NET Core Overview

.NET Core is a new version of .NET Framework, which is a free, open-source, general-purpose development platform maintained by Microsoft. It is a cross-platform framework that runs on Windows, macOS, and Linux operating systems.

.NET Core Framework can be used to build different types of applications such as mobile, desktop, web, cloud, IoT, machine learning, microservices, game, etc.

.NET Core is written from scratch to make it modular, lightweight, fast, and cross-platform Framework. It includes the core features that are required to run a basic .NET Core app. Other features are provided as NuGet packages, which you can add it in your application as needed. In this way, the .NET Core application speed up the performance, reduce the memory footprint and becomes easy to maintain.

Why .NET Core?

There are some limitations with the .NET Framework. For example, it only runs on the Windows platform. Also, you need to use different .NET APIs for different Windows devices such as Windows Desktop, Windows Store, Windows Phone, and Web applications. In addition to this, the .NET Framework is a machine-wide framework. Any changes made to it affect all applications taking a dependency on it. Learn more about the motivation behind .NET Core here.

Today, it's common to have an application that runs across devices; a backend on the web server, admin front-end on windows desktop, web, and mobile apps for consumers. So, there is a need for a single framework that works everywhere. So, considering this, Microsoft created .NET Core. The main objective of .NET Core is to make .NET Framework open-source, cross-platform compatible that can be used in a wide variety of verticals, from the data center to touch-based devices.

.NET Core Characteristics

Open-source Framework: .NET Core is an open-source framework maintained by Microsoft and available on GitHub under MIT and Apache 2 licenses. It is a .NET Foundation project.

Cross-platform: .NET Core runs on Windows, macOS, and Linux operating systems. There are different runtime for each operating system that executes the code and generates the same output.

Consistent across Architectures: Execute the code with the same behavior in different instruction set architectures, including x64, x86, and ARM.

Wide-range of Applications: Various types of applications can be developed and run on .NET Core platform such as mobile, desktop, web, cloud, IoT, machine learning, microservices, game, etc.

Supports Multiple Languages: You can use C#, F#, and Visual Basic programming languages to develop .NET Core applications. You can use your favorite IDE, including Visual Studio 2017/2019, Visual Studio Code, Sublime Text, Vim, etc.

Modular Architecture: .NET Core supports modular architecture approach using NuGet packages. There are different NuGet packages for various features that can be added to the .NET Core project as needed. Even the .NET Core library is provided as a NuGet package. The NuGet package for the default .NET Core application model is Microsoft.NETCore.App.

This way, it reduces the memory footprint, speeds up the performance, and easy to maintain.

CLI Tools: .NET Core includes CLI tools (Command-line interface) for development and continuous-integration.

Flexible Deployment: .NET Core application can be deployed user-wide or system-wide or with Docker Containers.

Compatibility: Compatible with .NET Framework and Mono APIs by using .NET Standard specification.

.NET Core Version History

Version	Latest Version	Visual Studio	Release Date	End of Support
.NET 5	Preview 1	VS 2019	16th March, 2020	
.NET Core 3.x - latest	3.1.3	VS 209	24th March, 2020	12th March, 2022
.NET Core 2.x	2.1.17	VS 2017, 2019	24th March, 2020	21st August, 2021
.NET Core 1.x	1.1.13	VS 2017	14th May, 2019	27th May, 2019

Both, .NET 3.1, and .NET Core 2.1 will have long term support.

.NET Core 3.x applications only run on .NET Core Framework.

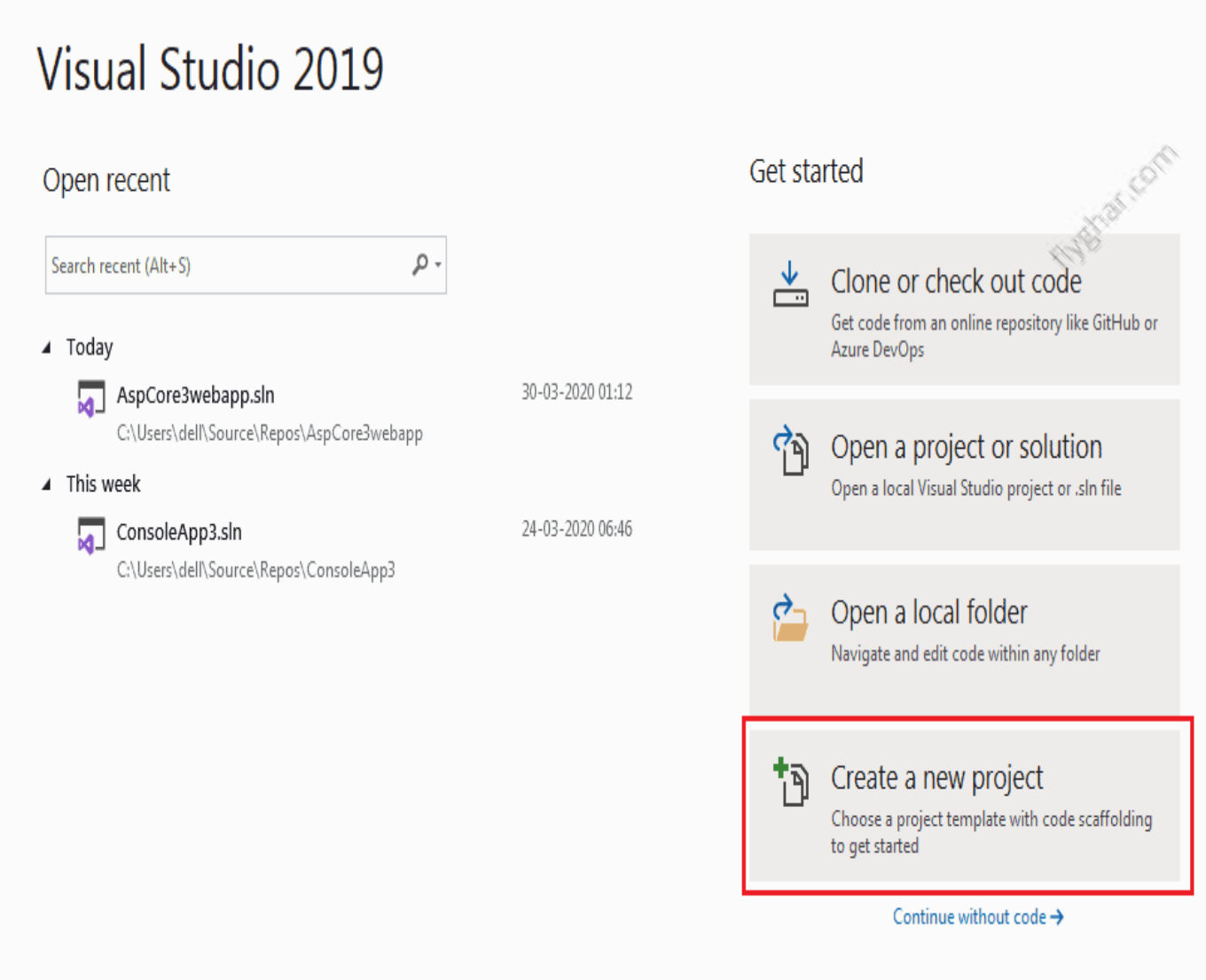
.NET Core 2.x applications run on .NET Core as well as .NET Framework.

Create ASP.NET Core Application

Here, we will learn how to create our first ASP.NET Core 3.0 application in Visual Studio 2019.

Open Visual Studio 2019 and click on **Create a new project**, as shown below.

