**Namespace**

* Namespaces in C# are used to organize too many classes so that it can be easy to handle the application.
* In simpler words you can say that it provides a way to keep one set of names(like class names) different from other sets of names
* The biggest advantage of using namespace is that the class names which are declared in one namespace will not clash with the same class names declared in another namespace
* Two classes with the same name can be created inside 2 different namespaces in a single program
* It is not necessary to keep each class in C# within Namespace but we do it to organize our code well.
* we can say that **Namespace** is a collection of [classes](https://www.tutlane.com/tutorial/csharp/csharp-classes-and-objects-with-examples) and [classes](https://www.tutlane.com/tutorial/csharp/csharp-classes-and-objects-with-examples) are the collection of objects and [methods](https://www.tutlane.com/tutorial/csharp/csharp-methods-functions-with-examples). Therefore, by using namespaces we can easily access all the class [methods](https://www.tutlane.com/tutorial/csharp/csharp-methods-functions-with-examples) just by importing the namespace in our application.
* Let us understand the concept of namespace with a real life scenario. We have a large number of files and folders in our computer. Imagine how difficult it would be to manage them if they are placed in a single directory. This is why we put related files and folders in a separate directory. This helps us to manage our data properly.
* The concept of namespace is similar in C#. It helps us to organize different members by putting related members in the same namespace.
* Namespace also solves the problem of naming conflict. Two or more classes when put into different namespaces can have same name

**Datatypes**

* C# is a strongly typed language
* We must declare the type of a variable which indicates the kind of values it is going to store such as integer, float, decimal, text, etc
* string stringVar = "Hello World!!";

int intVar = 100;

float floatVar = 10.2f;

char charVar = 'A';

bool boolVar = true;

* C# mainly categorized data types in two types: Value types and Reference types.
* Value types include simple types (e.g. int, float, bool, and char), enum types, struct types, and Nullable value types. Reference types include class types, interface types, delegate types, and array types
* In c#, a [data type](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples) is a **Value Type** if it hold the value of [variable](https://www.tutlane.com/tutorial/csharp/csharp-variables-with-examples) directly on its own memory space and Value Types will use **Stack** memory to store the variables values.
* For example, if we define and assign a value to the variable like int x = 123; then the system will use the same memory space of variable ‘**x**’ to store the value ‘**123**’
* **Reference Types** will contain a pointer, which points to other memory location that holds the data. The **Reference Types** will not store the variable value directly in its memory instead, it will store the memory address of the variable value to indicate where the value is being stored

**Class**

* It is a collection of objects
* That contains variables for storing data and functions to perform operations on the data.
* A class will not occupy any memory space
* Hence, it is only a logical representation of data.
* To create a class, you simply use the keyword "class" followed by the class name:
* Class Employee   
  {   
  }
* Class is important feature. Without class cannot develop application

**Object Creation and Instantiation**

* An object is an instance(example) of a class
* When an object is created using the new operator, memory is allocated for the class in the heap, the object is called an instance and its starting address will be stored in the object in stack memory.
* When an object is created without the new operator, memory will not be allocated in the heap, in other words an instance will not be created
* Instantiation means defining or creating new object for class to access all properties like methods, operators, fields, etc. from class
* Employee objEmp = new Employee();

**Methods and functions**

* Method is a collection of statements that perform some specific task and return the result to the caller.
* A method can also perform some specific task without returning anything



**Difference between methods and functions**

Basically 'method' and 'function' are same.Method is the correct term for a function associated with a class .but developer change their convenient method and function as in c# works that where as we mention "void" that is a method . function is a without void and return something .

