## OOPS

Object-Oriented Programming is a programming model that works on a principle that revolves around objects rather than action or logic. It allows the users to create objects based on the requirement and then create methods to operate upon those objects.

Working on these objects to obtain the desired result is the goal of object-oriented programming.

## **Namespace**

A namespace in C# is a collection of classes. It provides a structure to keep one class name separate from another class name by declaring them in a different namespace. So that the classes with the same name don’t conflict with one another.

## using statement

The using statement is used to set one or more than one resource. These resources are executed and the resource is released. The statement is also used with database operations.

The main goal is to manage resources and release all the resources automatically.

**C# using statement**

C# and .NET provide resource management for managed objects through the garbage collector - You do not have to explicitly allocate and release memory for managed objects. Clean-up operations for any unmanaged resources should be performed in the destructor in C#.

To allow the programmer to explicitly perform these clean-up activities, objects can provide a Dispose method that can be invoked when the object is no longer needed. The C# using statement defines a boundary for the object outside of which, the object is automatically destroyed. The using statement in C# is exited when the end of the "using" statement block or the execution exits the "using" statement block indirectly, for example - an exception is thrown.

The "using" statement allows you to specify multiple resources in a single statement. The object could also be created outside the "using" statement. The objects specified within the using block must implement the IDisposable interface. The framework invokes the Dispose method of objects specified within the "using" statement when the block is exited.

## ****Class****

A class is a user-defined **prototype** from which objects are created. Basically, a class combines the fields and methods(member function which defines actions) into a single unit

**Objects**

Objects are the instances of the class.

## ****Encapsulation****

Encapsulation is an object-oriented programming concept that allows programmers to wrap data and code snippets inside an enclosure. By using the encapsulation program, you can hide the members of one class from another class

**Encapsulation is implemented by using access specifiers**

**Public:** The public keyword allows its members to be visible from anywhere inside the project. This access specifier has the least visibility restriction.

**Private:** The private members can only be accessed by the member within the same class. This has one of the most restricted visibility.

**Protected:** Protected accessibility allows the member to be accessed from within the class and from another class that inherits this class.

**Internal:** Internal provides accessibility from within the project. Another similar internal accessibility is protected internal. This allows the same as the internal and the only difference is that a child class can inherit this class and reach its members even from another project.

## ****Polymorphism****

, it means one with many forms. It allows the class in to have multiple implementations with the same name,

### **Compile-time Polymorphism**

Compile-time polymorphism is also known as Static polymorphism. Method overloading is one of the ways in which compile-time polymorphism is achieved. It is known as compile-time polymorphism as the method calling decision is made at the time of compiling.

It is achieved by keeping the method name the same but passing different sets of parameters. In method overloading, the system first checks the parameter used and based on the set of parameter it decides to call the appropriate method.

### **Dynamic Polymorphism or Runtime Polymorphism**

Runtime polymorphism or dynamic polymorphism occurs when both method name and method signature have the same name and parameters.

Method overriding is an example of dynamic polymorphism. It allows the user to create an abstract class with partial interface implementation.

Method overriding is achieved using inheritance. To achieve method overriding both the base class and derived class should have the same name and parameter. During compile time the compiler is not able to recognize the overriding method, hence, it doesn’t throw any error. The decision to run a method is taken during runtime.

**Inheritance**

Inheritance is an important part of the OOPS concept. In inheritance, we define parent and child classes. The child class can inherit all the methods, objects and properties of the parent class. A child class can also have its own methods and specific implementation.

The parent class is also known as a base class and the child class that inherits the base class is also known as derived class.

## Abstraction

Abstraction allows the programmer to display only the necessary details to the world while hiding the others.

**Abstraction is achieved in C# by using the Abstract class and interface.**

A class can be declared as an abstract class by using the “Abstract” keyword. The Abstract class in C# is always the base class in the hierarchy. What makes them different from the other class is that they cannot be instantiated.

# **Abstract Class & Interface**

## Abstract class

An Abstract class is a class that cannot be instantiated .This class must contain at least one abstract method, which is marked by the keyword or modifier abstract in the class definition.  It is not possible to create instances of a static class using the new keyword.The Abstract classes are typically used to define a base class in the class hierarchy.

When we have the requirement of a class that contains some common properties or methods with some common properties whose implementation is different for different classes, in that situation, it's better to use Abstract Class then Interface.

* Generally, we use abstract class at the time of *inheritance*.
* A user must use the *override* keyword before the method which is declared as abstract in child class, the abstract class is used to inherit in the child class.
* An abstract class cannot be inherited by structures.
* It can contains constructors or destructors.
* It can implement functions with non-Abstract methods.
* It cannot support multiple inheritance.
* It can’t be static

## Interface

Interface is a contract contains the signature of the functionalities. An interface can only contain declarations but not implementations.. All the methods described inside the interface are abstract by default. It doesn’t have any method body and it cannot be instantiated.

Difference between Abstract Class and Interface

Using interface-based design concepts provides loose coupling, component-based programming, easier maintainability, makes your code base more scalable and makes code reuse much more accessible because the implementation is separated from the interface. Interfaces add a plug and play like architecture into your applications. Interfaces help define a contract (agreement or blueprint, however you chose to define it), between your application and other objects. This indicates what sort of methods, properties, and events are exposed by an object.

## Difference between abstract class and interface

|  |  |
| --- | --- |
| ABSTRACT CLASS | INTERFACE |
| It contains both declaration and definition part. | It contains only a declaration part. |
| Multiple inheritance is not achieved by abstract class. | Multiple inheritance is achieved by interface. |
| It contain [constructor](https://www.geeksforgeeks.org/c-sharp-constructors/). | It does not contain [constructor](https://www.geeksforgeeks.org/c-sharp-constructors/). |
| It can contain static members. | It does not contain static members. |
| It can contain different types of access modifiers like public, private, protected etc. | It only contains public access modifier because everything in the interface is public. |
| The performance of an abstract class is fast. | The performance of interface is slow because it requires time to search actual method in the corresponding class. |
| It is used to implement the core identity of class. | It is used to implement peripheral abilities of class. |
| A class can only use one abstract class. | A class can use multiple interface. |
| If many implementations are of the same kind and use common behavior, then it is superior to use abstract class. | If many implementations only share methods, then it is superior to use Interface. |
| Abstract class can contain methods, fields, constants, etc. | Interface can only contain methods . |
| It can be fully, partially or not implemented. | It should be fully implemented. |

**When do you use Abstract Class?**

When you have a requirement where your base class should provide the default implementation of certain methods whereas other methods should be open to being overridden by child classes that time you have to use abstract classes.

**Why can Abstract class not be Instantiated?**

* Because, it has not fully implemented the class as its abstract methods can not be executed.
* If the compiler allows us to create the object for the abstract class, then you can invoke the abstract method using that object which cannot be executed by CLR at runtime.
* Hence, to restrict the calling of abstract methods, the compiler does not allow you to instantiate an abstract class.

Which type of members can you define in an Abstract class?

You can define all static and non-static members including properties, fields, indexers and also abstract methods.

## Should I use an abstract class or an interface?

Abstract classes provide you the flexibility to have certain concrete methods and some other methods that the derived classes should implement. By contrast, if you use interfaces, you would need to implement all the methods in the class that extends the interface. An abstract class is a good choice if you have plans for future expansion – i.e. if a future expansion is likely in the class hierarchy. If you would like to provide support for future expansion when using interfaces, you’ll need to extend the interface and create a new one.