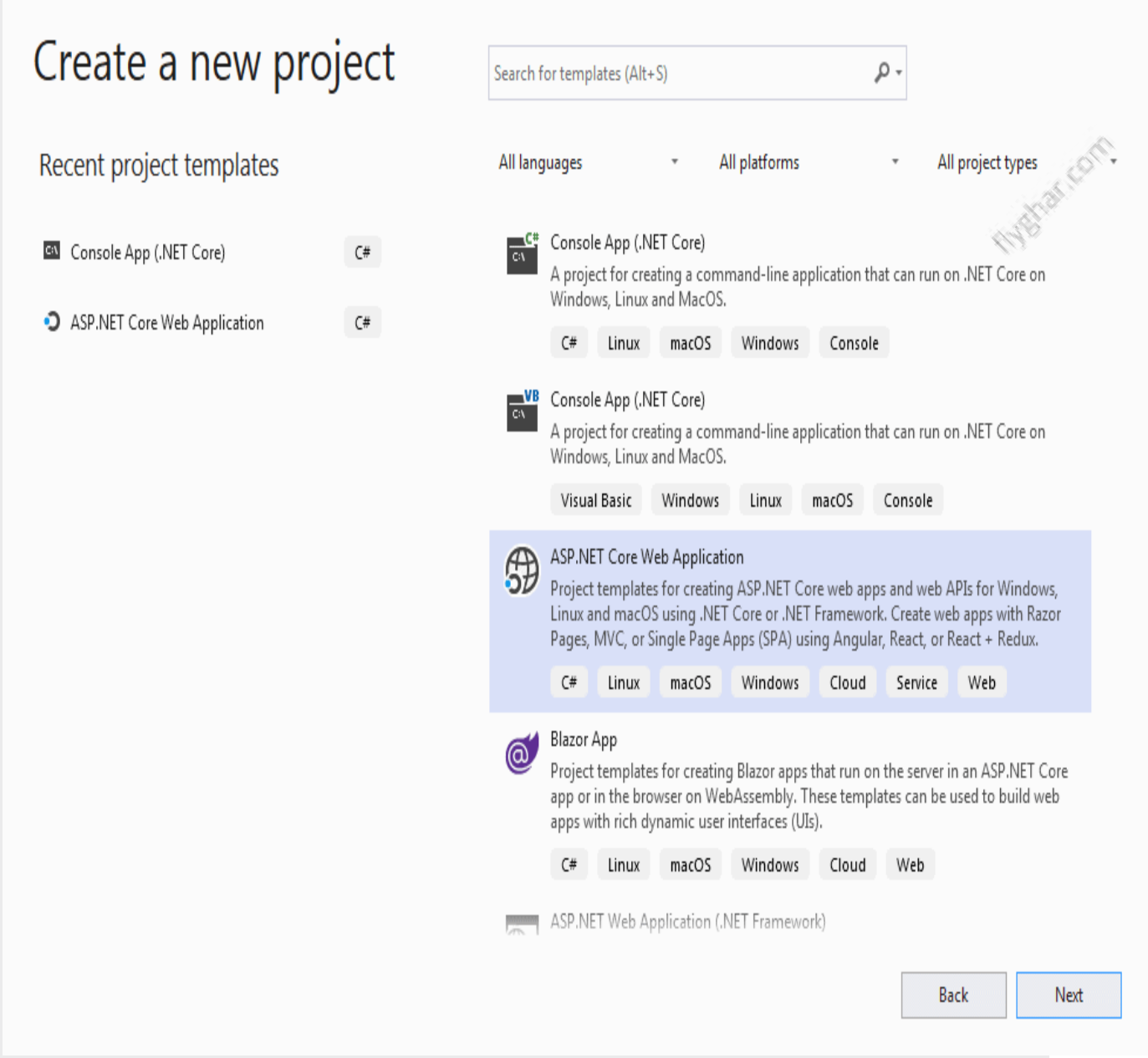
noteghar.com



Create a New ASP.NET Core 3.0 Project

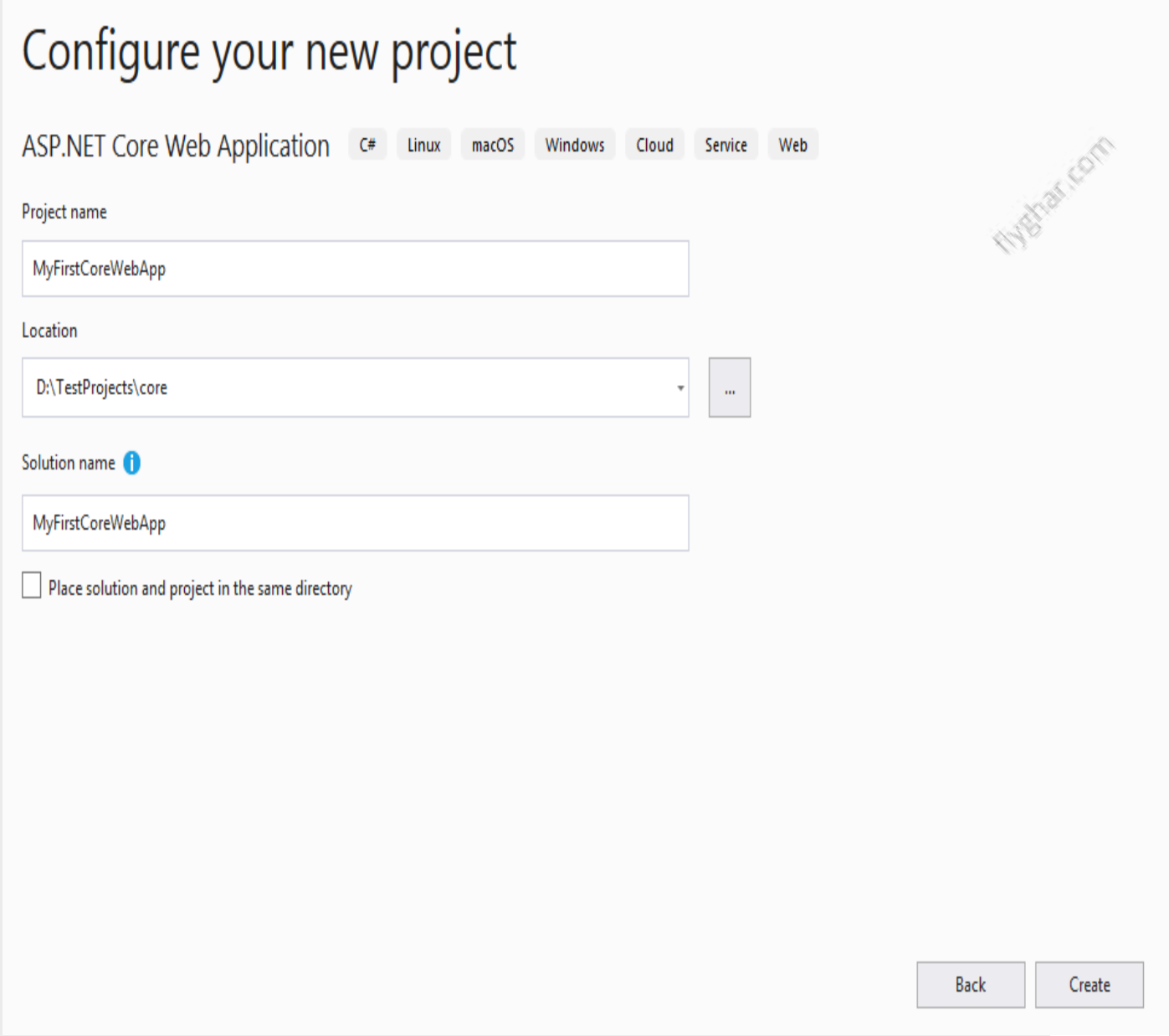
The "Create a new project" dialog box includes different .NET Core 3.0 application templates. Each will create predefined project files and folders depends on the application type. Here we will create a simple web application, so select **ASP.NET Core Web Application** template and click **Next**, as shown below.

noteghar.com



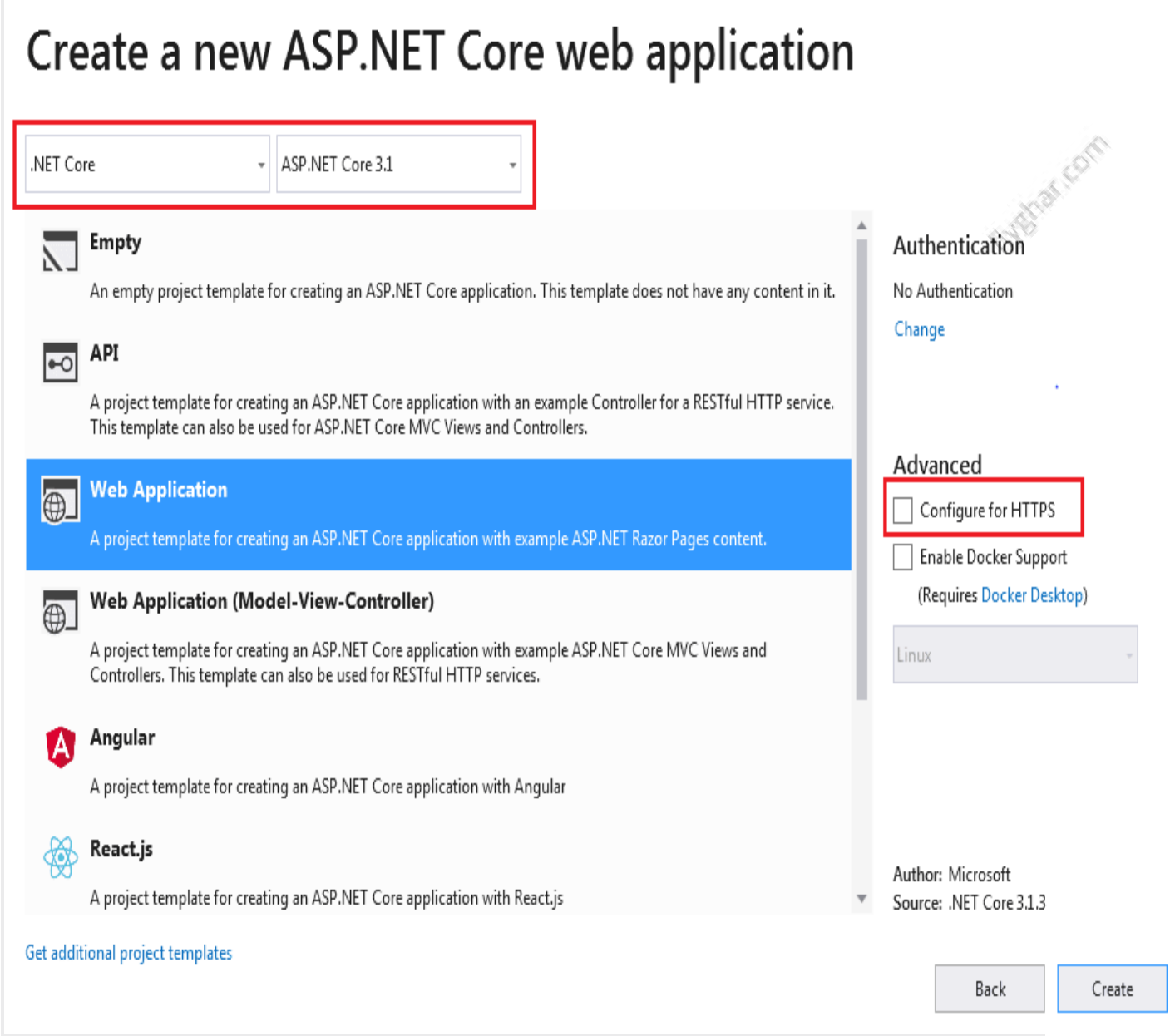
Select Application Template

Next, give the appropriate name, location, and the solution name for the ASP.NET Core application. In this example, we will give the name "MyFirstCoreWebApp" and click on the **Create** button, as shown below.



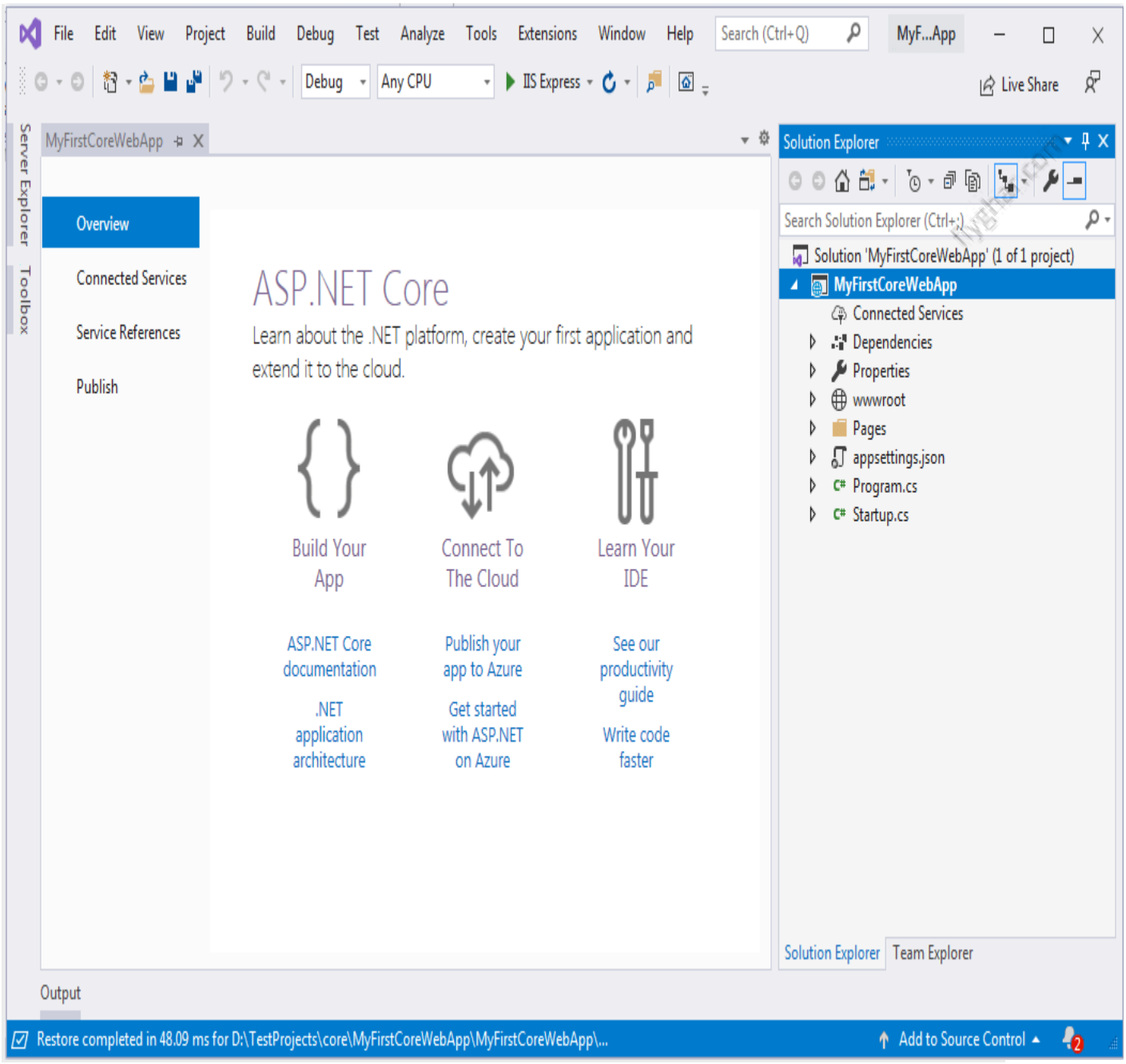
Configure Project

Next, select appropriate ASP.NET Core Web application template such as Empty, API, Web Application, MVC, etc. Here, we want to create a web application, so select the **Web Application** template. We don't want HTTPS at this point, so uncheck **Configure for HTTPS** checkbox, as shown below. Also, make sure you have selected the appropriate .NET Core and ASP.NET Core versions. Click on the **Create** button to create a project.



