a delegate which represents a function taking an int argument and returning a string.

Func<string, int> is a delegate which represents a function taking a string argument and returning an int.

using System;

namespace PassedMethodAsParameter

{

class Program

{

static void Main(string[] args)

{

int i=RunTheMethod(Method1);

Console.WriteLine("Method {0} Executed",i);

int j = RunTheMethod(Method2);

Console.WriteLine("Method {0} Executed", j);

Console.ReadLine();

}

public static int Method1(string input)

{

return 1;

}

public static int Method2(string input)

{

return 2;

}

public static int RunTheMethod(Func<string, int> myMethodName)

{

int i = myMethodName("My String");

return i;

}

}

}



**Optional Parameter in c#**

using System;

namespace OptionalParameter

{

class Program

{

static void Main(string[] args)

{

ADDNumbers(10, 20);

ADDNumbers(10, 20, new int[] { 30, 40, 50 });

Console.ReadLine();

}

public static void ADDNumbers(int FN, int SN, int[] restOfTheNumbers = null)

{

int result = FN + SN;

if (restOfTheNumbers != null)

{

foreach (int i in restOfTheNumbers)

{

result += i;

}

}

Console.WriteLine("Total = " + result.ToString());

}

}

}

**C# Programs**

Sum of Digit 12345 => 15

public static void Main(string[] args)

{

int n, sum = 0, m;

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

n = int.Parse(Console.ReadLine());

while (n > 0)

{

m = n % 10;

sum = sum + m;

n = n / 10;

}

Console.Write("Sum is= " + sum);

}

Reverse Digit Program 123456 => 654321

public static void Main()

{

int num, reverse = 0;

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

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

while (num != 0)

{

reverse = reverse \* 10;

reverse = reverse + num % 10;

num = num / 10;

}

Console.WriteLine("Reverse of Entered Number is : " + reverse);

Console.ReadLine();

}

Fibonacci Series

public static void Main(string[] args)

{

int a = 0, b = 1;

int nextTerm = a + b;

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

int n = int.Parse(Console.ReadLine());

Console.WriteLine(a);

Console.WriteLine(b);

for (int i = 3; i <= n; i++)

{

Console.WriteLine(nextTerm);

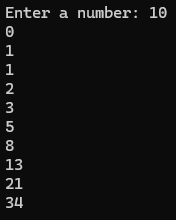
a = b;

b = nextTerm;

nextTerm = a + b;

}

}



Check Number is prime or not

public static void Main()

{

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

int n = int.Parse(Console.ReadLine());

int a = 0;

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

{

if (n % i == 0)

{

a++;

}

}

if (a == 2)

{

Console.WriteLine("{0} is a Prime Number", n);

}

else

{

Console.WriteLine("Not a Prime Number");

}

Console.ReadLine();

}

Prime Number Series

public static void Main()

{

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

int n = int.Parse(Console.ReadLine());

bool prime;

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

{

prime = true;

for (int j = 2; j < i; j++)

{

if (i % j == 0)

{

prime = false;

break; // Exiting the loop if a factor is found

}

}

if (prime == true)

{

Console.WriteLine(i);

}

}

Console.ReadLine();

}

Reverse Sting Kamal >= lamaK

public static void Main()

{

string Input;

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

Input = Console.ReadLine();

// Converting string to character array

char[] charArray = Input.ToCharArray();

string reversedString = String.Empty;

for (int i = charArray.Length - 1; i >= 0; i--)

{

// Append each character to the reversedstring.

reversedString += charArray[i];

}

Console.WriteLine(reversedString);

Console.ReadLine();

}

Count characters in string

public static void Main()

{

string str = "ssmmmjjkkkkrrr";

Dictionary<char, int> counts = new Dictionary<char, int>();

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

{

if (counts.ContainsKey(str[i]))

{

counts[str[i]]++;

}

else

{

counts.Add(str[i], 1);