On a different note, it is easy to add a new interface to the hierarchy if need be. However, if you already have an abstract class in your hierarchy, you can’t add another—i.e., you can add an abstract class only if none are available. You should use an interface if you want a contract on some behavior or functionality. You should not use an interface if you need to write the same code for the interface methods. In this case, you should use an abstract class, define the method once, and reuse it as needed. Do use interfaces to decouple your application’s code from specific implementations of it, or to restrict access to members of a certain type.

# **Static Class & Costructors**

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 you cannot inherit a static class from another class*..* It is not possible to create instances of a static class using the new keyword. Static classes are loaded automatically by the .NET Framework common language runtime (CLR) when the program or namespace containing the class is loaded.

Use a static class to contain methods that are not associated with a particular object. For example, it is a common requirement to create a set of methods that do not act on instance data and are not associated to a specific object in your code. You could use a static class to hold those methods.

* The main features of a static class are:
* They only contain static members.
* They cannot be instantiated.
* They are sealed.
* They cannot contain Instance Constructors or simply constructors as we know that they are associated with objects and operates on data when an object is created

Calculator.Store(result)

What is Sealed Class?

* Sealed class is used to restrict the inheritance feature of object oriented programming.
* Once a class is defined as a sealed class, the class can not be inherited.
* A class, which restricts inheritance for security reason is declared as a sealed class.
* Sealed class is the last class in the hierarchy.
* Sealed class can be a derived class but cannot be a base class.
* A sealed class can not also be an abstract class. Because abstract class has to provide functionality and here you are restricting it to inherit.
* If you have ever noticed, structs and enum are sealed.

What is the difference between Static class and Singleton instance?

* In C# a static class can not implement an interface. While a single instance class needs to implement an interface for some business reason or IoC purpose, you can use the Singleton pattern without a static class.
* You can clone the object of Singleton but, you cannot clone the static class object.
* Singleton object stores in heap but, static object stores in stack.
* A singleton can be initialized lazily or asynchronously while a static class is generally initialized when it is first loaded.

**In C#, the static class contains two types of static members as follows:**

## Static Data Members:

As static class always contains static data members, so static data members are declared using static keyword and they are directly accessed by using the class name. The memory of static data members is allocating individually without any relation with the object.

## Static Methods:

As static class always contains static methods, so static methods are declared using static keyword. These methods only access static data members, they cannot access non-static data members.

Difference between static and non-static class

## Difference between static class and non static class

|  |  |
| --- | --- |
| STATIC CLASS | NON-STATIC CLASS |
| Static class is defined using static keyword. | Non-Static class is not defined by using static keyword. |
| In static class, you are not allowed to create objects. | In non-static class, you are allowed to create objects using new keyword. |
| The data members of static class can be directly accessed by its class name. | The data members of non-static class is not directly accessed by its class name. |
| Static class always contains static members. | Non-static class may contain both static and non-static methods. |
| Static class does not contain an instance constructor. | Non-static class contains an instance constructor. |
| Static class cannot inherit from another class. | Non-static class can be inherited from another class. |

## Difference between Static Constructors and Non-Static Constructors

Static constructors are used to initializing the static members of the class and implicitly called before the creation of the first instance of the class.

Non-static constructors are used to initializing the non-static members of the class. Below are the differences between the Static Constructors and Non-Static Constructors.

**Declaration:** Static constructors are declared using a **static modifier** explicitly while all other remaining constructors are non-static constructors. Non-static constructors can also be called as **Instance Constructors** as they need instance to get executed.

**Calling:** Static constructors are always called implicitly but the non-static constructors are called explicitly i.e by creating the instance of the class.

**Execution:** Static constructor executes as soon as the execution of a class starts and it is the first block of code which runs under a class. But the non-static constructors executes only after the creation of the instance of the class. Each and every time the instance of the class is created, it will call the non-static constructor.

**Times of Execution:** A static constructor will always execute once in the entire life cycle of a class. But a non-static constructor can execute zero time if no instance of the class is created and n times if the n instances are created.

**Initialization of fields:** Static constructors are used to initialize the static fields and non-static constructors are used to initialize the non-static fields.

**Parameters:** We cannot pass any parameters to the static constructors because these are called implicitly and for passing parameters, we have to call it explicitly which is not possible. It will give runtime error as shown in below example. However, we can pass the parameters to the non-static constructor

Overloading: Non-static constructors can be overloaded but not the static constructors. Overloading is done on the parameters criteria. So if you cannot pass the parameters to the Static constructors then we can’t overload it.

Cases in which the constructor will be implicit: Every class except the static class(which contains only static members) always contains an implicit constructor if the user is not defining any explicit constructor. If the class contains any static fields then the static constructors are defined implicitly.

In c#, **Static Constructor** is useful to perform a particular action only once throughout the application. If we declare a [constructor](https://www.tutlane.com/tutorial/csharp/csharp-constructors-with-examples) as **static**, then it will be invoked only once irrespective of the number of class instances and it will be called automatically before the first instance is created.

Generally, in c# the static constructor will not accept any access modifiers and parameters. In simple words, we can say it’s parameterless.

The following are the properties of static constructor in c# programming language.

* Static constructor in c# won’t accept any parameters and access modifiers.
* The static constructor will invoke automatically, whenever we create the first instance of a class.
* The static constructor will be invoked by CLR so we don’t have a control on static constructor execution order in c#.
* In c#, only one static constructor is allowed to create.

## Private Constructors in C#

Private Constructor is a special instance constructor present in C# language. Basically, private constructors are used in class that contains only static members. The private constructor is always declared by using a private keyword.

**Important points:**

* It is the implementation of a singleton class pattern.
* Use private constructor when class have only static members.
* Using private constructor, prevents the creation of the instances of that class.
* If a class contains only private constructor without parameter, then it prevents the automatic generation of default constructor.
* If a class contains only private constructors and does not contain public constructor, then other classes are not allowed to create instances of that class except nested class.

**Note:**If we don’t use any access modifier to define a constructor, then the compiler takes that constructor as a private

## Inheritance in Constructors

In C#, both the base class and the derived class can have their own constructor. The constructor of a base class used to instantiate the objects of the base class and the constructor of the derived class used to instantiate the object of the derived class. In inheritance, the derived class inherits all the members(fields, methods) of the base class, but **derived class cannot inherit the constructor of the base class** because constructors are not the members of the class. Instead of inheriting constructors by the derived class, it is only allowed to invoke the constructor of base class.

# **Difference between readonly and const keyword in C#**

**const**

In C#, a **const**keyword is used to declare constant fields and constant local. The value of the constant field is the same throughout the program or in other words, once the constant field is assigned the value of this field is not be changed. In C#, constant fields and locals are not variables, a constant is a number, string, null reference, boolean values.

**readonly**

In C#, you can use a **readonly**keyword to declare a readonly variable. This readonly keyword shows that you can assign the variable only when you declare a variable or in a constructor of the same class in which it is declared.

## ReadOnly Vs Const Keyword

|  |  |
| --- | --- |
| READONLY KEYWORD | CONST KEYWORD |
| In C#, readonly fields can be created using readonly keyword | In C#, constant fields are created using const keyword. |
| ReadOnly is a runtime constant. | Const is a compile time constant. |
| The value of readonly field can be changed. | The value of the const field can not be changed. |
| It cannot be declared inside the method. | It can be declared inside the method. |
| In readonly fields, we can assign values in declaration and in the contructor part. | In const fields, we can only assign values in declaration part. |
| It can be used with static modifiers. | It cannot be used with static modifiers. |

**How to call base class constructor from derived class in C#?**

Use base keyword for this type of initialization.

**What is base keyword?**

* The base keyword is used to access members of the base class from within a derived class.
* Call a method on the base class that has been overridden by another method.
* Specify which base-class constructor should be called when creating instances of the derived class.
* It is an error to use the base keyword from within a static method.