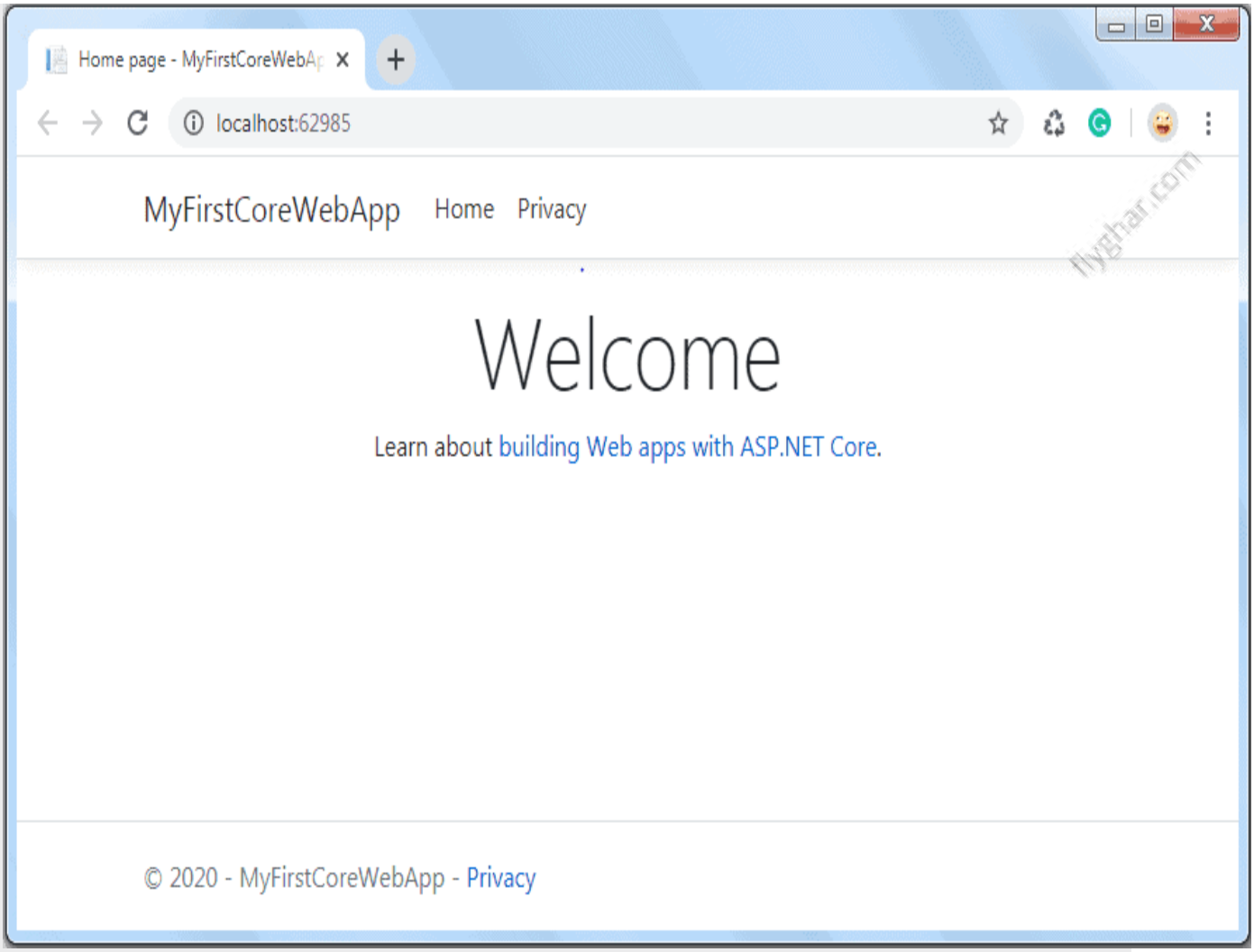
Select Web Application Template

This will create a new ASP.NET Core web project in Visual Studio 2019, as shown below. Wait for some time till Visual Studio restores the packages in the project. Restoring process means Visual Studio will automatically add, update or delete configured dependencies as NuGet packages in the project.



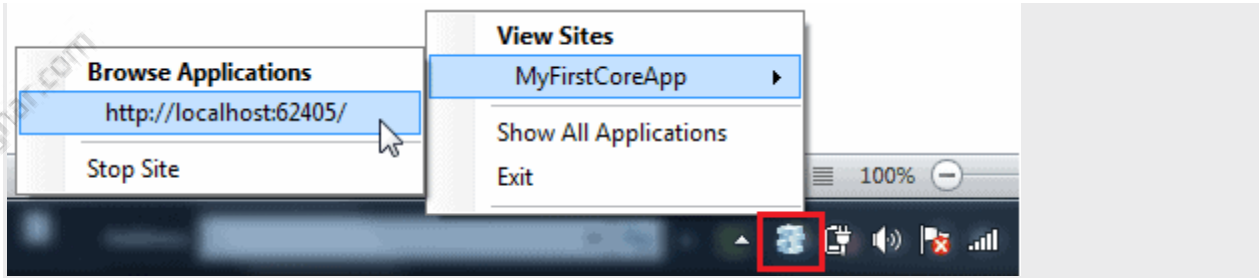
Create ASP.NET Core 3 Application

We will understand the project structure in the next chapter. To run this web application, click on **IIS Express** or press Ctrl + F5. This will open the browser and display the following result.



The above output comes from the `Index.cshtml` page under the **Pages** folder.

You can also see the IIS express icon on the system tray. Right click on it. You can see the ASP.NET sites currently running in your development machine.



ASP.NET Core sites in System Tray

Thus, we can create a new cross-platform ASP.NET Core 3.0 application that runs on .NET Core.

flyghar.com

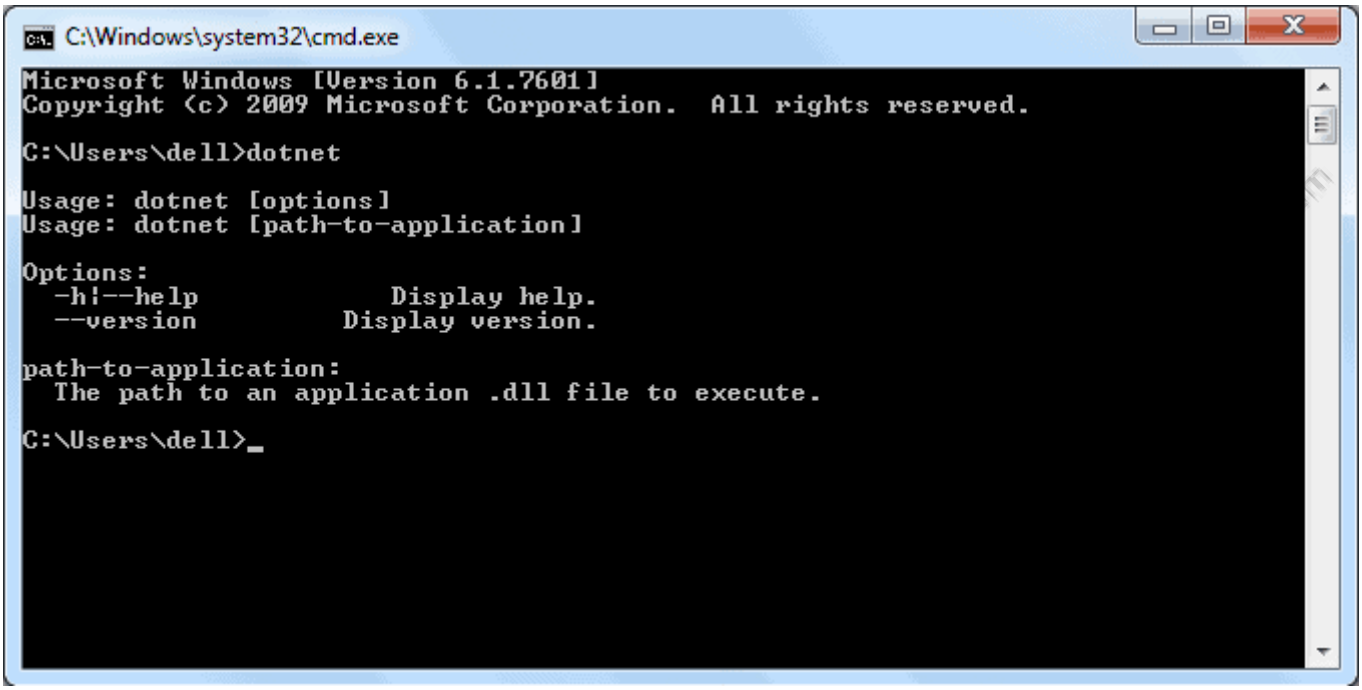
.NET Core Command-Line Interface

The .NET Core command-line interface (CLI) is a new cross-platform tool for creating, restoring packages, building, running and publishing .NET applications.

We created our first ASP.NET Core application using Visual Studio in the previous chapter. Visual Studio internally uses this CLI to restore, build and publish an application. Other higher level IDEs, editors and tools can use CLI to support .NET Core applications.

The .NET Core CLI is installed with .NET Core SDK for selected platforms. So we don't need to install it separately on the development machine. We can verify whether the CLI is installed properly by opening command prompt in Windows and writing dotnet and pressing Enter. If it displays usage and help as shown below then it means it is installed properly.

noteghar.com



.NET Core Command-line Interface

Command Structure

The following is a command structure.

```
dotnet <command> <argument> <option>
```

All the commands start with driver named dotnet. The driver starts the execution of the specified command. After dotnet, we can supply command (also known as verb) to perform a specific action. Each command can be followed by arguments and options. The following are .NET Core 2.x CLI commands.

Basic Commands	Description
new	Creates a new project, configuration file, or solution based on the specified template.
restore	Restores the dependencies and tools of a project.
build	Builds a project and all of its dependencies.
Run	Runs source code without any explicit compile or launch commands.
publish	Packs the application and its dependencies into a folder for deployment to a hosting system.

noteghar.com

Basic Commands	Description
test	Executes unit tests.
vtest	Runs tests from the specified files.
pack	Packs the code into a NuGet package.
clean	Cleans the output of a project.
sln	Modifies a .NET Core solution file.
help	Display help on the specified command
store	Stores the specified assemblies in the runtime package store.

Project Modification Commands	Description
add package	Adds a package reference to a project.
add reference	Adds project-to-project (P2P) references.
remove package	Removes package reference from the project.
remove reference	Removes project reference
list reference	Lists all project-to-project references

Advanced Commands	Description
nuget delete	Deletes or unlists a package from the server.
nuget locals	Clears or lists local NuGet resources.
nuget push	Pushes a package to the server and publishes it.
msbuild	Builds a project and all of its dependencies.
dotnet install script	Script used to install the .NET Core CLI tools and the shared runtime.

Let's create, restore, build, and run .NET Core console application using command-line interface without using Visual Studio.

ADVERTISEMENT

Create a New Project

To create a new .NET Core project, we have to use new command followed by template name argument. We can create console, class

library, web, mvc, webapi, razor, angular, react etc. projects using CLI. Use console template to create a new .NET Core console application.

The following creates new console project in the current directory with the same name as current directory.

```
dotnet new console
```

The following command creates a new console project named MyConsoleApp. The -n or --name option species the name of a project.

```
dotnet new console -n MyConsoleApp
```

The following command creates a new console application named MyConsoleApp to MyProjects directory. The -o or --output option is used to specify an output directory where the project should be generated.

```
dotnet new console -n MyConsoleApp -o C:\MyProjects
```

After creating a project, navigate to the project directories in command prompt to apply project specific commands which is C:\MyConsoleApp in our case.

Add Package Reference

We often need to add NuGet package reference for different purposes. For example, apply the following command to add Newtonsoft.json package to our console project.

```
C:\MyConsoleApp>dotnet add package Newtonsoft.json
```

This will add Newtonsoft.json package to our project. We can verify it by opening .csproj file.

Restore Packages

To restore packages or to update existing packages, we can use restore command as below.

```
C:\MyConsoleApp>dotnet restore
```

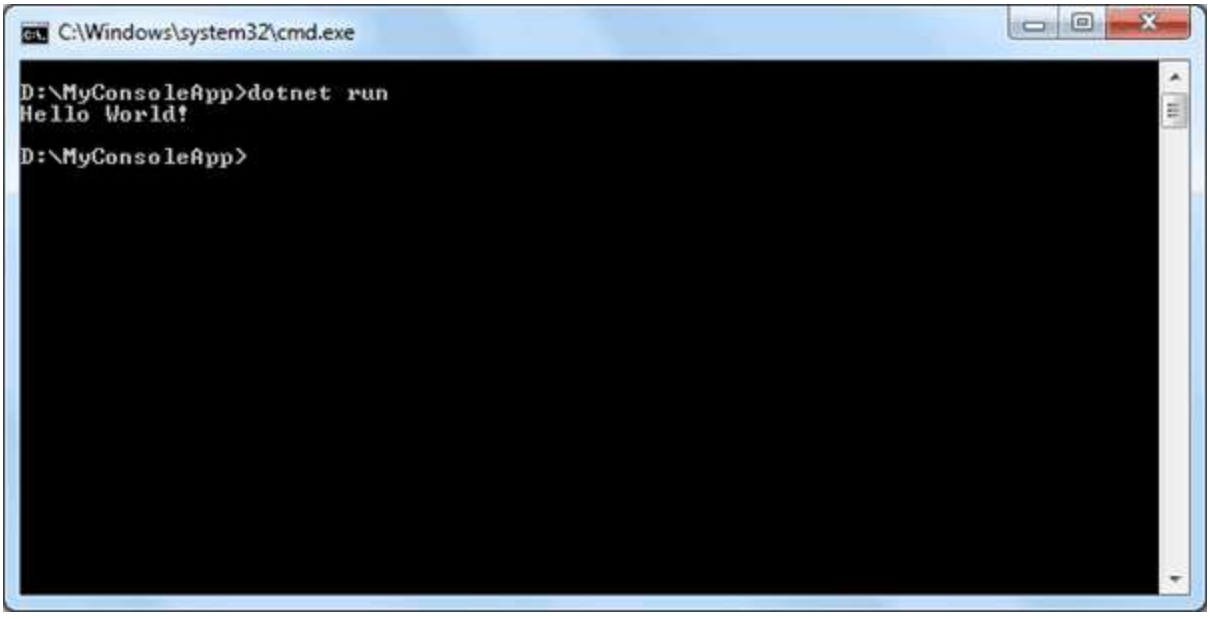
flyghar.com

Build Project

To build a new or existing project, apply `C:\MyConsoleApp>dotnet build` command.