Method:

public void mypgm()

{

//code

}

Function:

public string mypgm()

{

//code

return string

}

**Exception handling & types of exception**

* Exception Handling in C# is *a process to handle runtime errors*.
* We perform exception handling so that normal flow of the application can be maintained even after runtime errors
* All exceptions the derived from *System. Exception* class.
* It is a runtime error, which can be handled. If we do not handle the exception, it prints exception message and terminates the program.

try

{

// code that may raise exceptions

}

catch(Exception ex)

{

// handle exception

}

finally

{

// final cleanup code

}

* If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block
* If there is no exception occurred inside the try block, the control directly transfers to finally block. We can say that the statements inside the finally block is executed always

public static void Main()

    {

        int x = 0;

        int div = 0;

        try

        {

            div = 100 / x;

            Console.WriteLine("This linein not executed");

        }

        catch (DivideByZeroException)

        {

            Console.WriteLine("Exception occured");

        }

        Console.WriteLine($"Result is {div}");

    }

* it is possible to throw an exception programmatically. The 'throw' keyword is used for this purpose
* For example, the following statement throws an ArgumentException explicitly.

public static void Main()

    {

        try

        {

            throw new DivideByZeroException("Invalid Division");

        }

        catch (DivideByZeroException)

        {

            Console.WriteLine("Exception");

        }

        Console.WriteLine("LAST STATEMENT");

    }

* System.DivideByZeroException - handles the error generated by dividing a number with zero
* System.NullReferenceException - handles the error generated by referencing the null object.
* System.InvalidCastException- handles the error generated by invalid typecasting.
* System.IndexOutOfRangeException- Handles errors generated when a method refers to an array index out of range
* System.OutOfMemoryException - handles errors generated from insufficient free memory

**Finally block**

* The finally block will be executed after the try and catch blocks
* We can say that the statements inside the finally block is executed always
* The main purpose of finally block is to release the system resources and close database connection, close the file which is opened in try block
* Instead of placing the same close() method call statements in multiple places if we write it in finally block it is always executed irrespective of the exception raised or not raised.

**Access Modifiers**

* Access modifiers are used to implement encapsulation of OOP
* Access modifiers allow you to define who does or who doesn't have access to certain features.
* There are 4 access modifiers (public, protected, internal, private) which defines the 6 accessibility levels as follows:
* public
* protected
* internal
* protected internal
* private
* private protected

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

**Static & non-static members**

* To create a static member (class, variable, methods, constructor), precede its declaration with the keyword *static*.
* When a member is declared static, it can be accessed with the name of its class directly.
* A static class can only contain static data members, static methods, and a static constructor.
* It is not allowed to create objects of the static class.
* Static classes are [sealed](https://www.geeksforgeeks.org/c-sealed-class/), means one cannot inherit a static class from another class.
* Whenever you write a function or declare a variable, it doesn’t create instance in a memory until you create object of class. But if you declare any function or variable with static modifier, it directly create instance in a memory and acts globally. The static modifier doesn’t reference with any object
* Static members binding with class. So it can access with class
* Non-static members binded with object so it will access with object
* To access a particular class from anywhere inside the solution and no need to create object to access it just use it directly
* You can take example of Built in Console Class of C#

using System;

static class Tutorial {

    public static string Topic = "Static class";

}

public class GFG {

    // Main Method

    static public void Main()

    {

 Console.WriteLine("Topic name is : {0} ", Tutorial.Topic);

    }

}

using System;

class Nparks {

    static public int t = 104;

    public static void total()

    {

        Console.WriteLine("Total number of national parks"+

                           " present in India is :{0}", t);

    }

}

public class GFG {

    // Main Method

    static public void Main()

    {

        // Accessing the static method

        // using its class name

        Nparks.total();

    }

}

**Static VS Non-Static Members **

* Non-static (“regular”) classes can be instantiated.
* Static classes cannot be instantiated.
* Non-static classes can have instance methods and static methods.
* Static classes can only have static methods.
* Instance methods must be called on the instances of the class, not the class itself.
* Static methods must be called on the class itself, not on the instances of the class.
* The static variable is initialized immediately once the execution of the class starts whereas the non-static variables are initialized only after creating the object of the class and that is too for each time the object of the class is created.
* A static variable is initialized only once during the life cycle of a class whereas a non-static variable gets initialized either 0 or n number of times, depending on the number of objects created for that class.
* If you want to access the static members of a class, then you need to access them using the class name whereas you need an instance of a class to access the non-static members

**What is OOP?**

* Understanding OOPs concepts can help in making decisions about how you should design an application and what language one must use
* To make more complex with easier development