# **Type conversion`**

**Type conversion** is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms −

* **Implicit type conversion** − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.
* **Explicit type conversion** − These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

## Value Type:

A data type is a value type if it holds a data value within its own memory space. It means variables of these data types directly contain their values.

Its Stored in stack.

### **Passing by Value:**

When you pass a value type variable from one method to another method, the system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.

## Reference Type

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

### **Pass by Reference**

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the address of the variable. If we now change the value of the variable in a method, it will also be reflected in the calling method.

## Null

Reference types have null value by default, when they are not initialized. For example, a string variable (or any other variable of reference type datatype) without a value assigned to it. In this case, it has a null value, meaning it doesn't point to any other memory location, because it has no value yet.

A value type variable cannot be null because it holds a value not a memory address. However, value type variables must be assigned some value before use. The compiler will give an error if you try to use a local value type variable without assigning a value to it.

# Implicitly-Typed Local Variable - var

C# 3.0 introduced the implicit typed local variable "var". Var can only be defined in a method as a local variable. The compiler will infer its type based on the value to the right of the "=" operator.

Implicitly-typed variables must initialized at the time of declaration, otherwise C# compiler would give an error: Implicitly-typed variables must be initialized.

**variables can be used in the following different contexts:**

* Local variable in a function
* For loop
* Foreach loop
* Using statement
* As an anonymous type
* In a LINQ query expression

**Points to Remember :**

* *var* can only be declared and initialized in a single statement. Following is not valid:  
  var i; i = 10;
* *var* cannot be used as a field type at the class level.
* *var* cannot be used in an expression like var i += 10;
* Multiple vars cannot be declared and initialized in a single statement. For example, var i=10, j=20; is inval

# **Difference between Ref and Out keywords in C#**

The **out**is a keyword in C# which is used for the passing the arguments to methods as a reference type. It is generally used when a method returns multiple values. The out parameter does not pass the property.

The **ref**is a keyword in C# which is used for the passing the arguments by a reference. Or we can say that if any changes made in this argument in the method will reflect in that variable when the control return to the calling method. The ref parameter does not pass the [property](https://www.geeksforgeeks.org/c-properties/).

##### **Difference between Ref and Out keywords**

|  |  |
| --- | --- |
| REF KEYWORD | OUT KEYWORD |
| It is necessary the parameters should initialize before it pass to ref. | It is not necessary to initialize parameters before it pass to out. |
| It is not necessary to initialize the value of a parameter before returning to the calling method. | It is necessary to initialize the value of a parameter before returning to the calling method. |
| The passing of value through ref parameter is useful when the called method also need to change the value of passed parameter. | The declaring of parameter through out parameter is useful when a method return multiple values. |
| When ref keyword is used the data may pass in bi-directional. | When out keyword is used the data only passed in unidirectional. |

# **Reflection**

**Reflection** objects are used for obtaining type information at runtime. The classes that give access to the metadata of a running program are in the **System.Reflection** namespace.

The **System.Reflection** namespace contains classes that allow you to obtain information about the application and to dynamically add types, values, and objects to the application.

## Applications of Reflection

Reflection has the following applications −

* It allows view attribute information at runtime.
* It allows examining various types in an assembly and instantiate these types.
* It allows late binding to methods and properties
* It allows creating new types at runtime and then performs some tasks using those types.

# Difference between dispose and finalize

## Dispose

It is used to free unmanaged resources like files, database connections etc. at any time.

Explicitly, it is called by user code and the class which is implementing dispose method, must has to implement IDisposable interface

It belongs to IDisposable interface.

There is no performance costs associated with Dispose method.

## Finalize

It can be used to free unmanaged resources (when you implement it) like files, database connections etc. held by an object before that object is destroyed.

Internally, it is called by Garbage Collector and cannot be called by user code.

It belongs to Object class.

There is performance costs associated with Finalize method since it doesn't clean the memory immediately and called by GC automatically.

# **Structure**

C# includes a value type entity same as class called "structure". Structs are mainly useful to hold small data values. A structure can be defined using the *struct* operator. It can contain parameterized constructor, static constructor, constants, fields, methods, properties, indexers, operators, events and nested types.

## Structure Declaration

A structure is declared using struct keyword with public or internal modifier. The default modifer is internal for the struct and its members. However, you can use private or protected modifier when declared inside a class.

## Difference between Struct and Class:

* Class is reference type whereas struct is value type
* Struct cannot declare a default constructor or destructor. However, it can have parametrized constructors.
* Struct can be instasntiated without the new operator. However, you won't be able to use any of its methods, events or properties if you do so.
* Struct cannot be used as a base or cannot derive another struct or class.

# **enum**

In C#, enum is a value type data type. The enum is used to declare a list of named integer constants. It can be defined using the *enum* keyword directly inside a namespace, class, or structure. The enum is used to give a name to each constant so that the constant integer can be referred using its name.

**Points to Remember :**

* The enum is a set of named constant.
* The value of enum constants starts from 0. Enum can have value of any valid numeric type.
* String enum is not supported in C#.
* Use of enum makes code more readable and manageable.

# **String vs StringBuilder**

**StringBuilder** is used to represent a mutable string of characters. Mutable means the string which can be changed. So **String** objects are immutable but **StringBuilder** is the mutable string type. It will not create a new modified instance of the current string object but do the modifications in the existing string object. The complete functionality of StringBuilder is provided by StringBuilder class which is present in System.Text namespace.

**Need of the StringBuilder:**

 As stated above that the String class objects are immutable which means that if the user will modify any string object it will result into the creation of a new string object. It makes the use of string costly. So when the user needs the repetitive operations on the string then the need of StringBuilder come into existence. It provides the optimized way to deal with the repetitive and multiple string manipulation operations.

**When to use which one:**

* If a string is going to remain constant throughout the program, then use String class object because a String object is immutable.
* If a string can change (example: lots of logic and operations in the construction of the string) then using a StringBuilder is the best option.

**Converting String to StringBuilder:**  
To convert a String class object to StringBuilder class object, just pass the string object to the StringBuilder class constructor.

# **Partial Class**

C# provides the ability to have a single class implementation in multiple .cs files using the ***partial*** modifier [keyword](https://www.tutorialsteacher.com/csharp/csharp-keywords). The *partial* modifier can be applied to a class, method, interface or structure.

### **Partial Class Requirements:**

* All the partial class definitions must be in the same assembly and namespace.
* All the parts must have the same accessibility like public or private, etc.
* If any part is declared abstract, sealed or base type then the whole class is declared of the same type.
* Different parts can have different base types and so the final class will inherit all the base types.
* The Partial modifier can only appear immediately before the keywords class, struct, or interface.
* Nested partial types are allowed.

### **Advantages of Partial Class**

* Multiple developers can work simultaneously with a single class in separate files.
* When working with automatically generated source, code can be added to the class without having to recreate the source file. For example, Visual Studio separates HTML code for the UI and server side code into two separate files: .aspx and .cs files.

## Partial Methods

A partial class or struct may contain partial methods. A partial method must be declared in one of the partial classes. A partial method may or may not have an implementation. If the partial method doesn't have an implementation in any part then the compiler will not generate that method in the final class.

## Requirements for Partial Method

* The partial method declaration must began with the partial modifier.
* The partial method can have a ref but not an out parameter.
* Partial methods are implicitly private methods.
* Partial methods can be static methods.
* Partial methods can be generic.

# **Delegates**

Are function pointers is used to pass a function as a parameter. its used to handles the callback functions or event handler? The answer is - delegate.