Run project

To run our console project, apply `dotnet run` command as shown below.



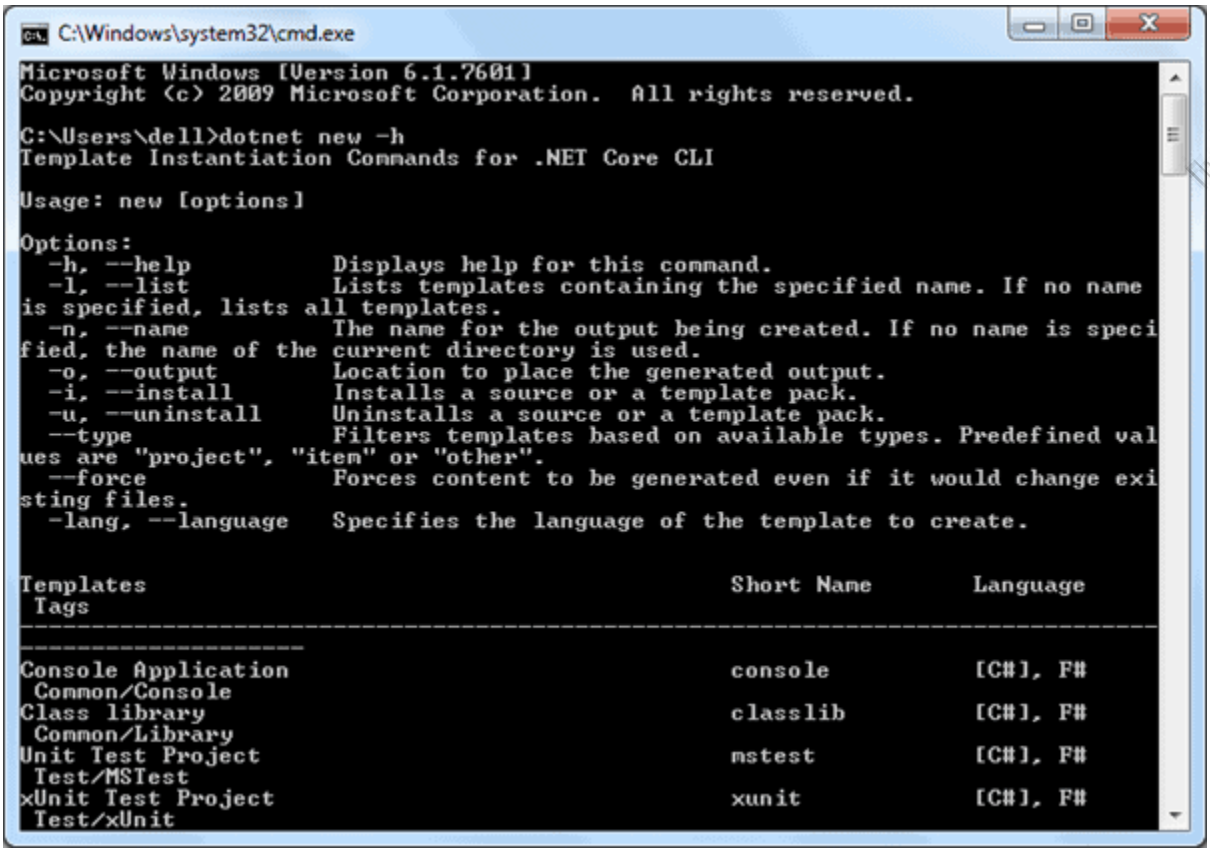
As you can see above, it displays an output "Hello World!".

noteghar.com

Getting Help

We can get help on any .NET Core CLI commands by typing -h or -help at the end of the command we want to get help on. For example,

dotnet new -h will display help on the new command, arguments and options we can use with it, as shown below.



Thus, we can use .NET Core command-line interface to create, restore packages, build, run, and publish different types of .NET Core applications.

noteghar.com

Introduction To .Net Framework

flyghar.com

- **.NET** is a software framework which is designed and developed by Microsoft and The first version of the .Net framework was 1.0 which came in the year 2002
- **.NET** stands for Network Enabled Technology. In .NET, dot (.) refers to object-oriented and NET refers to the internet. So the complete .NET means through object-oriented we can implement internet-based applications.
- **.NET** Framework supports more than 60 programming languages in which 11 programming languages are designed and developed by Microsoft. The remaining **Non-Microsoft Languages** which are supported by .NET Framework but not designed and developed by Microsoft.
- There is a variety of programming languages available on the .Net platform, VB.Net and C# being the most common ones.

noteghar.com

What does the DOTNET Framework provide?

flyghar.com

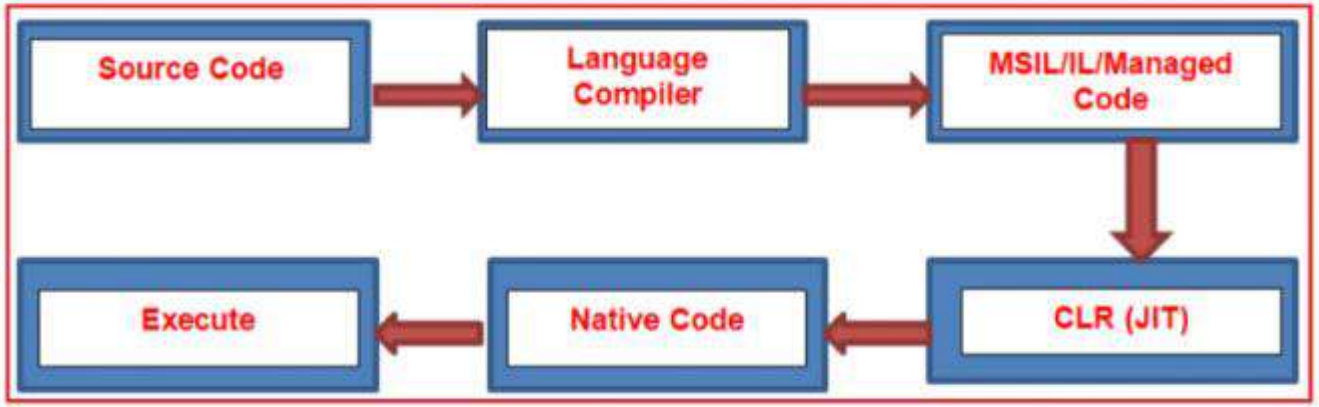
- The DOTNET Framework provides two things are as follows
    1. **BCL** (Base Class Libraries)
    2. **CLR** (Common Language Runtime)
- **BCL** (Base Class Libraries)

1. Base Class Libraries (BCL) is designed by Microsoft. Without BCL we can't write any code in .NET. So, BCL is also known as the Building block of .NET Programs.
2. The .NET Framework includes a set of standard class libraries. A class library is a collection of methods and functions that can be used for the core purpose.
3. These are installed into the machine when we installed the .NET framework. BCL contains pre-defined classes and these classes are used for the purpose of application development.
4. The physical location of BCL is **C:\Windows\assembly**
5. For example, there is a class library with methods to handle all file-level operations. So there is a method which can be used to read the text from a file. Similarly, there is a method to write text to a file.
6. The .NET base class library exists in order to encapsulate a huge number of common functions and makes them easily accessible to the developer.

noteghar.com

- **CLR (Common Language Runtime)**

1. It is the core component under the .NET framework which is responsible for converting the MSIL (Microsoft Intermediate Language) code into native code and then execution.

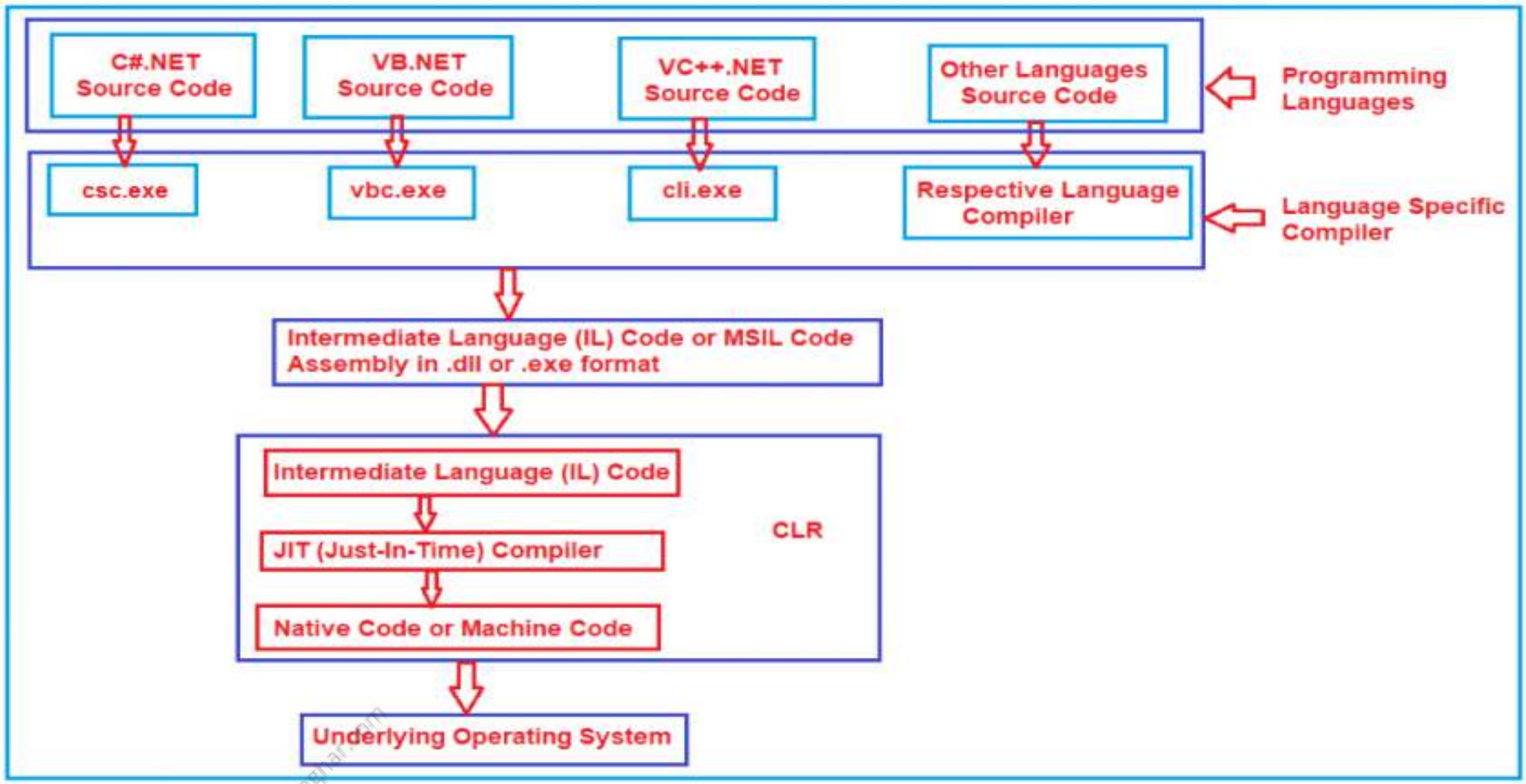


In the .NET framework, the code is compiled twice.

1. In the 1st compilation, the source code is compiled by respective language compiler and generates the intermediate code which is known as **MSIL (Microsoft Intermediate Language)** or **IL (Intermediate language code)** Or **Managed Code**.
2. In the 2nd compilation, **MSIL** is converted into **Native code** (native code means code specific to Operating system so that the code is executed by the Operating System) using **CLR**.

Always 1st compilation is slow and 2nd compilation is fast.

How a .NET application compiled and run?