**Polymorphism and Types of Polymorphism **

* In polymorphism, we will declare methods with same name and different parameters in same class or methods with same name and same parameters in different classes.
* Polymorphism has the ability to provide the different implementation of methods that are implemented with the same name
* its implemented using overload & override
* overload--->in a class having 2 or more methods with name & diff parameter
* override---> in derived class v will create method which is already
* Implemented in base class with same signature.
* Polymorphism has ability to provide different implementation of methods that are implemented with same name

**Data Abstraction **

* Abstraction is a process of hiding the implementation details and displaying the necessary features or providing relevant information
* abstraction is implemented using interface and abstract class
* The advantage of abstraction is that every user will get his own view of the data according to his requirements and will not get confused with unnecessary data
* Define common definition of class that multiple derived class can share

class Program

{

           static void Main(string[] args)

           {

                Hyundai hyn = new Hyundai();

                String descp = hyn.Describe();

                Console.WriteLine(descp);

                Console.ReadKey();

            }

 }

abstract class Car

{

              public virtual string Describe()

              {

                      return "Description of the car";

               }

   }

class Hyundai : Car

{

         public override string Describe()

         {

                 return "Description of the car is now Hyundai";

          }

 }

**Inheritance. **

* Create the new class from existing class or base class..
* so it can access all properties of base class
* purpose--- code reusability
* When a class includes a property of another class it is known as inheritance.
* Types of inheritance
* For example, a child includes  the properties of its parents

public class ParentClass

    {

        public ParentClass()

        {

            Console.WriteLine("Parent Constructor.");

        }

        public void print()

        {

            Console.WriteLine("I'm a Parent Class.");

        }

    }

    public class ChildClass : ParentClass

    {

        public ChildClass()

        {

            Console.WriteLine("Child Constructor.");

        }

        public static void Main()

        {

            ChildClass child = new ChildClass();

            child.print();

        }

    }

**Data Encapsulation **

* Encapsulation is used for hide the code and data in a single unit to protect the data from the outside the world.
* Encapsulation is implemented using access modifier.
* Class is the best example of encapsulation
* Encapsulation is a technique used to protect the information in an object from another object.

**Abstract class and Abstract Members **

* **Abstract Method:** A method which is declared abstract, has no “body” and declared inside the abstract class only
* **Abstract class** allows you to create functionality that subclasses can implement or override. An interface only allows you to define functionality, not implement it.
* And whereas a class can extend only one abstract class, it can take advantage of multiple interfaces
* An abstract class is a special type of class that cannot be instantiated. An abstract class is designed to be inherited by subclasses that either implement or override its methods.
* In other words, abstract classes are either partially implemented or not implemented at all.
* You can have functionality in your abstract class—the methods in an abstract class can be both abstract and concrete

using System;

public abstract class GeeksForGeeks {

    public abstract void gfg();

}

public class Geek1 : GeeksForGeeks

{

    public override void gfg()

    {

        Console.WriteLine("class Geek1");

    }

}

public class Geek2 : GeeksForGeeks

{

     public override void gfg()

    {

        Console.WriteLine("class Geek2");

    }

}

public class main\_method {

    public static void Main()

    {

        GeeksForGeeks g;

        g = new Geek1();

        g.gfg();

        g = new Geek2();

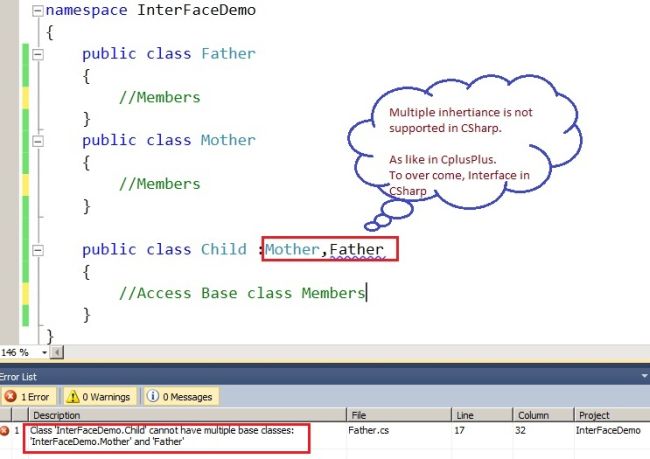
                g.gfg();