The delegate is a reference type data type that defines the method signature that is return type and parameter list types. You can define variables of delegate, just like other data type, that can refer to any method with the same signature as the delegate.

There are three steps involved while working with delegates:

1. Declare a delegate
2. Set a target method
3. Invoke a delegate

A delegate can be declared using the delegate [keyword](https://www.tutorialsteacher.com/csharp/csharp-keywords) followed by a function signature

## Multicast Delegate

The delegate can point to multiple methods. A delegate that points multiple methods is called a multicast delegate. The "+" or "+=" operator adds a function to the invocation list, and the "-" and "-=" operator removes it.

## Generic Delegate

A generic delegate can be defined the same way as a delegate but using generic type parameters or return type. The generic type must be specified when you set a target method.

**Points to Remember :**

* Delegate is the reference type data type that defines the signature.
* Delegate type variable can refer to any method with the same signature as the delegate.
* Syntax: *[access modifier] delegate [return type] [delegate name]([parameters])*
* A target method's signature must match with delegate signature.
* Delegates can be invoke like a normal function or Invoke() method.
* Multiple methods can be assigned to the delegate using "+" or "+=" operator and removed using "-" or "-=" operator. It is called multicast delegate.
* If a multicast delegate returns a value then it returns the value from the last assigned target method.
* Delegate is used to declare an event and anonymous methods in C#.

## Action Delegate

Action is also a delegate type defined in the System namespace. An Action type delegate is the same as [Func delegate](https://www.tutorialsteacher.com/csharp/csharp-func-delegate) except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.

## Func

It’s a built-in generic delegate, It has zero or more *input* parameters and one *out* parameter. The last parameter is considered as an out parameter.

public delegate TResult Func<in T, out TResult>(T arg);

A Func delegate type can include 0 to 16 input parameters of different types. However, it must include one out parameter for result

Func with an Anonymous Method

You can assign an anonymous method to the Func delegate by using the delegate keywordFunc with an Anonymous Method

You can assign an anonymous method to the Func delegate by using the delegate keyword

 Points to Remember :

Func is built-in delegate type.

Func delegate type must return a value.

Func delegate type can have zero to 16 input parameters.

Func delegate does not allow ref and out parameters.

Func delegate type can be used with an [anonymous method](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method) or [lambda expression](https://www.tutorialsteacher.com/linq/linq-lambda-expression).

## Predicate Delegate

A predicate is also a delegate like [Func](https://www.tutorialsteacher.com/csharp/csharp-func-delegate) and [Action](https://www.tutorialsteacher.com/csharp/csharp-action-delegate) delegates. It represents a method that contains a set of criteria and checks whether the passed parameter meets those criteria or not. A predicate delegate methods must take one input parameter and return a boolean - true or false.

The Predicate delegate is defined in the System namespace as shown below:

Predicate signature: public delegate bool Predicate<in T>(T obj);

**https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember:**

1. Predicate delegate takes one input parameter and boolean return type.
2. [Anonymous method](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method) and [Lambda expression](https://www.tutorialsteacher.com/linq/linq-lambda-expression) can be assigned to the predicate delegate.

## Advantages of Action and Func Delegates

1. Easy and quick to define delegates.
2. Makes code short.
3. Compatible type throughout the application.

# **nullable types**

nullable types that allow you to assign null to value type variables. You can declare nullable types using Nullable<t> where T is a type

A nullable of type *int* is the same as an ordinary *int* plus a flag that says whether the *int* has a value or not (is null or not). All the rest is compiler magic that treats "null" as a valid value

The HasValue returns **true** if the object has been assigned a value; if it has not been assigned any value or has been assigned a null value, it will return **false**.

Accessing the value using NullableType.value will throw a runtime exception if nullable type is null or not assigned any value. For example, i.Value will throw an exception if i is null:

## Shorthand Syntax for Nullable Types

You can use the '?' operator to shorthand the syntax e.g. int?, long? instead of using Nullable<T>.

### **?? Operator**

Use the '??' operator to assign a nullable type to a non-nullable type.

### **Assignment Rules**

A nullable type has the same assignment rules as a value type. It must be assigned a value before using it if nullable types are declared in a function as local variables. If it is a field of any class then it will have a null value by default

## Characteristics of Nullable Types

1. Nullable types can only be used with value types.
2. The Value property will throw an InvalidOperationException if value is null; otherwise it will return the value.
3. The HasValue property returns true if the variable contains a value, or false if it is null.
4. You can only use == and != operators with a nullable type. For other comparison use the Nullable static class.
5. Nested nullable types are not allowed. Nullable<Nullable<int>> i; will give a compile time error.

# **Generics in C#**

Generics allow us to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.

A generic class can be defined using angle brackets <>.

Now, the compiler assigns the type based on the type passed by the caller when instantiating a class.

## Generic Class as Base Class

When deriving from a generic base class, you must provide a type argument instead of the base-class's generic type parameter as shown below.

If the generic base class has constraints, the derived class must use the same [constraints](https://www.tutorialsteacher.com/csharp/constraints-in-generic-csharp).

The followings can be generic in C#:

* Interface
* Abstract class
* Class
* Method
* Static method
* Property
* Event
* Delegates
* Operator

## Advantages of Generics

1. Increases the reusability of the code.
2. Generic are type safe. You get compile time errors if you try to use a different type of data than the one specified in the definition.
3. Generic has a performance advantage because it removes the possibilities of boxing and unboxing.

Points to Remember :

1. Generics denotes with angel bracket <>.
2. Compiler applys specified type for generics at compile time.
3. Generics can be applied to interface, abstrct class, method, static method, property, event, delegate and operator.
4. Generics performs faster by not doing boxing & unboxing.

## Constraints in Generics

C# includes **Constraints** to specify which type of placeholder type with the generic class is allowed. It will give a compile time error if you try to instantiate a generic class using a placeholder type that is not allowed by a constraints.

Constraints can be applied using the **where** keyword.

The following table lists the types of generic constraints.

| Constraint | Description |
| --- | --- |
| where T : class | Type must be reference type. |
| where T: struct | Type must be value type. |
| where T: new() | Type must have public parameterless constructor. |
| where T: <base class name> | Type must be or derive from the specified base class |
| where T: <interface name> | Type must be or implement the specified interface. |
| where T: U | Type supplied for T must be or derive from the argument supplied for U. |

**Multiple constraints:**

A generic class can have multiple constraints as shown below.

**Constraint on Generic Methods**

You can apply constraints on the generic methods also.

https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember :

* Constraints specifies the kind of types allowed with the generics.
* Constraints can be applied using the where keyword.
* Six types of constraints can be applied: class, struct, new(), base class name, interface and derived type.
* Multiple constraints also can be applied.

# **Covariance and Contravariance in C#**

Covariance and contravariance allow us to be flexible when dealing with class hierarchy.

## Covariance in C#

Covariance enables you to pass a derived type where a base type is expected. Co-variance is like variance of the same kind. The base class and other derived classes are considered to be the same kind of class that adds extra functionalities to the base type. So covariance allows you to use a derived class where a base class is expected (rule: can accept big if small is expected).

Covariance can be applied on delegate, generic, array, interface, etc.

## C# Contravariance

Contravariane is applied to **parameters**. Cotravariance allows a method with the parameter of a base class to be assigned to a delegate that expects the parameter of a derived class.

# Extension Method and Anonymous ,Dynacmic Types

**Extension methods**

Extension methods, as the name suggests, are additional methods. Extension methods allow you to inject additional methods without modifying, deriving or recompiling the original class, struct or interface. Extension methods can be added to your own custom class, .NET framework classes, or third party classes or interfaces.

An extension method is actually a special kind of static method defined in a static class. To define an extension method, first of all, define a static class.