- First, the developer has to write the code using any dot net supported programming languages such as C#, VB, J#, etc. Then the respective language compiler will compile the program into something called Intermediate language (IL) code. For example, if the programming language is C#, then the compiler is csc and if the programming language is VB, then the compiler will be vbc. This Intermediate Language (IL) code is half compiled code i.e. partially compiled code and cannot be executed directly by the operating system. So, when you want to execute this IL code on your machine, the dot net framework provides something called CLR or Common Language Runtime which takes the responsibility to execute your IL Code.
- The CLR takes the IL (Intermediate Language) code and gives it to something called JIT (Just-in-Time) Compiler. The JIT compiler takes the IL code and reads each and every line of the IL code and converts it to machine-specific instructions (i.e. into binary format) which can be executed by the underlying operating system.

Intermediate Language (IL) Code:

- IL code is a half compiled or partially compiled or CPU independent partially compiled code.

noteghar.com

CLR is the heart of .NET Framework and it contains the following components

1. Security Manager
2. JIT Compiler
3. Memory Manager
4. Garbage Collector
5. Exception Manager
6. Common Language Specification (CLS)
7. Common Type System (CTS)

1. Security Manager

1. CAS (Code Access Security)
2. CV (Code Verification)

These two components are basically used to check privileges of the current user that the user is allowed to access the assembly or not. The CLR also sees what kind of rights or what kind of authorities this code has and whether it is safe to be executed by the operating system. So, basically these types of checks are maintained by the Security Manager.

noteghar.com

2. JIT Compiler:

The JIT Compiler is responsible for Converting the MSIL code into native code that is executed by the machine. The native code is directly understandable by the system hardware. JIT compiles the code just before the execution and then save this translation in memory.

3. **Memory Manager:**

The memory manager component allocates the necessary memory for the variables and objects that are to be used by the application.

4. **Garbage Collector:**

When a dot net application runs, lots of objects are created. At a given point in time, it is possible that we don’t need these objects. So, Garbage Collector is nothing but it is a small routine or you can say it’s a background process thread that runs and try to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

5. **Exception Manager:**

This component redirects the processor to execute catch or finally blocks whenever an exception has occurred at runtime.

noteghar.com

6.Common Type System (CTS):

The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own data types. One programming language cannot understand other programming language data types. But, there can be situations when you want to code in one language to be called in other languages. In order to ensure smooth communication between these languages, the most important thing is that they should have a common type system (CTS) which ensures that type defined in two different languages gets compiled to a common data type.

CLR will execute all programming language's data types; this is possible because CLR will contain its own data types which are common to all programming languages.

At the time of compilation, all language-specific data types are converted into CLR's data type. This data type system of CLR which is common to all programming languages of .NET is known as the Common Type System (CTS).

noteghar.com

7. **CLS (Common Language Specification):**

CLS is a part of CLR. .NET supports many programming languages and every programming language has its own syntactical rules for writing the code which is known as language specification i.e. every programming language has its own language specification.

One programming language cannot understand other programming language syntactical rules (language specification). But CLR will execute all programming languages code. This is possible because CLR cannot understand any programming language specification rather CLR has its own language specification (syntactical rules) for its MSIL.

Every language compiler should follow this language specification of CLR at the time of compilation and should generate MSIL; CLR's JIT compiler will generate native code from MSIL. This language specification of CLR is common for all programming languages code execution of .NET and is known as CLS.

noteghar.com

Why Partial Compiled Code or why not fully compiled Code?

As a developer, you may thinking why partial compiled code or why not fully compiled code. The reason is very simple. We don’t know in what kind of environment .NET Code will run (for example, Windows XP, Windows 7, Windows 10, Windows Server, etc.). In other words, we don’t know what operating system is going to run our application; we also don’t know the CPU configuration, machine configuration, security configuration, etc. So, the IL code is partially compiled and at runtime, this IL code is compiled to machine-specific instructions using the environmental properties such as OS, CPU, Machine Configuration, etc.

noteghar.com

.NET Core vs. .NET Framework for server apps

flyghar.com

- There are two supported .NET implementations for building server-side apps: .NET Framework and .NET Core. Both share many of the same components and you can share code across the two. However, there are fundamental differences between the two and your choice depends on what you want to accomplish. This article provides guidance on when to use each.
- Use .NET Core for your server application when:
    1. You have cross-platform needs.
    2. You're targeting microservices.
    3. You're using Docker containers.
    4. You need high-performance and scalable systems.
    5. You need side-by-side .NET versions per application

- Use .NET Framework for your server application when:
    1. Your app currently uses .NET Framework
    2. Your app uses third-party .NET libraries or NuGet packages not available for .NET Core.
    3. Your app uses .NET technologies that aren't available for .NET Core.
    4. Your app uses a platform that doesn’t support .NET Core. Windows, macOS, and Linux support .NET Core.

noteghar.com

Memory Management

Garbage

Garbage consists of objects created during a program's execution on the managed heap that are no longer accessible by the program.

flyghar.com

Garbage Collector (GC)

The Garbage Collector (GC) is the part of the .NET Framework that allocates and releases memory for your .NET applications. The Common Language Runtime (CLR) manages allocation and deallocation of a managed object in memory. C# programmers never do this directly, there is no delete keyword in the C# language. It relies on the garbage collector.

The .NET objects are allocated to a region of memory termed the managed heap. They will be automatically destroyed by the garbage collector. Heap allocation only occurs when you are creating instances of classes. It eliminates the need for the programmer to manually delete objects that are no longer required for program execution. This reuse of memory helps reduce the amount of total memory that a program needs to run. Objects are allocated in the heap continuously, one after another. It is a very fast process, since it is just adding a value to a pointer.

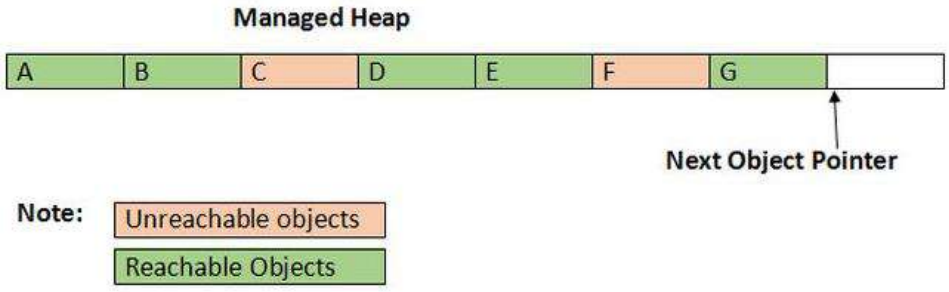
The process of releasing memory is called garbage collection. It releases only objects that are no longer being used in the application. A root is a storage location containing a reference to an object on the managed heap. The runtime will check objects on the managed heap to determine whether they are still reachable (in other words, rooted) by the application. The CLR builds an object graph, that represents each

noteghar.com

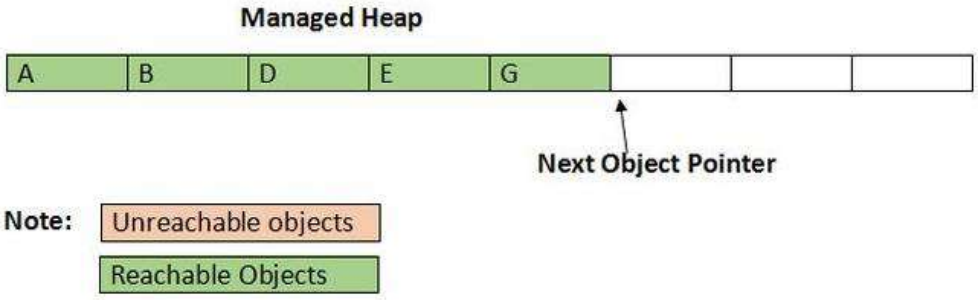
reachable object on the heap. Object graphs are used to document all reachable objects.

Example

Assume the managed heap contains a set of objects named A, B, C, D, E, F and G. During a garbage collection, these objects are examined for active roots. After the graph has been constructed, unreachable objects (that we will assume are objects C and F) are marked as garbage in reddish color in following diagram.



After the objects have been marked for termination, they are deleted from memory. At this point, the remaining space on the heap is compacted, that in turn causes the CLR to modify the set of active application roots to refer to the correct memory location. Last but not least, the next object pointer is readjusted to point to the next available slot.



flyghar.com

Object Generation

For better performance of memory releasing, the managed heap is divided into segments called “Generations”. There are only 3 generations: 0, 1 and 2. This idea of generations is simple: the longer an object has existed on the heap are reachable objects. Objects that have only recently been placed on the heap are unreachable.

1.When objects are just created, they are placed to the Generation 0 (Gen0).

2.When Gen 0 is full, the GC performs a garbage collection. During the collection, the GC removes all unreachable objects from the heap. All reachable objects are promoted to the Generation 1 (Gen 1). The Gen 0 collection is a rather quick operation.

noteghar.com

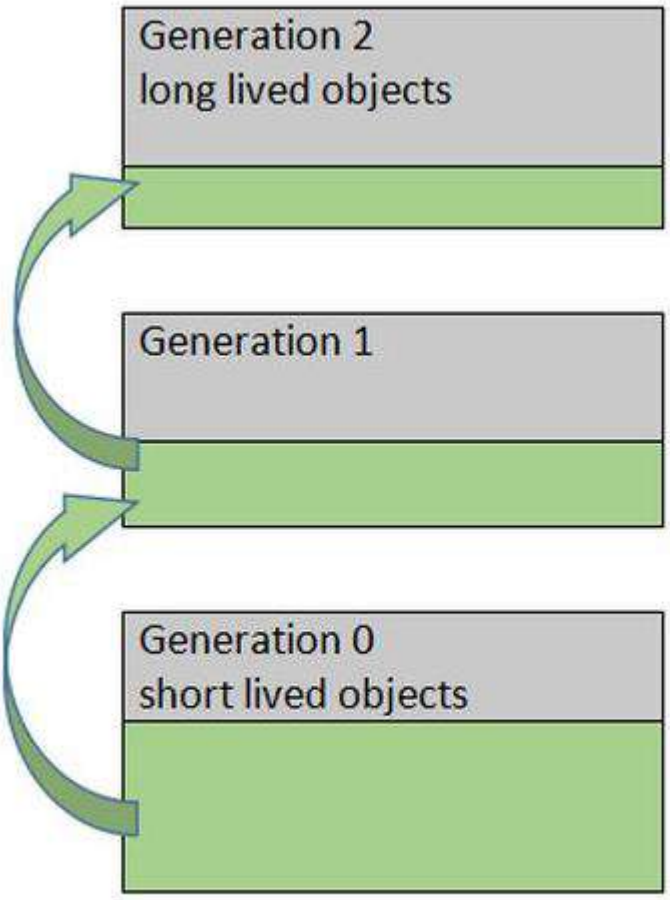
3.When Gen 1 is full, the Gen 1 garbage collection is performed. All objects that survive the collection are promoted to Gen 2. The Gen 0 collection also takes place here.

4.When Gen 2 is full, the GC performs full garbage collection. First, Gen 2 collection is performed, then the Gen 1 and Gen 0 collections take place. If there is still not enough memory for new allocations, the GC raises the OutOfMemory exception.

5.During a full garbage collection, the GC must pass through all objects in the heap, so this process might have a great impact on system resources.

The following diagram is a short explanation of generations work:

generation



noteghar.com
flyghar.com

Attempt any two Questions (2 X 10 = 20)

1. Define Exception Handling with an example.Explain method overloading with an example.[2.5+2.5+5]

Write a Program to create class Person with following specifications.

a. Field Members : Name , Address , PhoneNumber , CitizenshipNo

b. Default constructor , Parameterized constructor

c. Mehod: Display() method for showing details of Person

d. Create ChangeName(string name) method to change name of the person

e. Now create object of this class and call display method and changeName method

2. Discuss about .Net Core Characteristics .Explain Razor Syntax with example . How does ASP.NET Core MVC identify classes that can act as controllers ?[4 + 4 + 2]

3. Differentiate between DataSet and DataReader . Explain different ADO.NET Classes ?[5 + 5]

Attempt any eight Questions (5 X 8 = 40)

1. Explain compilation and execution of .Net Application?[5]

2. Explain Delegate and Event with example ?[2.5+2.5]

3. Expain different common web application Architecture?[5]

4. What is Tag Helper in ASP.NET Core.What are the various Json file available in ASP.NET Core?[1 + 4]

5. Describe about CRUD operation in ASP.NET Core Using Entity Framework ?[5]

6. Define State Management ? How can we maintain state using Temp Data Explain with an example ? [1 + 4]

7. Write about Angular . How can we handle events with Jquery mehtods ?[2 + 3]

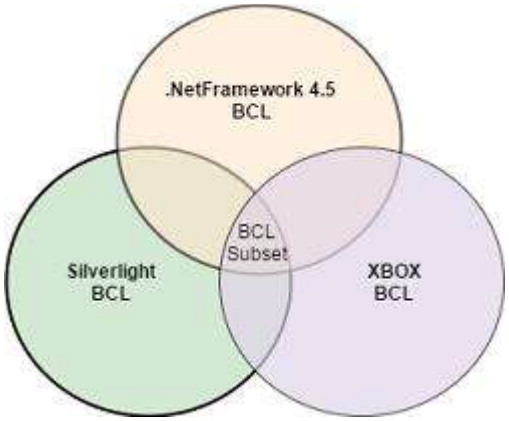
8. Define .Net Core Identity ? Explain CSRF Attack with Prevention methods ? [1 + 4]

9. Explain about the hosting models in ASP.NET Core ? [5]

noteghar.com
flyghar.com

What is Portable Class Library?