    }

}

**Interface **

* An interface looks like a class, but has no implementation.
* The only thing it contains are declarations of events, indexers, methods and/or properties.
* The reason interfaces only provide declarations is because they are inherited by structs and classes
* It is used to achieve loose coupling
* To achieve multiple inheritance



**Sealed Class **

* Sealed classes are used to restrict the inheritance feature of object-oriented programming. Once a class is defined as a sealed class, the class cannot be inherited
* Sealed method is used, so that no other class can override it and implement its own method
* We will have only 1BHK, 2BHK and 3BHK flat in the Apartment. This is the constraint and more than this i.e. 4BHK or above will not be allowed in this apartment.
* Second constraint is 1BHK flat will have rectangular balcony and 2BHK or above will have only circular balcony. Above 2BHK, flats cannot have their own balcony design but circular only.
* In this design, we will use multilevel inheritance as some of the features of 1BHK can be inherited in 2 BHK and some 2 BHK features can be inherited in 3 BHK etc.
* **Use of Sealed class** – Since, first constraint is that the Apartment can have only up to 3BHK flats. Therefore, we will have to mark 3BHK class as sealed. So, that no further extension is possible, means we cannot create 4BHK or above in the apartment.
* **Use of Sealed method** – As per second constraint i.e. 1 BHK has only rectangular balcony and 2BHK or above will have only circular balcony. Also note that 3BHK or above will not be allowed to have their own design (implementation) and will have circular only. Means, it will inherit the balcony design of 2BHK.  
  Hence, we will mark balcony () method as sealed in 2BHK to prevent overriding in 3BHK or above classes

**Events and Delegates **

* It holds the address or reference of single or more method
* Signature of delegate should match with method signature(void)
* Suppose if you have multiple methods with same signature (return type & number of parameters) and want to call all the methods with single object then we can go for delegates.
* Delegates are two types
  + Single Cast Delegates
  + Multi Cast Delegates
* Single cast delegate means, which hold address of single method like as, explained in above example.
* Multi cast delegate is used to hold address of multiple methods in single delegate. To hold multiple addresses with delegate we will use overloaded += operator and if you want remove addresses from delegate we need to use overloaded operator -=

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

public class Sampleclass

{

public static void Add(int x, int y)

{

Console.WriteLine("Addition Value: "+(x + y));

}

public static void Sub(int x, int y)

{

Console.WriteLine("Subtraction Value: " + (x - y));

}

public static void Mul(int x, int y)

{

Console.WriteLine("Multiply Value: " + (x \* y));

}

}

class Program

{

static void Main(string[] args)

{

Sampleclass sc=new Sampleclass();

MultiDelegate del = Sampleclass.Add;

del += Sampleclass.Sub;

del += Sampleclass.Mul;

del(10, 5);

Console.ReadLine();

}

}

Output

|  |
| --- |
| Addition Value : 15  Subtraction Value : 5  Multiply Value : 50 |

**Value Type vs. Reference Type **

* In c#, a [data type](https://www.tutlane.com/tutorial/csharp/csharp-data-types-with-examples) is a **Value Type** if it hold the value of [variable](https://www.tutlane.com/tutorial/csharp/csharp-variables-with-examples) directly on its own memory space and Value Types will use **Stack** memory to store the variables values.
* For example, if we define and assign a value to the variable like int x = 123; then the system will use the same memory space of variable ‘**x**’ to store the value ‘**123**’

using System;

namespace Tutlane

{

    class Program

    {

        static void Square(int a, int b)

        {

            a = a \* a;

            b = b \* b;

            Console.WriteLine(a + " " + b);

        }

        static void Main(string[] args)

        {

            int num1 = 5;

            int num2 = 10;

            Console.WriteLine(num1 + " " + num2);

            Square(num1, num2);