Now, define a static method as an extension method where the first parameter of the extension method specifies the type on which the extension method is applicable. We are going to use this extension method on int type. So the first parameter must be int prec=\eded with the ***this*** modifier.

Now, you can include the ExtensionMethods namespace wherever you want to use this extension method.

https://www.tutorialsteacher.com/Content/images/bulb-glow.png **Points to Remember :**

1. Extension methods are additional custom methods which were originally not included with the class.
2. Extension methods can be added to custom, .NET Framework or third party classes, structs or interfaces.
3. The first parameter of the extension method must be of the type for which the extension method is applicable, preceded by the **this** keyword.
4. Extension methods can be used anywhere in the application by including the namespace of the extension method.

## Anonymous Method

As the name suggests, an anonymous method is a method without a name. Anonymous methods in C# can be defined using the delegate keyword and can be assigned to a variable of delegate type.

## Anonymous Method Limitations

* It cannot contain jump statement like goto, break or continue.
* It cannot access ref or out parameter of an outer method.
* It cannot have or access unsafe code.
* It cannot be used on the left side of the is operator.

**Points to Remember :**

1. Anonymous method can be defined using the delegate keyword
2. Anonymous method must be assigned to a delegate.
3. Anonymous method can access outer variables or functions.
4. Anonymous method can be passed as a parameter.
5. Anonymous method can be used as event handlers.

## C# - Anonymous Type

In C#, an anonymous type is a type (class) without any name that can contain public read-only properties only. It cannot contain other members, such as fields, methods, events, etc.

You create an anonymous type using the *new* operator. The [implicitly typed variable- ‘var](https://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable)’ is used to hold the reference of anonymous types.

The properties of anonymous types are read-only and cannot be initialized with a null, anonymous function, or a pointer type. The properties can be accessed using dot (.) notation, same as object properties. However, you cannot change the values of properties as they are read-only.

An anonymous type will always be local to the method where it is defined. It cannot be returned from the method. However, an anonymous type can be passed to the method as object type parameter, but it is not recommended. If you need to pass it to another method, then use struct or class instead of an anonymous type.

* Anonymous type is created using the new keyword and object initializer syntax.
* The implicitly typed variable - var, is used to hold the reference of an anonymous type.
* Anonymous type is a reference type data type and all the properties are read-only. It cannot contain methods, events, indexer or any other members.
* A field, property, event, indexer, or return type of a method cannot be anonymous types.
* The scope of an anonymous type is local to the method where it is defined. Mostly used in LINQ queries for temperory use.

Internally, all the anonymous types are directly derived from the Object classMostly, anonymous types are created using the [Select](https://www.tutorialsteacher.com/linq/linq-projection-operators) clause of a LINQ queries to return a subset of the properties from each object in the collection.

var students = from s in studentList

select new { Id = s.StudentID, Name = s.StudentName };

foreach(var stud in students)

Console.WriteLine(stud.Id + "-" + stud.Name);

## Anatomy of the Lambda Expression

C# 3.0(.NET 3.5) introduced the lambda expression along with LINQ. The lambda expression is a shorter way of representing [anonymous method](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method) using some special syntax.

Points to Remember :

1. Lambda Expression is a shorter way of representing anonymous method.
2. Lambda Expression syntax: *parameters => body expression*
3. Lambda Expression can have zero parameter.
4. Lambda Expression can have multiple parameters in parenthesis ().
5. Lambda Expression can have multiple statements in body expression in curly brackets {}.
6. Lambda Expression can be assigned to Func, Action or Predicate delegate.
7. Lambda Expression can be invoked in a similar way to delegate.

## C# - Dynamic Type

C# 4.0 (.NET 4.5) introduced a new type that avoids compile time type checking. You have learned about the implicitly typed variable- [var](https://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable) in the previous section where the compiler assigns a specific type based on the value of the expression. A dynamic type escapes type checking at compile time; instead, it resolves type at run time

A dynamic type changes its type at runtime based on the value of the expression to the right of the "=" operator.

1. The dynamic types are resolved at runtime instead of compile time.
2. The compiler skips the type checking for dynamic type. So it doesn't give any error about dynamic types at compile time.
3. The dynamic types do not have intellisense support in visual studio.
4. A method can have parameters of the dynamic type.
5. An exception is thrown at runtime if a method or property is not compatible.

# **Data Structure & Array**

**Array**

Array is the data structure that stores fixed number of literal values of the same [data type](https://www.tutorialsteacher.com/csharp/csharp-data-types).

## Array Class

All the arrays in C# are derived from an abstract base class [System.Array](https://docs.microsoft.com/en-us/dotnet/api/system.array?view=netframework-4.7.2). The Array class includes static methods for creating, manipulating, searching, and sorting arrays

**Access Array Elements**

We can add, insert, update and access values using index,

We can **sort** a one-dimensional array in two ways, using Array.Sort() method and using LINQ query.

**Array.Sort()**

[**String.Join()**](https://docs.microsoft.com/en-us/dotnet/api/system.string.join)

We can get a comma-separated string from an array using [**String.Join()**](https://docs.microsoft.com/en-us/dotnet/api/system.string.join) method

Removing duplicate values from an array in C# is essentially getting distinct values. In C#, we cannot remove values in the array. Instead, we will have to create a new array with the values we want. So, we have to get the distinct values from the specified array and create a new array of distinct values instead of removing duplicate values.

The following example gets distinct values from an array using the [Distinct()](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.distinct) method and creates a new array.

.

To remove duplicate values and get distinct values from an object array, we need to implement either IEquatable or IEqualityComparer.

The following class implements IEqualityComparer<T> to be used with the Distinct() method.

[**Count()**](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.count)

You can count the total number of elements or some specific elements in the array using an extension method [Count()](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.count) method.

**Union()**

You can combine two arrays with distinct values using Union() method.

## between Array and ArrayList in C#.

| Array | ArrayList |
| --- | --- |
| Must include **System** namespace to use array. | Must include **System.Collections** namespace to use ArraList. |
| Array stores a fixed number of elements. The size of an Array must be specified at the time of initialization. | ArrayList grows automatically and you don't need to specify the size. |
| Array is strongly typed. This means that an array can store only specific type of items\elements. | ArrayList can store any type of items\elements. |
| No need to cast elements of an array while retrieving because it is strongly typed and stores a specific type of items only. | The items of ArrayList need to be cast to an appropriate data type while retrieving. So, boxing and unboxing happens. |
| Performs faster than ArrayList because it is strongly typed. | Performs slows because of [boxging and unboxing](https://www.tutorialsteacher.com/articles/boxing-unboxing-in-csharp). |
| Use static helper class Array to perform different tasks on the array. | ArrayList itself includes various utility methods for various tasks. |

# **Indexers**

An Indexer is a special type of property that allows a class or structure to be accessed like an array for its private collection.

An indexer can be defined the same way as property with **this** keyword and square brackets [].

# **Tuple**

The Tuple<T> class was introduced in .NET Framework 4.0. A tuple is a data structure that contains a sequence of elements of different data types. It can be used where you want to have a data structure to hold an object with properties, but you don't want to create a separate type for it.

Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>

## Accessing Tuple Elements

The elements of a tuple can be accessed with Item<elementNumber> properties e.g. Item1, Item2, Item3 and so on up to Item7 property. The Item1 property returns the first element, Item2 returns the second element and so on. The last element (the 8th element) will be returned using the Rest property.

## Usage of Tuple

Tuples can be used in the following scenarios:

* When you want to return multiple values from a method without using ref or out parameters.
* When you want to pass multiple values to a method through a single parameter.
* When you want to hold a database record or some values temporarily without creating a separate class.