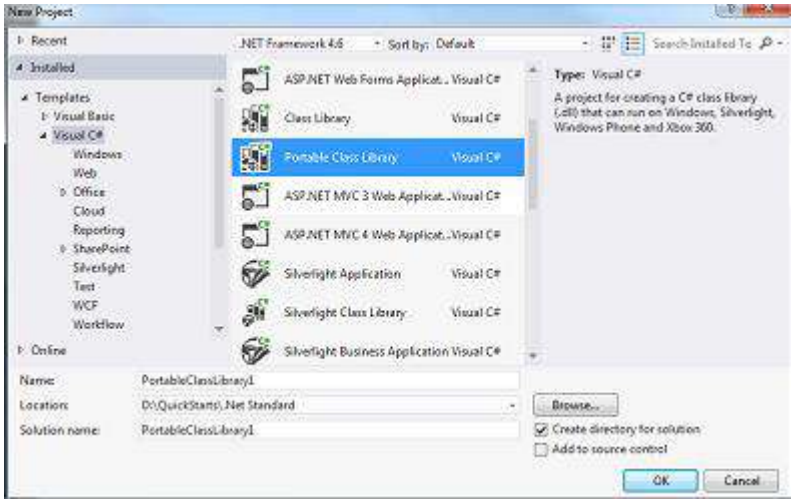
Portable Class Libraries are introduced to solve this code sharing problems. It is a re-usable library project where we specify the .Net Platforms we are planning to target during creation of library. This project has allowed developers to build portable libraries which exposed only those Base Class Libraries (or API's) that are common between the selected platforms. The Visual Studio tools will manage this for us. For example, if you select the target platforms as .NetFramework and Silverlight, the library can then use only those API's which are commonly available in both of this platforms. Refer the below Venn diagram,



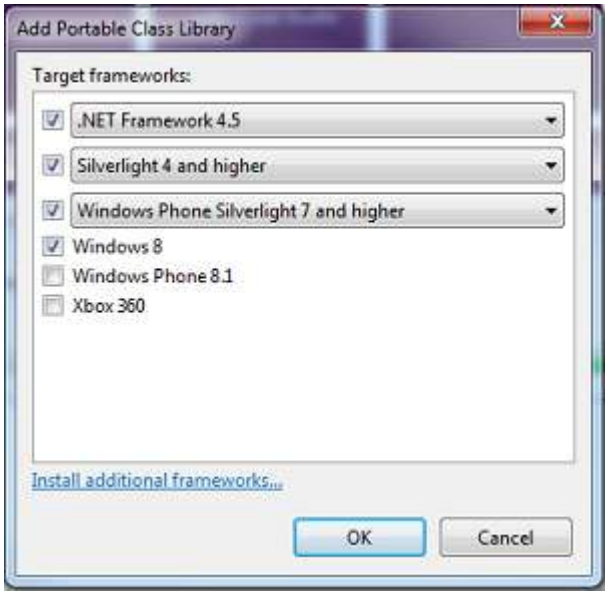
Assuming, we are creating a portable class library project that targets .Net 4.5, Silverlight and Xbox platforms, the library project can use only the common libraries (BCL subset in Venn diagram) that are obtained by intersection of all platforms BCL. So, higher the number of platforms then lower the surface (number) of common API.

In Visual Studio 2012, we have a project type called Portable Class Library project for this. Refer below.

noteghar.com



Creating this project will ask you to select target platforms the library is targeting like below.



The above selection will create a portable library class that allows us to consume only the API's that are common between .NetFramework 4.5.Silverlight 4, Windows Phone Silverlight 7 and Windows 8.

Though the portable libraries allowed code sharing between multiple platforms, it still had disadvantages like, adding a new platform support requires re-compiling against the new set of target platforms and removing the API references that are not part of new intersection of base class libraries. Portable Class Libraries also does not have any control on the target platforms which were still evolving separately.

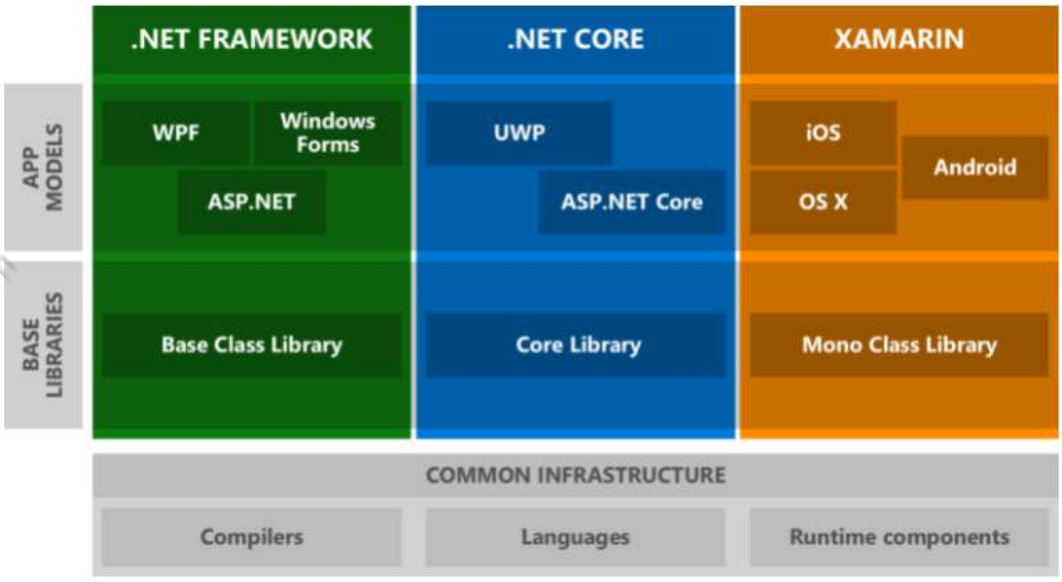
.Net Standard

.Net Standard is a specification which dictates what the Base Class Libraries of different .Net platforms should implement to unify the Base Class Libraries of different .Net Platforms. Here, Platform means full .NetFramework, .Net Core, Xamarin, Silverlight, XBox etc. This also enables code sharing between applications that runs on these different platforms. For example, a library or a component that is developed on top of a platform that implements specific .Net Standard version can be shared by all the applications that runs on any of the platforms that implements same .Net Standard version.

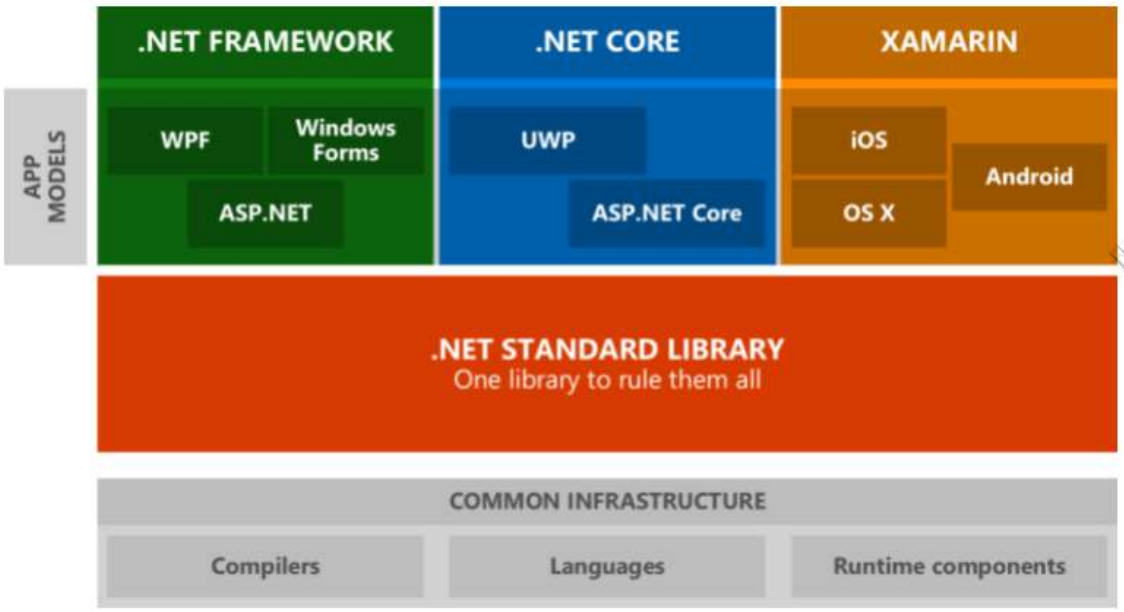
Is it a Framework of its own?

.Net Standard is not a framework or platform of its own. It does not have implementations or a runtime, it just defines a specification what different .Net platforms has to implement to remain .Net Standard compliant. For Instance, .Net Core is a framework with runtime that implements .NetStandard.

Understanding .NET Framework, .NET Core, and Xamarin



Understanding .NET Standard



.Net Standard 2.0

It is a new version that significantly increases the number of APIs compared to the previous version(1.6.1). In fact , the API surface has more than doubled with .Net Standard 2.0.

The .Net standard 2.0 is supported by the following .Net Implementation :

- .NET Core 2.0 or later
- .Net Framework 4.6.1 or later
- Mono 5.4 or later
- Xamarin.Ios 10.14 or later
- Xamarin.Mac 3.8 or later
- Xamarin.Android 8.0 or later
- Universal Windows Platform 10.0.16299 or later

noteghar.com

What's new in the .Net Standard 2.0?

1. A vastly expanded set of APIs
2. Support fore .NET Framework libraries
3. Support for visual Basic
4. Tooling Support for .NET Standard libraries

tiyeghar.com

noteghar.com

Applied Technologies

Following are the different applied technology in .Net

1.WinForm

Windows Forms (WinForms) is a free and open-source graphical (GUI) class library included as a part of Microsoft .NET Framework providing a platform to write rich client applications for desktop, laptop, and tablet PCs. Largely, WinForms allows the developer to drag and drop controls on a Windows Form and allows the developer to manipulate these controls with a code-behind file that can have C#, VB.NET, or any other .NET language.

2.ASP.NET

ASP.NET extends the .NET platform with tools and libraries specifically for building web apps.

These are some things that ASP.NET adds to the .NET platform:

- **Base framework for processing web requests in C# or F#**
- **Web-page templating syntax**, known as Razor, for building dynamic web pages using C#
- **Libraries for common web patterns**, such as Model View Controller (MVC)
- **Authentication system** that includes libraries, a database, and template pages for handling logins, including multi-factor authentication and external authentication with Google, Twitter, and more.

noteghar.com
flyghar.com

3.ADO.NET

ADO.NET is a module of .Net Framework, which is used to establish a connection between application and data sources. Data sources can be such as SQL Server and XML. ADO .NET consists of classes that can be used to connect, retrieve, insert, and delete data.

tiyeghar.com

4.Entity Framework

It is an ORM based open source framework which is used to work with a database using .NET objects. It eliminates a lot of developers effort to handle the database. It is Microsoft's recommended technology to deal with the database.

5.LINQ (Language Integrated Query)

It is a query language, introduced in .NET 3.5 framework. It is used to make the query for data sources with C# or Visual Basics programming languages.

6.Uinversal Window Platform(UWP)

UWP is an application development platform created by Microsoft. A UWP app can run on multiple devices, including Windows PCs, tablets, and smartphones. Some UWP apps can run on other types of Microsoft hardware including Xbox, HoloLens, and IoT devices.

noteghar.com

What is object-oriented programming (OOP)?

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. For example, a person is an object which has certain properties such as height, gender, age, etc. It also has certain methods such as move, talk, and so on.

flyghar.com

Class

A class in C# is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C# class is like a blueprint for an object.