}

}

foreach (var count in counts)

Console.WriteLine("{0} = {1}", count.Key, count.Value.ToString());

Console.ReadLine();

}



Bubble SORT

static void Main(string[] args)

{

int[] number = { 89, 76, 45, 92, 67, 12, 99 };

int temp;

int numLength = number.Length;

//sorting an array

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

{

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

{

if (number[j] > number[j+1])

{

temp = number[j];

number[j] = number[j + 1];

number[j + 1] = temp;

}

}

}

//Sorted array

foreach (int num in number)

{

Console.Write("\t {0}", num);

}

Console.Read();

}

**Remove Duplicate Value From Array**

static void Main(string[] args)

{

int[] arr = { 1, 2, 2, 6, 6,3,6,5,5,5,88,8,8,88,88,88 ,6, 6, 6, 3, 4, 4, 4, 5, 5 };

int size = arr.Length;

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

{

for (int j = i + 1; j < size; j++)

{

if (arr[i] == arr[j])

{

for (int k = j; k < size - 1; k++)

{

arr[k] = arr[k + 1];

}

size--;

j--;

}

}

}

for (int l = 0; l < size; l++)

{

Console.WriteLine(arr[l]);

}

Console.ReadKey();

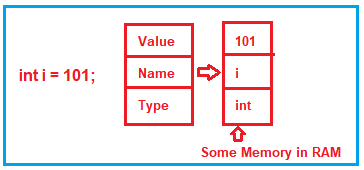
}

**Stack and Heap Memory in .NET**

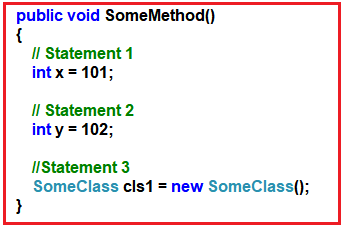
**What Happens Internally When We Declare a Variable in a .NET Application?**

When we declare a variable in a .NET application, it allocates some memory in the RAM. The memory that it allocates in RAM has three things are as follows:

1. **Name of the Variable,**
2. **The Data Type of the Variable, and**
3. **Value of the Variable.**

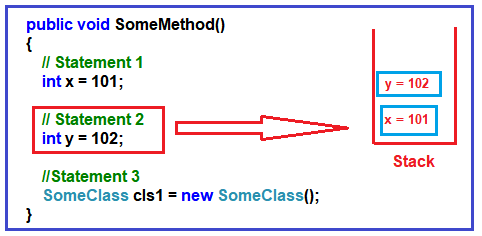


There are two types of memory allocation for the variables we created in the .NET Application, i.e., Stack Memory and Heap Memory. Let us understand the Stack and Heap Memory with an Example.

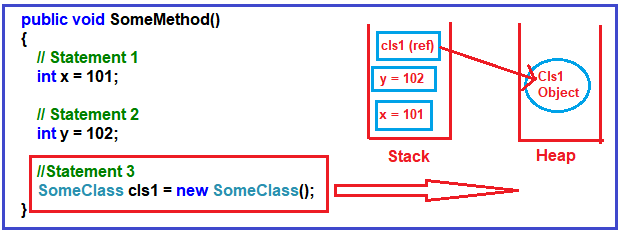


As you can see in the above image, the SomeMethod has three statements. Let’s understand statement by statement how things are executed internally.

When the second statement is executed, it stacks this memory allocation (memory allocation for variable y) on top of the first memory allocation (memory allocation for variable x). You can think about the stack as a series of plates or dishes put on top of each other.