## Tuple Limitations:

* Tuple is a reference type and not a value type. It allocates on heap and could result in CPU intensive operations.
* Tuple is limited to include 8 elements. You need to use nested tuples if you need to store more elements. However, this may result in ambiguity.
* Tuple elements can be accessed using properties with a name pattern Item<elementNumber> which does not make sense.

# Unsafe Codes

The **unsafe code** or the unmanaged code is a code block that uses a **pointer** variable.

A **pointer** is a variable whose value is the address of another variable i.e., the direct address of the memory location. similar to any variable or constant, you must declare a pointer before you can use it to store any variable address.

# Collection

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces. With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc.

NET supports two types of collections, generic collections and non-generic collections

**IEnumerator:** The [IEnumerator](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator?view=netframework-4.7.2) interface supports a simple iteration over a non-generic collection. It includes methods and property which can be implemented to support easy iteration using foreach loop.

**IEnumerable:** The [IEnumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable?view=netframework-4.7.2) interface includes GetEnumerator() method which returns an object of IEnumerator.

So, all the built-in collection classes and custom collection classes must implement IEnumerator and IEnumerable interfaces for easy iteration using foreach loop.

**ICollection:** The [ICollection](https://docs.microsoft.com/en-us/dotnet/api/system.collections.icollection?view=netframework-4.7.2) interface is the base interface for all the collections that defines sizes, enumerators, and synchronization methods for all non-generic collections. The Queue and Stack collection implement ICollection inferface.

**IList:** The [IList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ilist?view=netframework-4.7.2) interface includes properties and methods to add, insert, remove elements in the collection and also individual element can be accessed by index. The ArrayList and BitArray collections implement IList interface.

**IDictionary:** The [IDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.idictionary?view=netframework-4.7.2) interface represents a non-generic collection of key/value pairs. The Hashtable and SortedList implement IDictionary interface and so they store key/value pairs.

As you can see from the diagram, ArrayList, BitArray, Hashtable, SortedList, Queue, and Stack collections implement different interfaces and so, they are used for the different purposes.

## Non Generic Collections

It can contain elements of any data types.

| Non-generic Collections | Usage |
| --- | --- |
| [ArrayList](https://www.tutorialsteacher.com/csharp/csharp-arraylist) | ArrayList stores objects of any type like an array. However, there is no need to specify the size of the ArrayList like with an array as it grows automatically. |
| [SortedList](https://www.tutorialsteacher.com/csharp/csharp-sortedlist) | SortedList stores key and value pairs. It automatically arranges elements in ascending order of key by default. C# includes both, generic and non-generic SortedList collection. |
| [Stack](https://www.tutorialsteacher.com/csharp/csharp-stack) | Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. C# includes both, generic and non-generic Stack. |
| [Queue](https://www.tutorialsteacher.com/csharp/csharp-queue) | Queue stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. C# includes generic and non-generic Queue. |
| [Hashtable](https://www.tutorialsteacher.com/csharp/csharp-hashtable) | Hashtable stores key and value pairs. It retrieves the values by comparing the hash value of the keys. |
| BitArray | BitArray manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0). |

## C# - ArrayList

ArrayList is a non-generic type of collection in C#. It can contain elements of any data types. It is similar to an [array](https://www.tutorialsteacher.com/csharp/array-csharp), except that it grows automatically as you add items in it. Unlike an array, you don't need to specify the size of ArrayList.

Important Properties and Methods of ArrayList

| Properties | Description |
| --- | --- |
| Capacity | Gets or sets the number of elements that the ArrayList can contain. |
| Count | Gets the number of elements actually contained in the ArrayList. |
| IsFixedSize | Gets a value indicating whether the ArrayList has a fixed size. |
| IsReadOnly | Gets a value indicating whether the ArrayList is read-only. |
| Item | Gets or sets the element at the specified index. |

| Methods | Description |
| --- | --- |
| [Add()/AddRange()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#add) | Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection into ArrayList. |
| [Insert()/InsertRange()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#insert) | Insert() method insert a single elements at the specified index in ArrayList. InsertRange() method insert all the elements of the specified collection starting from specified index in ArrayList. |
| [Remove()/RemoveRange()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#remove) | Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList. |
| [RemoveAt()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#removeat) | Removes the element at the specified index from the ArrayList. |
| [Sort()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#sort) | Sorts entire elements of the ArrayList. |
| [Reverse()](https://www.tutorialsteacher.com/csharp/csharp-arraylist#sort) | Reverses the order of the elements in the entire ArrayList. |
| [Contains](https://www.tutorialsteacher.com/csharp/csharp-arraylist#contains) | Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false. |
| Clear | Removes all the elements in ArrayList. |
| CopyTo | Copies all the elements or range of elements to compitible Array. |
| GetRange | Returns specified number of elements from specified index from ArrayList. |
| IndexOf | Search specified element and returns zero based index if found. Returns -1 if element not found. |
| ToArray | Returns compitible array from an ArrayList. |

## C# - Hashtable

C# includes Hashtable collection in *System.Collections* namespace, which is similar to generic [Dictionary](https://www.tutorialsteacher.com/csharp/csharp-dictionary) collection. The Hashtable collection stores key-value pairs. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.

Important Propertis and Methods of Hashtable

| Property | Description |
| --- | --- |
| Count | Gets the total count of key/value pairs in the Hashtable. |
| IsReadOnly | Gets boolean value indicating whether the Hashtable is read-only. |
| Item | Gets or sets the value associated with the specified key. |
| Keys | Gets an ICollection of keys in the Hashtable. |
| Values | Gets an ICollection of values in the Hashtable. |

| Methods | Usage |
| --- | --- |
| [Add](https://www.tutorialsteacher.com/csharp/csharp-hashtable#add) | Adds an item with a key and value into the hashtable. |
| [Remove](https://www.tutorialsteacher.com/csharp/csharp-hashtable#remove) | Removes the item with the specified key from the hashtable. |
| Clear | Removes all the items from the hashtable. |
| [Contains](https://www.tutorialsteacher.com/csharp/csharp-hashtable#contains) | Checks whether the hashtable contains a specific key. |
| [ContainsKey](https://www.tutorialsteacher.com/csharp/csharp-hashtable#contains) | Checks whether the hashtable contains a specific key. |
| [ContainsValue](https://www.tutorialsteacher.com/csharp/csharp-hashtable#contains) | Checks whether the hashtable contains a specific value. |
| GetHash | Returns the hash code for the specified key. |

**Add key-value into Hashtable**

The Add() method adds an item with a key and value into the Hashtable. Key and value can be of any data type. Key cannot be null whereas value can be null.

Add() Signature: *void Add(object key, object value);*

## C# - SortedList

The SortedList collection stores key-value pairs in the ascending order of key by default. SortedList class implements IDictionary & ICollection interfaces, so elements can be accessed both by key and index.

C# includes two types of SortedList, [generic SortedList](https://www.tutorialsteacher.com/csharp/csharp-generic-sortedlist) and non-generic SortedList. Here, we will learn about non-generic SortedList.