For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage* range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

noteghar.com

Object

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. This is the basic unit of object-oriented

programming. That is both data and function that operate on data are bundled as a unit called an object.

OOP has four basic concepts on which it is totally based. Let's have a look at them individually –

1. Abstraction

Abstraction is the concept of object-oriented programming that "shows" only essential attributes and "hides" unnecessary information. The main purpose of abstraction is hiding the unnecessary details from the users. Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts. It is one of the most important concepts of OOPs.

2. **Encapsulation**

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of bindling data and methods that work on that data within one unit, e.g., a class in C#. Encapsulation is implemented by using **access specifiers**

This concept is also often used to hide the internal representation, or state, of an object from the outside. This is called information hiding. The general idea of this mechanism is simple. If you have an attribute that is not visible from the outside of an object, and bind it with methods that provide read or write access to it, then you can hide specific information and control access to the internal state of the object.

3. **Inheritance**

The ability to create a new class from an existing class is called Inheritance. Using inheritance, we can create a Child class from a Parent class such that it inherits the properties and methods of the parent class and can have its own additional properties and

methods. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.

4. Polymorphism

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. In C# polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Object Based Language

In a programming sense, the term "object-based language" may be used to describe any programming language that is based on the idea of encapsulating state and operations inside "objects". Object-based languages need not support inheritance or polymorphism . While Object oriented language support all feature of OOPS i.e. Inheritance, polymorphism etc. In Object Oriented Language there is no built in type of objects are available but in Object Based Language Build in type object are available like window object in javascript.

Object based Programming	Object oriented Programming
• Object-based language doesn't support all the features of OOPs like Polymorphism and Inheritance. • Object-based language has built-in object like JavaScript has window object. • Examples : JavaScript, VB etc.	• Object-oriented language supports all the features of OOPs . • Object-oriented language doesn't have built-in object. • Examples : C++, C#, Java etc.

noteghar.com

Why Javascript isnot an Object Oriented Programming Languag?

JavaScript is an Object Based Language, it can build actual objects from a constructor function and it has almost any feature that any object could have such as Constructor, functions (Methods), Properties, Instances etc. But, Javascript is not an Object Oriented Programming Language because it has no feature that fits the requirements of the definition of Object Oriented Programming Language such as Inheritance, Polymorphism etc.

flyghar.com

Difference Between Procedure Oriented Programming and Object Oriented Programming

Procedural Oriented Programming	Object Oriented Programming
In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc.
Procedural programming does not have any proper way for hiding data so it is ***less secure***.	Object oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
Procedural programming is based on ***unreal world***.	Object oriented programming is based on ***real world***.
Examples: C, FORTRAN, Pascal, Basic etc.	Examples: C++, Java, Python, C# etc.

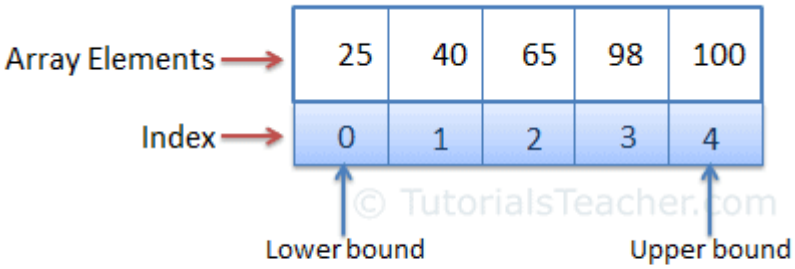
noteghar.com

Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type stored at contiguous memory locations.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Array Representation

C# Arrays Declaration

In c#, Arrays can be declared by specifying the type of elements followed by the square brackets [] like as shown below.

```
type[] array_name;
```

Here the type is nothing but a data type of elements to store in an array and array_name represents the name of an array.

For example, the following are the different ways of declaring an array with different data types in c# programming language.

```
// Store only int values
int[] numbers;
//Store only string values
string[] names;
//Store only double values
double[] ranges;
```

If you observe the above examples, we declared arrays with the required data type based on our requirements.

In c#, the array elements can be of any type and by default, the values of numeric array elements are set to zero and the reference elements are set to null.

flyghar.com

C# Arrays Initialization

In c#, Arrays can be initialized by creating an instance of an array with a **new** keyword. By using a **new** keyword we can declare and initialize an array at the same time based on our requirements.

Following are the different ways of declaring and initializing array elements by using the **new** keyword in c# programming language.

```
// Declaring and Initializing an array with size of 5
int[] array = new int[5];
//Defining and assigning an elements at the same time
int[] array2 = new int[5]{1,2,3,4,5};
//Initialize with 5 elements will indicates the size of an array
int[] array3 = new int[] { 1, 2, 3, 4, 5 };
// Another way to initialize an array without size
int[] array4 = { 1, 2, 3, 4, 5 };
// Declare an array without initialization
int[] array5;
array5 = new int[]{ 1, 2, 3, 4, 5 };
```

If you observe above examples, in first statement we declared and initialized an integer array with the size of **5** to allow an array to store **5** integer values and the array can contain an elements from **array[0]** to **array[4]**.

In the second statement, we declared and initialized an array same as the first statement but also assigned values to each index followed by curly brackets **{ }**.

noteghar.com

In a third or fourth statement, while declaration we initialized an array with values, but without specifying any size. Here, the size of an array can be determined by the number of elements so the size initializer is not required if we are assigning elements during the initialization.

In c#, we can declare an array variable without initialization, but we must use the **new** keyword to assign an array to the variable.

In the fifth statement, we declared an array without initialization and we used a **new** keyword to assign array values to the variable.

In c#, after an array declaration, we can initialize array elements using index values. Following is an example of declaring and initializing array elements using individual index values in c#.

tiyghar.com

```
int[] array = new int[5];
array[0] = 1;
array[1] = 2;
array[2] = 3;
array[3] = 4;
array[4] = 5;
```

If you observe above example, we are initializing an array elements individual using individual index values.

Generally, in c# initializing an array without **size** or assigning values to an array without a **new** operator will throw compile-time errors. For example:

```
// Error. Initialize an array without size
int[] array = new int[];
// Error. Initialize an array without new keyword
int[] array1;
array1 = { 1, 2, 3, 4, 5 };
```

If you observe the above examples, in the first statement we initialized an array without any **size** and in the second statement, we declared and initializing array elements without using the **new** keyword. These two statements will throw compile-time errors in our c# applications.

C# Accessing an Array Elements

In c#, we can access array elements by using for loop or foreach loop or with particular index numbers.

noteghar.com

Following is the code snippet of accessing array elements by using particular index numbers.

```
int[] array = new int[5] { 1, 2, 3, 4, 5 };
int a = array[1]; // It will return 2
int b = array[4]; // It will return 5
```

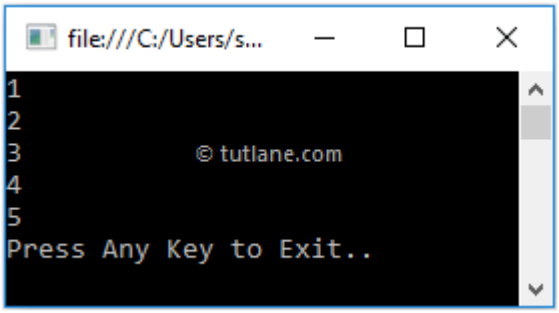
If you observe above code, we are trying to access an array elements using index values in c#.

Following is the example of declaring, initializing and accessing array elements with particular index numbers in c# programming language.

```
using System;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = new int[5] { 1, 2, 3, 4, 5 };
            Console.WriteLine(array[0]);
            Console.WriteLine(array[1]);
            Console.WriteLine(array[2]);
            Console.WriteLine(array[3]);
            Console.WriteLine(array[4]);
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we declared and initialized an array with 5 elements and we are accessing an array elements using index values.

When we execute the above c# program, we will get the result as shown below.



If you observe the above result, we are able to access array elements using index numbers based on our requirements.

C# Access Array Elements with For Loop

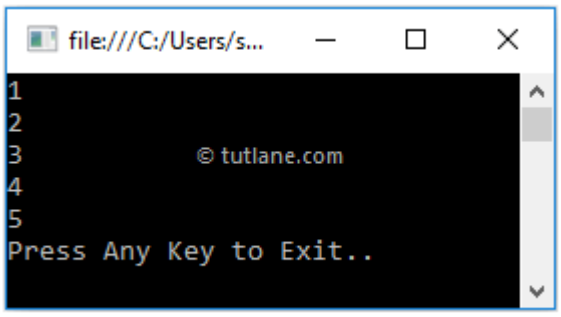
In c#, by using for loop we can iterate through array elements and access the values of an array with length property.

Following is the example of accessing array elements using for loop in c# programming language.

```
using System;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = new int[5] { 1, 2, 3, 4, 5 };
            for (int i = 0; i < array.Length; i++)
            {
                Console.WriteLine(array[i]);
            }
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we looping through an array elements with for loop to access an array elements based on our requirements.

When we execute the above c# program, we will get the result as shown below.



If you observe the above result, we are able to loop through elements in an array with for loop and print array values based on our requirements.

C# Access Array Elements with Foreach Loop

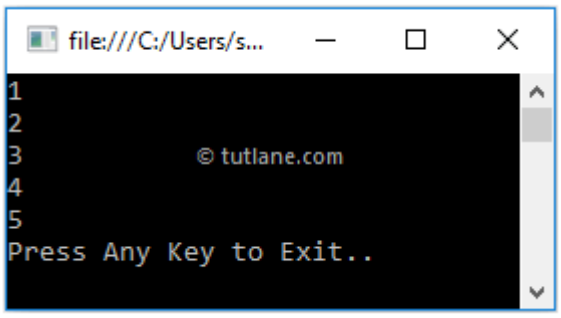
In c#, same as for loop we can use the foreach loop to iterate through array elements and access the values of an array based on our requirements.

Following is the example of accessing array elements using a foreach loop in c# programming language.

```
using System;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = new int[5] { 1, 2, 3, 4, 5 };
            foreach(int i in array)
            {
                Console.WriteLine(i);
            }
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we looping through an array elements with foreach loop to access an array elements based on our requirements.

When we execute the above c# program, we will get the result as shown below.



If you observe the above result, we are able to loop through elements in an array with foreach loop and print array values based on our requirements.

This is how we can access array elements in c# programming language based on our requirements.

C# Array Types

In c#, There are 3 type of arrays available, those are

1. Single-Dimensional Arrays
2. Multi-Dimensional Arrays
3. Jagged Arrays

flyghar.com

1. Single-Dimensional Arrays

In this array contains only one row for storing the values. All values of this array are stored contiguously starting from 0 to the array size. For example, declaring a single-dimensional array of 5 integers :

```
int[] arrayint = new int[5];
```

The above array contains the elements from arrayint[0] to arrayint[4]. Here, the new operator has to create the array and also initialize its element by their default values. Above example, all elements are initialized by zero, Because it is the int type.

Example :

```
// C# program to creating an array
// of the string as week days, store
// day values in the weekdays,
// and prints each value.
using System;
namespace geeksforgeeks {

class GFG {

    // Main Method
    public static void Main()
    {

        // declares a 1D Array of string.
        string[] weekDays;

        // allocating memory for days.
        weekDays = new string[] {"Sun", "Mon", "Tue", "Wed",
                                 "Thu", "Fri", "Sat"};

        // Displaying Elements of array
        foreach(string day in weekDays)
```

noteghar.com

```
            Console.Write(day + " ");
        }
    }
    }
```

Output :

```
Sun Mon Tue Wed Thu Fri Sat
```

2. Multi-Dimensional Arrays

C# supports multidimensional arrays up to 32 dimensions. The multidimensional array can be declared by adding commas in the square brackets. For example, [,] declares two-dimensional array, [, ,] declares three-dimensional array, [, , ,] declares four-dimensional array, and so on. So, in a multidimensional array, no of commas = No of Dimensions - 1.

The following declares multidimensional arrays.

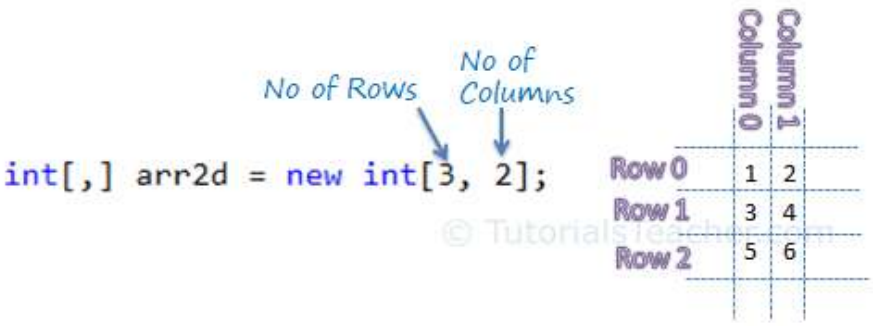
Example: Multidimensional Arrays