            Console.WriteLine(num1 + " " + num2);

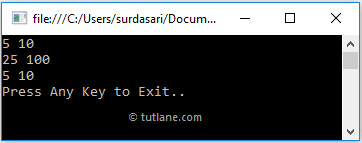
            Console.WriteLine("Press Enter Key to Exit..");

            Console.ReadLine();

        }

    }

}



* In c#, **Reference Types** will contain a pointer, which points to other memory location that holds the data.
* The **Reference Types** will not store the variable value directly in its memory instead, it will store the memory address of the variable value to indicate where the value is being stored

using System;

namespace CsharpExamples

{

    class Person

    {

        public int age;

    }

    class Program

    {

        static void Square(Person a, Person b)

        {

            a.age = a.age \* a.age;

            b.age = b.age \* b.age;

            Console.WriteLine(a.age + " " + b.age);

        }

        static void Main(string[] args)

        {

            Person p1 = new Person();

            Person p2 = new Person();

            p1.age = 5;

            p2.age = 10;

            Console.WriteLine(p1.age + " " + p2.age);

            Square(p1, p2);

            Console.WriteLine(p1.age + " " + p2.age);

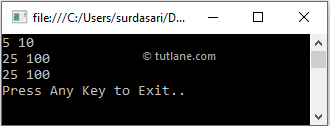
            Console.WriteLine("Press Any Key to Exit..");

            Console.ReadLine();

        }

    }

}



**Boxing and Unboxing **

* Boxing is a process of converting value type into reference type
* Unboxing is a process of converting reference type to value type.

**Usage of REF and OUT Keyword. **

* Ref and out parameters in C# allows us to pass the parameters by reference instead of Value.
* We can change this default behavior. If we want to change the original values inside our methods, we can do that by using ref and out keywords inside the method signature and inside the method call as well.
* We can use the ref keyword only if the variable, which we use as an argument, is initialized before calling a method.
* By using the out keyword, we do not have to initialize a variable before calling a method but we must initialize it inside a method.
* So, let us simplify that. If we want to change an existing value of a variable inside a method, we are going to use the ref keyword.
* But, if we want to assign a completely new value to the variable inside a method, then we use the out keyword

class Program

{

    public static void ChangeRef(ref int numberRef)

    {

        numberRef = 25;

        Console.WriteLine($"Inside the ChangeRef method the numberRef is {numberRef}");

    }

    public static void ChangeOut( out int numberOut)

    {

        numberOut = 60;

        Console.WriteLine($"Inside the ChangeOut method the numberOut is {numberOut}");

    }

    static void Main(string[] args)

    {

        int numberRef = 15;

        Console.WriteLine($"Before calling the ChangeRef method the numberRef is {numberRef}");

        ChangeRef(ref numberRef);

        Console.WriteLine($"After calling the ChangeRef method the numberRef is {numberRef}");

        Console.WriteLine();

        int numberOut;

        Console.WriteLine("Before calling the ChangeOut method the numberOut is unassigned");

        ChangeOut(out numberOut);

        Console.WriteLine($"After calling the ChangeOut method the numberOut is {numberOut}");

        Console.ReadKey();

}

**Conditional Statements **

* A statement that can be executed based on a condition is known as a “Conditional Statement”.
* The following are the 2 types:

Conditional Branching

Conditional Looping

* In C# are the following 2 conditional branching statements:

IF statement

Switch statement

* C# provides 4 loops that allow you to execute a block of code repeatedly until a certain condition is met; they are:
* For Loop
* While loop
* Do ... While Loop
* Foreach Loop

**Ex:**

int i = 20;

        if (i == 10)

            Console.WriteLine("i is 10");

        else if (i == 15)

            Console.WriteLine("i is 15");

        else if (i == 20)

            Console.WriteLine("i is 20");

        else

            Console.WriteLine("i is not present");

int number = 30;

        switch(number)

        {

        case 10: Console.WriteLine("case 10");

                 break;

        case 20: Console.WriteLine("case 20");

                 break;

        case 30: Console.WriteLine("case 30");

                 break;

        default: Console.WriteLine("None matches");

                 break;