Important Properties and Methods of SortedList

| Property | Description |
| --- | --- |
| Capacity | Gets or sets the number of elements that the SortedList instance can store. |
| Count | Gets the number of elements actually contained in the SortedList. |
| IsFixedSize | Gets a value indicating whether the SortedList has a fixed size. |
| IsReadOnly | Gets a value indicating whether the SortedList is read-only. |
| Item | Gets or sets the element at the specified key in the SortedList. |
| Keys | Get list of keys of SortedList. |
| Values | Get list of values in SortedList. |

| Method | Description |
| --- | --- |
| Add(object key, object value) | Add key-value pairs into SortedList. |
| Remove(object key) | Removes element with the specified key. |
| RemoveAt(int index) | Removes element at the specified index. |
| Contains(object key) | Checks whether specified key exists in SortedList. |
| Clear() | Removes all the elements from SortedList. |
| GetByIndex(int index) | Returns the value by index stored in internal array |
| GetKey(int index) | Returns the key stored at specified index in internal array |
| IndexOfKey(object key) | Returns an index of specified key stored in internal array |
| IndexOfValue(object value) | Returns an index of specified value stored in internal array |

## C# - Stack

C# includes a special type of collection which stores elements in LIFO style (Last In First Out). C# includes a generic and non-generic Stack. Here, you are going to learn about the non-generic stack.

Stack allows null value and also duplicate values. It provides a Push() method to add a value and Pop() or Peek() methods to retrieve values.

**Important Properties and Methods of Stack:**

| Property | Usage |
| --- | --- |
| Count | Returns the total count of elements in the Stack. |

| Method | Usage |
| --- | --- |
| [Push](https://www.tutorialsteacher.com/csharp/csharp-stack#push) | Inserts an item at the top of the stack. |
| [Peek](https://www.tutorialsteacher.com/csharp/csharp-stack#peek) | Returns the top item from the stack. |
| [Pop](https://www.tutorialsteacher.com/csharp/csharp-stack#pop) | Removes and returns items from the top of the stack. |
| [Contains](https://www.tutorialsteacher.com/csharp/csharp-stack#contains) | Checks whether an item exists in the stack or not. |
| [Clear](https://www.tutorialsteacher.com/csharp/csharp-stack#clear) | Removes all items from the stack. |

## C# - Queue

C# includes a Queue collection class in the *System.Collection* namespace. Queue stores the elements in FIFO style (First In First Out), exactly opposite of the [Stack](https://www.tutorialsteacher.com/csharp/csharp-stack) collection. It contains the elements in the order they were added.

Queue collection allows multiple null and duplicate values. Use the Enqueue() method to add values and the Dequeue() method to retrieve the values from the Queue.

**Important Properties and Methods of Queue:**

| Property | Usage |
| --- | --- |
| Count | Returns the total count of elements in the Queue. |

| Method | Usage |
| --- | --- |
| [Enqueue](https://www.tutorialsteacher.com/csharp/csharp-queue#enqueue) | Adds an item into the queue. |
| [Dequeue](https://www.tutorialsteacher.com/csharp/csharp-queue#dequeue) | Removes and returns an item from the beginning of the queue. |
| [Peek](https://www.tutorialsteacher.com/csharp/csharp-queue#peek) | Returns an first item from the queue |
| [Contains](https://www.tutorialsteacher.com/csharp/csharp-queue#contains) | Checks whether an item is in the queue or not |
| [Clear](https://www.tutorialsteacher.com/csharp/csharp-queue#clear) | Removes all the items from the queue. |
| TrimToSize | Sets the capacity of the queue to the actual number of items in the queue. |

The limitation of these non generic collection is that while retrieving items, you need to cast into the appropriate data type, otherwise the program will throw a runtime exception. It also affects on performance, because of boxing and unboxing.

## Generic collection

it holds elements of same type, A generic collection gets all the benefit of generics. It doesn't need to do boxing and unboxing while storing or retrieving items and so performance is improved.

generic collection classes in the ***System.Collections.Generic*** namespace.

The following are widely used generic collections:

| Generic Collections | Description |
| --- | --- |
| [List<T>](https://www.tutorialsteacher.com/csharp/csharp-list) | Generic List<T> contains elements of specified type. It grows automatically as you add elements in it. |
| [Dictionary<TKey,TValue>](https://www.tutorialsteacher.com/csharp/csharp-dictionary) | Dictionary<TKey,TValue> contains key-value pairs. |
| [SortedList<TKey,TValue>](https://www.tutorialsteacher.com/csharp/csharp-generic-sortedlist) | SortedList stores key and value pairs. It automatically adds the elements in ascending order of key by default. |
| Hashset<T> | Hashset<T> contains non-duplicate elements. It eliminates duplicate elements. |
| Queue<T> | Queue<T> stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. |
| Stack<T> | Stack<T> stores the values as LIFO (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. |

## List<T>

You have already learned about ArrayList in the previous section. An ArrayList resizes automatically as it grows. The List<T> collection is the same as an ArrayList except that List<T> is a generic collection whereas ArrayList is a non-generic collection.

| Property | Usage |
| --- | --- |
| Items | Gets or sets the element at the specified index |
| Count | Returns the total number of elements exists in the List<T> |

| Method | Usage |
| --- | --- |
| Add | Adds an element at the end of a List<T>. |
| AddRange | Adds elements of the specified collection at the end of a List<T>. |
| BinarySearch | Search the element and returns an index of the element. |
| Clear | Removes all the elements from a List<T>. |
| Contains | Checks whether the speciied element exists or not in a List<T>. |
| Find | Finds the first element based on the specified predicate function. |
| Foreach | Iterates through a List<T>. |
| Insert | Inserts an element at the specified index in a List<T>. |
| InsertRange | Inserts elements of another collection at the specified index. |
| Remove | Removes the first occurence of the specified element. |
| RemoveAt | Removes the element at the specified index. |
| RemoveRange | Removes all the elements that match with the supplied predicate function. |
| Sort | Sorts all the elements. |
| TrimExcess | Sets the capacity to the actual number of elements. |
| TrueForAll | Determines whether every element in theÂ List<T> matches the conditions defined by the specified predicate. |

## Dictionary<TKey, TValue>

The Dictionary<TKey, TValue> class is a generic collection class in the System.Collection.Generics namespace. TKey denotes the type of key and TValue is the type of TValue.

Dictionary cannot include duplicate or null keys, where as values can be duplicated or set as null. Keys must be unique otherwise it will throw a runtime exception.

Important Properties and Methods of IDictionary

| Property | Description |
| --- | --- |
| Count | Gets the total number of elements exists in the Dictionary<TKey,TValue>. |
| IsReadOnly | Returns a boolean indicating whether the Dictionary<TKey,TValue> is read-only. |
| Item | Gets or sets the element with the specified key in the Dictionary<TKey,TValue>. |
| Keys | Returns collection of keys of Dictionary<TKey,TValue>. |
| Values | Returns collection of values in Dictionary<TKey,TValue>. |

| Method | Description |
| --- | --- |
| Add | Adds an item to the Dictionary collection. |
| Add | Add key-value pairs in Dictionary<TKey, TValue> collection. |
| Remove | Removes the first occurrence of specified item from the Dictionary<TKey, TValue>. |
| Remove | Removes the element with the specified key. |
| ContainsKey | Checks whether the specified key exists in Dictionary<TKey, TValue>. |
| ContainsValue | Checks whether the specified key exists in Dictionary<TKey, TValue>. |
| Clear | Removes all the elements from Dictionary<TKey, TValue>. |
| TryGetValue | Returns true and assigns the value with specified key, if key does not exists then return false. |

## SortedList<TKey, TValue>

The generic SortedList SortedList<TKey, TValue> represents a collection of key-value pairs that are sorted by key based on associated [IComparer<T>](https://msdn.microsoft.com/en-us/library/8ehhxeaf(v=vs.110).aspx). A SortedList collection stores key and value pairs in ascending order of key by default.