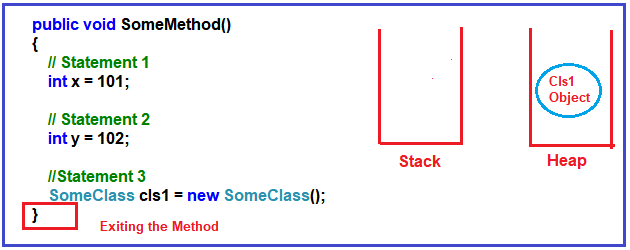
In the 3rd statement, we have created an object of SomeClass. When the 3rd statement is executed, it internally creates a pointer on the stack memory, and the actual object is stored in a different memory location called Heap memory. The Heap Memory location does not track running memory. Heap is used for dynamic memory allocation.



**Note:** The reference pointers are allocated on the stack. The statement, **SomeClass cls1,** does not allocate any memory for an instance of **SomeClass.**It only allocates a variable with the name cls1 in the stack and sets its value to null. When it hits the new keyword, it allocates memory in the heap.

**What Happens When the Method Completes Its Execution?**

When the three statements are executed, the control will exit from the method. When it passes the end control, i.e., the end curly brace “},” it will clear all the memory variables created on the stack. It will de-allocate the memory from the stack in a ‘LIFO’ fashion. For a better understanding, please have a look at the below image.

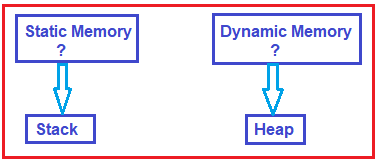


It will not de-allocate the Heap memory. Later, the heap memory will be de-allocated by the garbage collector. Now, you may have one question in your mind: why two types of memory? Can’t we allocate everything to just one memory type?

**Why do we have two types of memory?**

In C#, primitive data types, such as int, double, bool, etc., hold a single value. On the other hand, the reference data types or object data types are complex, i.e., an object data type or reference data type can have reference to other objects and other primitive data types.

So, the reference data type holds references to other multiple values, and each one of them must be stored in memory. Object types need dynamic memory, while primitive data types need static memory. Please have a look at the following image for a better understanding.

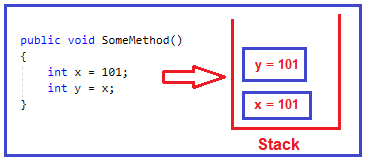


**Value Types and Reference Types in C#.NET**

As we understood the concept of Stack and Heap, Now, let us move forward and understand the concept value types and reference types in C#. The Value types are the types that hold both data and memory in the same location. On the other hand, a reference type is a type that has a pointer that points to the actual memory location.

**Understanding Value Type in C#:**

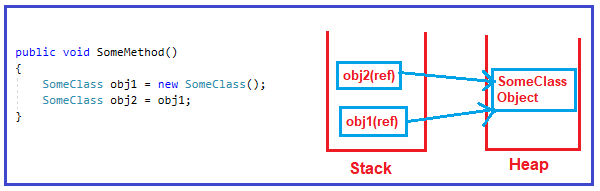
Let us understand value type with an example. Please have a look at the following image. As you can see in the image, first, we create an integer variable with the name x, and then we assign this x integer value to another integer variable named y. In this case, the memory allocation for these two variables will be done inside the stack memory.



In .NET, when we assign one integer variable value to another integer variable, it creates a completely different copy in the stack memory. That’s what you can see in the above image. So, if you change one variable value, the other variable will not be affected. In .NET, these data types are called Value types. So, bool, byte, char, decimal, double, enum, float, long, sbyte, int, short, ulong, struct, uint, ushort are examples of value types.

**Understanding Reference Type in C#:**

Let us understand reference type with an example. Please have a look at the following image. Here, first, we create an object, i.e., obj1) and then assign this object to another object, i.e., obj2. In this case, both reference variables (obj1 and obj2) will point to the same memory location.



In this case, when you change one of them, the other object is also affected. These kinds of data types are termed Reference types in .NET. So, class, interface, object, string, and delegate are examples of Reference Types.

**Convert.ToString and ToString Method in C#**

Both these methods are used to convert a value to a string. The difference is **Convert.ToString()** method handles null whereas the **ToString()** doesn’t handle null in C#.