        }

for (int i = 0; i <= 50; i++)

            {

                Console.WriteLine(i);

            }

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

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

foreach (string i in cars)

{

Console.WriteLine(i);

}

**Array **

* As we know a primitive type variable such as int, double can hold only a single value at any given point of time. For example, **int no = 10;**. Here the variable **“no”** holds a value of **10**. As per your business requirement, if you want to store 100 integer values, then you may create 100 integer variables which are not a good programming approach as it will take lots of time as well as your code becomes bigger. So to overcome the above problems, Arrays in C# are introduced
* In simple words, we can define an array as a collection of similar type of values which are stored in sequential order
* int[] nums; // stores integer values
* string[] names; // stores string values
* double[] salaries; // stores double values
* object[] objs; // stores objects
* Student[] students; // stores objects of the Student class
* int[] nums = new int[5];
* int[] nums = new int[5]{ 10, 15, 16, 8, 6 };
* int[] nums = new int[]{ 10, 15, 16, 8, 6 };
* int[] nums = { 10, 15, 16, 8, 6 };
* int[] nums = new int[5];
* nums[0] = 10;
* nums[1] = 15;
* nums[2] = 16;
* nums[3] = 8;
* nums[4] = 6;

int[] nums = { 10, 15, 16, 8, 6 };

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

Console.WriteLine(nums[i]);

* The advantages of using an array in C# are as follows:
* It is used to represent similar types of multiple data items using a single name.
* We can use arrays to implement other data structures such as linked lists, trees, graphs, stacks, queues, etc.
* The two-dimensional arrays in C# are used to represent matrices.
* The Arrays in C# are strongly typed. That means they are used to store similar types of multiple data items using a single name. As the arrays are strongly typed so we are getting two advantages. First, the performance of the application will be much better because boxing and unboxing will not happen. Secondly, runtime errors will be prevented because of a type mismatch. In this case, at compile time it will give you the error if there is a type mismatch

**Limitation of array**

* we can say that the Arrays in C# are the simple data structure which is used to store similar types of data items in sequential order
* you need to specify the array’s size while creating the array
* The array size is fixed
* We can never insert an element into the middle of an array

**Collections. **

* Collection in C# is a dynamic array. That means the collections in C# have the capability of storing multiple values but with the following features.
* Size can be increased dynamically.
* We can insert an element into the middle of a collection.
* It also provides the facility to remove or delete elements from the middle of a collection.
* .NET supports two types of collections, generic collections and non-generic collections.
* The following table lists and matches these classes.

 Non-generic                          Generic

 ArrayList     ------------->          List

 HashTable ------------->          Dictionary

 SortedList   ------------->          SortedList

 Stack           ------------->          Stack

 Queue         ------------->          Queue

**ArrayList**

* Arraylist is a class that is similar to an array, but it can be used to store values of various types.
* An Arraylist does not have a specific size.
* Any number of elements can be stored.
* ArrayList al = new ArrayList();
* string str = "kiran teja jallepalli";
* int x = 7;
* DateTime d = DateTime.Parse("8-oct-1985");
* al.Add(str);
* al.Add(x);
* al.Add(d);
* foreach (object o in al)
* {
* Response.Write(o);
* Response.Write("<br>");
* }

**Difference between Array and Collections. **

|  |  |
| --- | --- |
| **Array** | **Collection** |
| 1. Array is Group of Homogeneous data type object. | 1. Collection is Group of Homogeneous and Heterogeneous data type object. |
| 2. Array is fixed in size. | 2. Collection is not fixed in size. |
| 3. Array is Strong type. | 3. Collection is not strong Type. |
| 4. There is no boxing and unboxing process on Array | 4. There is process of Boxing and Unboxing on Collection. |
| Performance wise faster than Array | Slower |