Important Properties and Methods of Generic SortedList

| Property | Description |
| --- | --- |
| Capacity | Gets or sets the number of elements that the SortedList<TKey,TValue> can store. |
| Count | Gets the total number of elements exists in the SortedList<TKey,TValue>. |
| IsReadOnly | Returns a boolean indicating whether the SortedList<TKey,TValue> is read-only. |
| Item | Gets or sets the element with the specified key in the SortedList<TKey,TValue>. |
| Keys | Get list of keys of SortedList<TKey,TValue>. |
| Values | Get list of values in SortedList<TKey,TValue>. |

| Method | Description |
| --- | --- |
| Add | Add key-value pairs into SortedList<TKey, TValue>. |
| Remove | Removes element with the specified key. |
| RemoveAt | Removes element at the specified index. |
| ContainsKey | Checks whether the specified key exists in SortedList<TKey, TValue>. |
| ContainsValue | Checks whether the specified key exists in SortedList<TKey, TValue>. |
| Clear | Removes all the elements from SortedList<TKey, TValue>. |
| IndexOfKey | Returns an index of specified key stored in internal array of SortedList<TKey, TValue>. |
| IndexOfValue | Returns an index of specified value stored in internal array of SortedList<TKey, TValue> |
| TryGetValue | Returns true and assigns the value with specified key, if key does not exists then return false. |

# **IEnumerable and IQueriable Differences**

**I Enumerable**

* I Enumerable exists in the System.Collections namespace.
* IEnumerable can move forward only over a collection, it can’t move backward and between the items.
* IEnumerable is best to query data from in-memory collections like List, Array, etc.
* While query data from a database, IEnumerable execute a select query on the server side, load data in-memory on a client-side and then filter data.
* IEnumerable is suitable for LINQ to Object and LINQ to XML queries.
* IEnumerable supports deferred execution.
* IEnumerable doesn’t support custom query.

## IQueryable

1. IQueryable exists in System. Linq Namespace.
2. IQueryable can move forward only over a collection, it can’t move backward and between the items.
3. IQueryable is best to query data from out-memory (like remote database, service) collections.
4. While query data from a database, IQueryable execute the select query on the server side with all filters.
5. IQueryable is suitable for[**LINQ to SQL**](https://www.dotnettricks.com/learn/entityframework/difference-between-linq-to-sql-and-entity-framework)queries.
6. IQueryable supports deferred execution.
7. IQueryable supports custom query using CreateQuery and Execute methods.
8. IQueryable support lazy loading. Hence it is suitable for paging like scenarios.
9. Extension methods support by IQueryable takes expression objects means expression tree.

# **Dependency Injection in C#**

Dependency injection makes it easy to create loosely coupled components, which t means that components consume functionality defined by interfaces without having any first-hand knowledge of which implementation classes are being used.

Dependency injection makes it easier to change the behavior of an application by changing the components that implement the interfaces that define application features.

## Constructor Injection

1. This is a widely used way to implement DI.
2. Dependency Injection is done by supplying the DEPENDENCY through the class’s constructor when creating the instance of that class.
3. Injected component can be used anywhere within the class.
4. Recommended to use when the injected dependency, you are using across the class methods.
5. It addresses the most common scenario where a class requires one or more dependencies.

**Constructor Injection**

This is the most commonly used dependency pattern in Object Oriented Programming. The constructor injection normally has only one parameterized constructor, so in this constructor dependency there is no default constructor and we need to pass the specified value at the time of object creation. We can use the injection component anywhere within the class. It addresses the most common scenario where a class requires one or more dependencies.

You cannot control when the dependency is set at all, it can be changed at any point in the object's lifetime.

1. Knowing the types of each IService
2. According to the request, feed the abstract IService to the Client

## Property/Setter Injection

1. Recommended using when a class has optional dependencies, or where the implementations may need to be swapped.
2. Different logger implementations could be used in this way.
3. Does not require the creation of a new object or modifying the existing one. Without changing the object state, it could work.

**Method Injection**

In method injection we need to pass the dependency in the method only. The entire class does not need the dependency, just the one method. I have a class with a method that has a dependency. I do not want to use constructor injection because then I would be creating the dependent object every time this class is instantiated and most of the methods do not need this dependent object.

## Method Injection

1. Inject the dependency into a single method and generally for the use of that method.
2. It could be useful, where the whole class does not need the dependency, only one method having that dependency.
3. This is the way is rarely used.

## Advantages of Dependency Injection

1. Reduces class coupling
2. Increases code reusability
3. Improves code maintainability
4. Make unit testing possible

**What is CORS?**

CORS stands for Cross-Origin Resource Sharing. It is a mechanism that allows restricted resources on a web page to be requested from another domain, outside the domain from which the resource originated. A web page may freely embed images, stylesheets, scripts, iframes, and videos.

For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts. For example, XMLHttpRequest follows the same-origin policy. So, a web application using XMLHttpRequest could only make HTTP requests to its own domain. To improve web applications, developers asked browser vendors to allow XMLHttpRequest to make cross-domain requests.

**same domain, other domain(domain1.com), other domain(domain2.com) with CORS enabled.**

1. When the Request comes from same domain, it works perfectly.
2. When the Request comes from some other domain(domain1.com), it throws an error. It means, the browser has a property called Access-Control-Allow-Origin which restricts the requests from different domains for security purposes. So, we need to enable CORS to accomplish the request.

**Set the allowed origins**

The *origins* parameter of the **[EnableCors]** attribute specifies which origins are allowed to access the resource. The value is a comma-separated list of the allowed origins.

C#Copy

[EnableCors(origins: "http://www.contoso.com,http://www.example.com",

headers: "\*", methods: "\*")]

**Set the allowed HTTP methods**

The *methods* parameter of the **[EnableCors]** attribute specifies which HTTP methods are allowed to access the resource. To allow all methods, use the wildcard value "\*". The following example allows only GET and POST requests.

C#Copy

[EnableCors(origins: "http://www.example.com", headers: "\*", methods: "get,post")]

public class TestController : ApiController

{

public HttpResponseMessage Get() { ... }

public HttpResponseMessage Post() { ... }

public HttpResponseMessage Put() { ... }

}

**Set the allowed request headers**

This article described earlier how a preflight request might include an Access-Control-Request-Headers header, listing the HTTP headers set by the application (the so-called "author request headers"). The *headers* parameter of the **[EnableCors]** attribute specifies which author request headers are allowed. To allow any headers, set *headers* to "\*". To whitelist specific headers, set *headers* to a comma-separated list of the allowed headers:

C#Copy

[EnableCors(origins: "http://example.com",

headers: "accept,content-type,origin,x-my-header", methods: "\*")]

**What is same origin policy?**

Browsers allow a web page to make AJAX requests only within the same domain. Browser security prevents a web page from making AJAX requests to another domain. This is called origin policy.

We can enable CORS in WebAPI,

1. Using JSONP
2. Using Microsoft.AspNet.WebApi.Cors

**Resource files in dot net**

# ***NET Resource files (.resx)***

A resource file contains non-executable data that are used by the application and deployed along with it. Bitmaps, Icons etc are the examples of resource files. In ASP.NET, resource files are used to make application to support multiple cultures. we can create resource files each of them correspond to specific locale or culture of the application. You can use resgen utility to compile resource file into an assembly. You can create a satellite assembly from a compiled resource file using the AL utility provided with Microsoft .NET SDK.  
  
Advantages of resource files are as follows.  
  
- It supports Globalization features in ASP.NET.  
- You can have culture based information separate from the content and logic of the application.  
- You can change resource content without effecting application's code.

SatelliteAssemblies  
Satellite Assemblies are the special kinds of assemblies that exist as DLL and contain culture-specific resources in a binary format. They store compiled localized application resources. They can be created using the AL utility and can be deployed even after deployment of the application.  
Satellite Assemblies encapsulate resources into binary format and thus make resources lighter and consume lesser space on the disk