**String and String Builder in C#**

**String**

A String instance is immutable, which means, we cannot change it after it was created. If we perform any operation on a String it will return a new instance (creates a new instance in memory) instead of modifying the existing instance value.

**StringBuilder**

StringBuilder is mutable, that is, if we perform any operation on StringBuilder it will update the existing instance value and it will not create a new instance.

**What is Garbage Collection in .NET Framework?**

When a dot net application runs, lots of objects are created. At a given point in time, it is possible that the application does not use some of those objects. The Garbage Collector in the .NET Framework is nothing but a Small Routine, or you can say it’s a Background Process Thread that runs periodically and tries to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

So, Garbage Collector is nothing but a feature provided by CLR that helps us clean or destroy unused managed objects. Cleaning or destroying those unused managed objects basically reclaims the memory.

Garbage Collection (GC) in the .NET Framework is an automatic memory management system that helps manage the allocation and release of memory in your applications. In .NET, when we create an object using the new keyword, it automatically allocates memory on the managed heap.

**Note:** The Garbage Collector will destroy only the unused managed objects. It does not clean unmanaged objects.

**Managed and Unmanaged Objects in .NET Framework**

Whenever we create any EXE (i.e., Console Application, Windows Application, etc.) or Web Application (i.e., ASP.NET MVC, Web API, ASP.NET, Class Library, etc.) in .NET Framework using Visual Studio and using any .NET supported programming language such as C#, VB, F#, etc. These applications are run completely under the control of CLR (Common Language Runtime). That means if your applications have unused objects, then CLR will clean those objects using Garbage Collector.

On the other hand, Skype, PowerPoint, and Microsoft Excel do not require dot net runtime. They run under their own environment. So, in short, the code (EXE, Web App) that is not run under the control of CLR is called unmanaged code. CLR will not provide any facilities and features of .NET to the unmanaged code in C# execution like Language Interoperability, Automatic memory management, Exception handling mechanism, code access security, etc.

**Managed Objects**

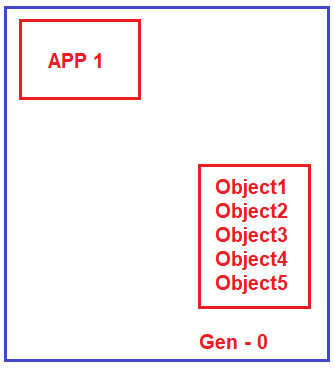
Managed objects are allocated on the managed heap and controlled by the .NET Garbage Collector (GC). These objects are typically instances of classes and structures defined in .NET. The GC automatically manages the memory for managed objects.

**Unmanaged Objects**

Unmanaged objects are objects whose memory is not managed by the .NET GC. These are typically objects allocated using native code, like calls to Windows API or using languages such as C or C++. The developer is responsible for allocating and freeing the memory for unmanaged objects. Examples of unmanaged objects are file handles, database connections, COM objects, or any other resources that are not managed by the .NET runtime.

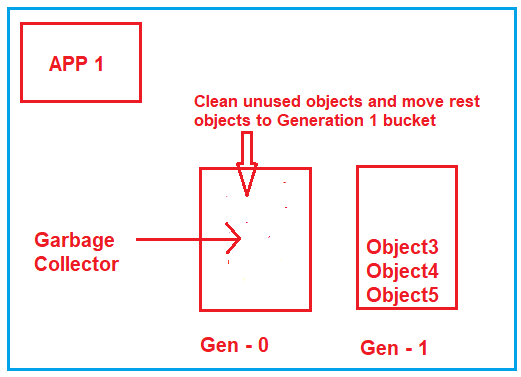
**Understanding Generation 0, 1, and 2:**

Let’s say you have a simple application called App1. As soon as the application starts, it creates 5 managed objects. Whenever any new objects (fresh objects) are created, they are moved into a bucket called Generation 0. For a better understanding, please have a look at the following image.