**Constructor, Destructor & Constructor types **

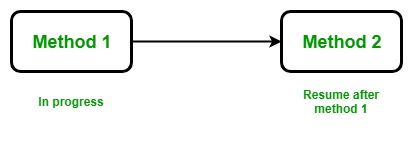
* Constructors are a particular type of method that is associated with a class and gets automatically invoked when the classes instance (i.e., objects) are created. Like other member functions, i.e., methods, these constructors also contain specific instructions that get executed when the objects are created. It is specifically used to assign values to all or some data members of a class.
* Some Special Characteristics of the Constructor:
* Constructor will not return anything.
* Constructor name is same as class name.
* By default C# will create default constructor internally.
* Constructor with no arguments and no body is called default constructor.
* Constructor with arguments is called parameterized constructor.
* Constructor by default public.
* We can create private constructors.
* A method with same name as class name is called constructor there is no separate keyword.
* Types of Constructors in C#
* **Default Constructor**: When constructors do not have parameters, then it is called the default constructor. These types of constructors have all its instance initialized with the same value.
* **Parameterized Constructor**: When any constructor has at least one parameter, then it is called the parameterized constructor.
* **Copy Constructor:** When the constructor is used to create an object just by copying all of its variables from another object, such constructors are called copy constructor. They are used for initializing a new instance from an existing one.
* **Private Constructor**: When a constructor is produced with a private access modifier, it is called Private Constructor. It does not make it possible for other classes to inherit any data from this class.
* **Static Constructor:** When a constructor needs to be invoked only once and when that constructor needs to be invoked at the creation of the first reference, and then those constructors are made static and are called static constructors

**Destructor**

* Destructor is another method that uses the class-name but is preceded with a ~ (tilde) operator/symbol. Destructors are used to de-initialize object, and the memories occupied when constructors are created. You can consider them as the opposite of constructors
* ~ EmpGkr()
* {
* val = 0;
* }

**Multi-threading **

* Multitasking is the simultaneous execution of multiple tasks
* Windows operating system is an example of multitasking because it is capable of running more than one process at a time like running Google Chrome, Notepad, VLC player etc. at the same time
* Every program that executes on your system is a process and to run the code inside the application a process uses a term known as an *thread*
* So every program has logic and a thread is responsible for executing this logic



* **Explain in Thread Project**
* Here, first of all, *method1*executes. In *method1*, *for*loop starts from 0 when the value of is equal to 5 then the method goes into sleep for 6 seconds and after 6 seconds it resumes its process and print remaining value. Until *method2*is in the waiting state. *method2*start its working when *method1*complete its assigned task. So to overcome the drawback of single threaded model multithreading is introduced.
* For example, we have a class and this call contains two different methods, now using multithreading each method is executed by a separate thread. So the major advantage of multithreading is it works simultaneously, means multiple tasks executes at the same time

**Using Dependency Injection. **

* The Dependency Injection is a design pattern that allows us to develop loosely coupled software components
* <https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/>

**CRUD operation (Insert, update, delete and select). **

**CRUD operation using store procedure (Insert, update, delete and select). **

**What is Service? and Types of Services in .NET **

**Web service and working with Web services.**

* Let's think of a scenario where I am planning to show information on regional, national and international news, weather information, ongoing sports scores and other personalized content in a web site. Just think about how much effort and time it will take me to develop this application if I write the code for all these functionalities. On the other hand all these functionalities are already provided by other available sites. So, what if I can use this existing logic in my application? But, the question here is “how I can use someone else's business logic in my application?”.
* For situations of this sort (& many other), we have techniques like **Web Services**
* With Web Services, you can reuse someone else's business logic instead of replicating it yourself, using just a few lines of code
* Web Service is an application that is designed to interact directly with other applications over the internet. In simple sense, Web Services are means for interacting with objects over the Internet.
* **Note 1:**Web Services are not limited to the .NET Framework. The standards were defined before .NET was released and they are exposed, used and supported by vendors other than Microsoft.  
  **Note 2:** Web Services are cross-platform; a service written in one language can be invoked by an application in some other language. The only requirement for accessing a service is an internet connection to make the HTTP request
* Since a web service is cross-platform, there should be some commonly understandable language for requesting a service and getting a response from the service. Such a standard common language is XML. That's why Web Services are built on XML-based standards for exchanging data
* Explained in Project