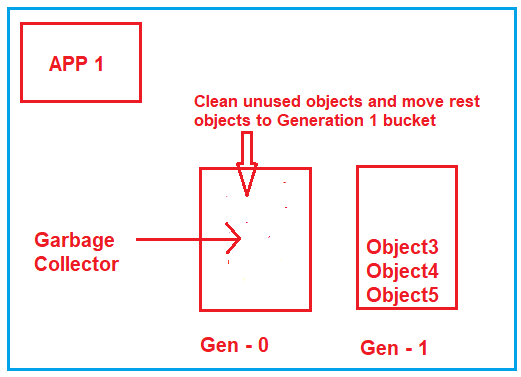
We know Garbage Collector, runs continuously as a background process thread to check whether there are any unused managed objects so that it reclaims the memory by cleaning those objects. Now, let’s say the application does not need two objects (Object1 and Object2). So, the Garbage Collector will destroy these two objects (Object1 and Object2) and reclaim the memory from the Generation 0 bucket. But the application still needs the remaining three objects (Object3, Object4, and Object5). So, the Garbage collector will not clean those three objects. The Garbage Collector will move those three managed objects (Object3, Object4, and Object5) to the Generation 1 bucket, as shown in the image below.

three managed objects (Object3, Object4, and Object5) to the Generation 1 bucket, as shown in the image below.

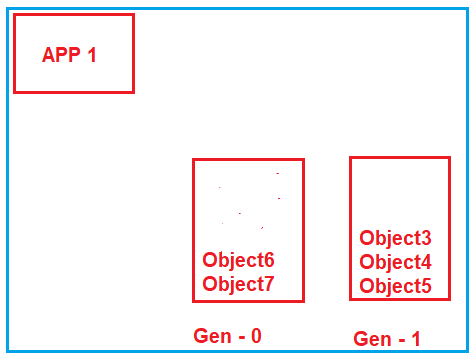


Let’s say your application creates two more fresh objects (Object6 and Object7). As fresh objects, they should be created in the Generation 0 bucket, as shown in the image below.

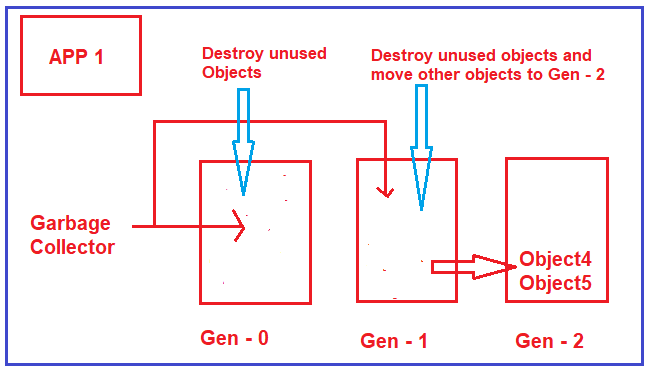
three managed objects (Object3, Object4, and Object5) to the Generation 1 bucket, as shown in the image below.



Let’s say your application creates two more fresh objects (Object6 and Object7). As fresh objects, they should be created in the Generation 0 bucket, as shown in the image below.



Now, again, the Garbage Collector runs, and it comes to the Generation 0 bucket and checks which objects are used. Let’s say both objects (Object6 and Object7) are unused by the application, so it will remove both objects and reclaim the memory. Now, it goes to the Generation 1 bucket and checks which objects are unused. Let’s say the application still needs Object4 and Object5 while object3 is not needed. So, what Garbage Collector will do is destroy Object3, reclaim the memory, and move Object4 and Object5 to the Generation 2 bucket, as shown in the image below.



**Why do we need Enums in C#?**

The Enums are Strongly Typed Name Constants. Let us understand Enums i.e. what do you mean by Strongly Typed Name Constants with an Example? Let us assume we have an Employee class with the Name and Gender Properties. Gender is an Integer Property and the values of this property represent the following.

1. 0 is an Unknown Gender
2. 1 is Male
3. 2 is Female

That means if the Value of the Gender Property is 1, then its meaning is Gender is Male, similarly, if the value is 2, then its meaning is Female, and for 0, the meaning is Gender is Unknown. For a better understanding, please have a look at the below example.

using System;

using System.Collections.Generic;

namespace EnumsDemo

{

class Program

{

static void Main(string[] args)

{

List<Employee> empList = new List<Employee>

{

new Employee() { Name = "Anurag", Gender = 0 },

new Employee() { Name = "Pranaya", Gender = 1 },

new Employee() { Name = "Priyanka", Gender = 2 },

new Employee() { Name = "Sambit", Gender = 3 }

};

//Loop through Each Employees and Print the Name and Gender

foreach (var emp in empList)

{

//To Print the Actual Gender of the Employee,

//we need to call the GetGender Method by passing the Integer Gender Value

Console.WriteLine($"Name = {emp.Name} && Gender = {GetGender(emp.Gender)}");

}

Console.ReadLine();

}

public static string GetGender(int gender)

{

switch (gender)

{

case (int)Gender.Unknown:

return "Unknown";

case (int)Gender.Male:

return "Male";

case (int)Gender.Female:

return "Female";

default:

return "Invalid Data for Gender";

}

}

}

public class Employee

{

public string Name { get; set; }

public int Gender { get; set; }

}

public enum Gender

{

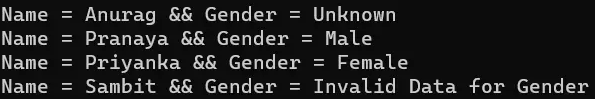
Unknown,

Male,

Female

}

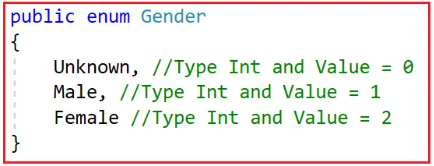
}



Note: The default value for the first element of the enum is ZERO and gets incremented by 1.

**What is the Default Type of Enum Members in C#?**

The default underlying type of enum is int and the value starts at ZERO and is incremented by 1 for each enum member. For example, in the below Enum, the type of Unknown, Male, and Female, and the value for Unknown is 0, Male is 1, and Female is 2.



**What is a Property in C#?**

A Property in C# is a member of a class that is used to set and get the data from a data field (i.e. variable) of a class. The most important point that you need to remember is that a property in C# is never used to store any data, it just acts as an interface or medium to transfer the data. We use the Properties as they are public data members of a class, but they are actually special methods called accessors.

**What are Accessors in C#?**

The Assessors are nothing but special methods which are used to set and get the values from the underlying data member (i.e. variable) of a class. Assessors are of two types. They are as follows:

1. **Set Accessor**
2. **Get Accessor**

using System;

namespace PropertyDemo

{

public class Employee

{

//Private Data Members

private int \_EmpId;

private string \_EmpName;

private int \_Salary;

//Public Properties

public int EmpId

{

set

{

\_EmpId = value;

}

get

{

return \_EmpId;

}

}

public string EmpName

{

set

{

\_EmpName = value;

}

get

{

return \_EmpName;

}

}

public int Salary

{

set

{

\_Salary = value;

}

get

{

return \_Salary;

}

}

}

class Program

{

static void Main(string[] args)

{

Employee employee = new Employee();

//We cannot access the private data members

//So, using public properties (SET Accessor) we are setting

//the values of private data members

employee.EmpId = 101;

employee.EmpName = "Pranaya";

employee.Salary = 10000;

//Using public properties (Get Accessor) we are Getting

//the values of private data members

Console.WriteLine("Employee Details:");

Console.WriteLine("Employee id:" + employee.EmpId);

Console.WriteLine("Employee name:" + employee.EmpName);

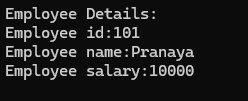
Console.WriteLine("Employee salary:" + employee.Salary);

Console.ReadKey();

}

}

}



**Why do we need Properties in C# Real-time Applications?**

Declaring the class variables as public and exposing those variables to the outside world (which means outside of the class) is bad as we do not have any control over what gets assigned and what gets returned.

**Problems with the above public fields are as follows**

1. An EmpId value should always be a non-negative number.
2. The EmpName cannot be set to NULL.
3. If a employee name is missing then we should return “No Name”.
4. The Salary value should always be read-only.

using System;

using System.Security.Cryptography;

using System.Xml.Linq;

namespace PropertyDemo

{

public class Employee

{

//Private Data Members

private int \_EmpId;

private string \_EmpName;

private int \_Salary=100;

public void SetEmpId(int EmpId)

{

if (EmpId < 0)

{

throw new Exception("ID value should be greater than zero");

}

\_EmpId = EmpId;

}

public int GetEmpId()

{

return \_EmpId;

}

public void SetEmpName(string EmpName)

{

if (string.IsNullOrEmpty(EmpName))

{

throw new Exception("Name should not be empty");

}

\_EmpName = EmpName;

}

public string GetEmpName()

{

if (string.IsNullOrEmpty(\_EmpName))

{

return "No Name";

}

return \_EmpName;

}

public int GetSalary()

{

return \_Salary;

}

}

class Program

{

static void Main(string[] args)

{

Employee employee = new Employee();

//We cannot access the private data members

//So, using public properties (SET Accessor) we are setting

//the values of private data members

employee.SetEmpId(-101);

employee.SetEmpName("Pranaya");

//Using public properties (Get Accessor) we are Getting

//the values of private data members

Console.WriteLine("Employee Details:");

Console.WriteLine("Employee id:" + employee.GetEmpId());

Console.WriteLine("Employee name:" + employee.GetEmpName());

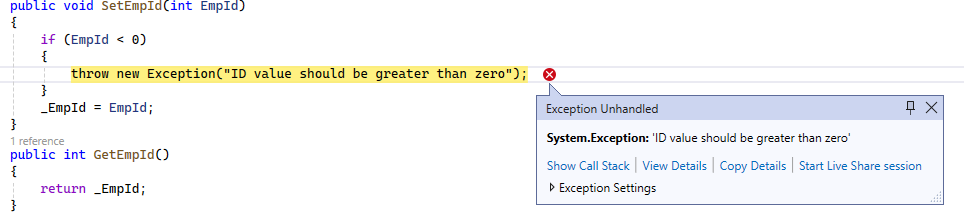
Console.WriteLine("Employee salary:" + employee.GetSalary());

Console.ReadKey();

}

}

}



**Dispose Method**

The Dispose method is part of the IDisposable interface and it is implemented to release both managed and unmanaged resources deterministically. Unlike Finalize, Dispose is called explicitly in your code, usually when you are finish using an object. This allows for the immediate freeing of resources.

There are a couple of resources that GC is not able to release as it doesn't have information, on how to claim memory from those resources like **File handlers**, **window handlers, network sockets, database connections**, etc. If your application has these resources then it's the program's responsibility to release unmanaged resources.

**For example**, if we open a file in our program and not close it after processing then that file will not be available for other operations or if it is being used by other applications they cannot open or modify that file. For this purpose, the FileStream class provides the Dispose method. We must call this method after file processing is finished. Otherwise, it will be through exception **Access Denied or the file is being used by another program**.

**Dispose Vs Close**

For database connections, there's a small difference between both.  
Dispose will release the database connection completely, so it can't be reopened.  
Close will close the connection but will keep the connection string, connection timeout etc intact, so you can reopen it.  
  
But there's no benefit in closing then disposing, as dispose already does the close.