```
int[,] arr2d; // two-dimensional array
int[, ,] arr3d; // three-dimensional array
int[, , ,] arr4d ; // four-dimensional array
int[, , , ,] arr5d; // five-dimensional array
```

Let's understand the two-dimensional array. The following initializes the two-dimensional array.

Example: two-dimensional Array

```
int[,] arr2d = new int[3,2]{
                                {1, 2},
                                {3, 4},
                                {5, 6}
                            };

// or
int[,] arr2d = {
                    {1, 2},
                    {3, 4},
                    {5, 6}
                };
```



Two-dimensional Array

The following access values of the two-dimensional array.

Example: Access two-dimensional Array

```
int[,] arr2d = new int[3,2]{
                                {1, 2},
                                {3, 4},
                                {5, 6}
                            };

arr2d[0, 0]; //returns 1
arr2d[0, 1]; //returns 2
arr2d[1, 0]; //returns 3
arr2d[1, 1]; //returns 4
arr2d[2, 0]; //returns 5
arr2d[2, 1]; //returns 6
//arr2d[3, 0]; //throws run-time error as there is no 4th row
```

In the above example, the value of a two-dimensional array can be accessed by index no of row and column as [row index, column index]. So, `[0, 0]` returns the value of the first row and first column and `[1, 1]` returns the value from the second row and second column.

Now, let's understand the three-dimensional array. The following declares and initializes three-dimensional arrays.

Example: Three-dimensional Array

```
int[, ,] arr3d1 = new int[1, 2, 2]{
                { { 1, 2}, { 3, 4} }
            };

int[, ,] arr3d2 = new int[2, 2, 2]{
                { {1, 2}, {3, 4} },
                { {5, 6}, {7, 8} }
            };

int[, ,] arr3d3 = new int[2, 2, 3]{
                { { 1, 2, 3}, {4, 5, 6} },
                { { 7, 8, 9}, {10, 11, 12} }
            };

arr3d2[0, 0, 0]; // returns 1
arr3d2[0, 0, 1]; // returns 2
arr3d2[0, 1, 0]; // returns 3
arr3d2[0, 1, 1]; // returns 4
arr3d2[1, 0, 0]; // returns 5
arr3d2[1, 0, 1]; // returns 6
arr3d2[1, 1, 0]; // returns 7
arr3d2[1, 1, 1]; // returns 8
```

flyghar.com

As you can see in the above example, [1, 2, 2] of `arr3d1` specifies that it will contain one row of two-dimensional array [2, 2]. `arr3d2` specifies dimensions [2, 2, 2], which indicates that it includes two rows of two-dimensional array of [2, 2]. Thus, the first rank indicates the number of rows of inner two-dimensional arrays.

Now, consider the following four-dimensional array.

noteghar.com

```
int[,,,] arr4d1 = new int[1, 1, 2, 2]{
                {
                    { { 1, 2}, { 3, 4} }
                }
            };

arr4d1[0, 0, 0, 0]; // returns 1
arr4d1[0, 0, 0, 1]; // returns 2
arr4d1[0, 0, 1, 0]; // returns 3
arr4d1[0, 0, 1, 1]; // returns 4

int[,,,] arr4d2 = new int[1, 2, 2, 2]{
                {
                        { {1, 2}, {3, 4} },
                        { {5, 6}, {7, 8} }
                }
        };

arr4d2[0, 0, 0, 0]; // returns 1
arr4d2[0, 0, 0, 1]; // returns 2
arr4d2[0, 0, 1, 0]; // returns 3
arr4d2[0, 0, 1, 1]; // returns 4
arr4d2[0, 1, 0, 0]; // returns 5
arr4d2[0, 1, 0, 1]; // returns 6
arr4d2[0, 1, 1, 0]; // returns 7
arr4d2[0, 1, 1, 1]; // returns 8
```

flyghar.com

In the above example, the four-dimensional array `arr4d1` specifies [1, 1, 2, 2], which indicates that it includes one row of the three-dimensional array.

In the same way, you can declare and initialize five-dimensional, six-dimensional array, and up to 32-dimensional arrays in C#.

3. Jagged Arrays: An Array of Array

noteghar.com

A jagged array is an array of array. Jagged arrays store arrays instead of literal values.

A jagged array is initialized with two square brackets [][]. The first bracket specifies the size of an array, and the second bracket specifies the dimensions of the array which is going to be stored.

The following example declares jagged arrays.

Example: Jagged Arrays

```
int[][] jArray1 = new int[2][]; // can include two single-dimensional arrays
int[][,] jArray2 = new int[3][,]; // can include three two-dimensional arrays
```

In the above example, jArray1 can store up to two single-dimensional arrays. jArray2 can store up to three two-dimensional, arrays [,] specifies the two-dimensional array.

Example: Jagged Array

```
int[][] jArray = new int[2][];

jArray[0] = new int[3]{1, 2, 3};

jArray[1] = new int[4]{4, 5, 6, 7 };
```

You can also initialize a jagged array upon declaration like the below.

Example: Jagged Array

```
int[][] jArray = new int[2][]{
                new int[3]{1, 2, 3},

                new int[4]{4, 5, 6, 7}
            };

jArray[0][0]; //returns 1
jArray[0][1]; //returns 2
jArray[0][2]; //returns 3
jArray[1][0]; //returns 4
jArray[1][1]; //returns 5
jArray[1][2]; //returns 6
jArray[1][3]; //returns 7
```

You can access a jagged array using two for loops, as shown below.

flyghar.com
noteghar.com

Example: Jagged Array

```
int[][] jArray = new int[2][]{
                new int[3]{1, 2, 3},

                new int[4]{4, 5, 6, 7}
            };

for(int i=0; i<jArray.Length; i++)
{
        for(int j=0; j < (jArray[i]).Length; j++)
                Console.WriteLine(jArray[i][j]);
}
```

flyghar.com

The following jagged array stores two-dimensional arrays where the second bracket [,] indicates the two-dimensional array.

Example: Jagged Array

```
int[][,] jArray = new int[2][,];

jArray[0] = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };
jArray[1] = new int[2, 2] { { 7, 8 }, { 9, 10 } };

jArray[0][1, 1]; //returns 4

jArray[1][1, 0]; //returns 9

jArray[1][1, 1]; //returns 10
```

If you add one more bracket then it will be array of array of array.

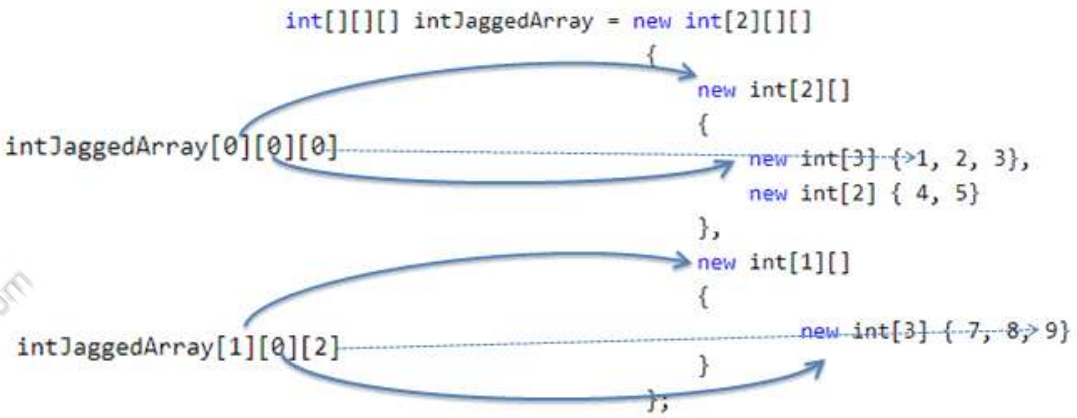
noteghar.com

Example: Jagged Array

```
int[][][] intJaggedArray = new int[2][][]
                           {
                               new int[2][]
                               {
                                   new int[3] { 1, 2, 3},
                                   new int[2] { 4, 5}
                               },
                               new int[1][]
                               {
                                   new int[3] { 7, 8, 9}
                               }
                           };

Console.WriteLine(intJaggedArray[0][0][0]); // 1

Console.WriteLine(intJaggedArray[0][1][1]); // 5

Console.WriteLine(intJaggedArray[1][0][2]); // 9
```

flyghar.com

In the above example of a jagged array, three brackets [][][] means an array of array of array. So, intJaggedArray will contain two elements, which means two arrays. Now, each of these arrays also contains an array (single-dimension). intJaggedArray[0][0][0] points to the first element of first inner array. intJaggedArray[1][0][2] points to the third element of the second inner array. The following figure illustrates this.



noteghar.com

Jagged Array

C# Array Class

In c#, we have a class called **Array** and it will act as a base class for all the arrays in common language runtime (CLR). The Array class provides methods for creating, manipulating, searching and sorting arrays.

For example, by using **Sort** or **Copy** methods of **Array** class we can sort the elements of an array and copy the elements of one array to another based on our requirements.

Following is the example of using an Array class to sort or filter or reverse array elements in c# programming language.

```
using System;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = new int[5] { 1, 4, 2, 3, 5 };
            Console.WriteLine("---Initial Array Elements---");
            foreach (int i in array)
            {
                Console.WriteLine(i);
            }
            Array.Sort(array);
            Console.WriteLine("---Elements After Sort---");
            foreach (int i in array)
            {
                Console.WriteLine(i);
            }
            Array.Reverse(array);
            Console.WriteLine("---Elements After Reverse---");
            foreach (int i in array)
            {
                Console.WriteLine(i);
            }
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we sorting and changing the order of array elements using **Sort** and **Reverse** methods of an **Array** class.

When we execute the above c# program, we will get the result as shown below.

flyghar.com

If you observe the above result, we sorted the elements of an array and changed the order of array elements using the Array class based on our requirements.

This is how we can use the required methods of **Array** class in our applications to implement the required functionality.

noteghar.com

C# Comments (Single Line, Multiline, XML)

In c#, **Comments** are the self-explanatory notes to provide detailed information about the code which we wrote in our applications.

It's always a good practice to include the comments in our c# code to provide detailed information about the functionality, like what a specific block or line of code can do. It will benefit anyone else who examines the code.

In c#, we can include the comments anywhere in the program without affecting our code. The comments in c# do not affect an application's performance because the comments won't be compiled and executed by the compiler.

In c#, there are three types of comments available, those are

- Single-line Comments
- Multi-line Comments
- XML Comments

C# Single Line Comments

In c#, **Single Line Comments** are defined by using **//** (**double forward slash**).

Following is the syntax of defining the single-line comments in the c# programming language.

```
// Single Line Comment
```

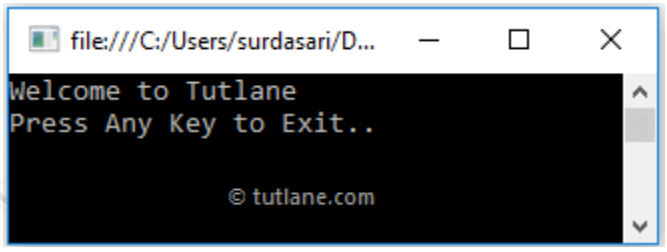
Now we will see how to define a single line comment in c# programs with examples.

Following is the example of defining the single-line comments in the c# programming language.

```
using System;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Calling Method to Show Greet Messaging
            GreetMessage();
            Console.WriteLine("Press Any Key to Exit..");
            Console.ReadLine(); // This method to read the commands fr
om console
        }
        // This Method will display the welcome message
        public static void GreetMessage()
        {
            Console.WriteLine("Welcome to Tutlane");
        }
    }
}
```

If you observe the above code, we defined a single line comment to describe the specific block or line of code in our c# application.

When we execute the above c# program, we will get the result as shown below.



This is how we can use single-line comments to provide detailed information about our c# application's functionality.

C# Multiline comments

In c#, Multiline Comments are used to comment the multiple lines of code, and the multiple lines of comments are surrounded by slash and asterisk like **/* …… */**.

Following is the syntax of defining the multiline comments in the c# programming language.

```
/* Multi Line Comment */
```

Now we will see how to define multiline comments in c# programs with examples.

Following is the example of defining the multiline comments in the c# programming language.

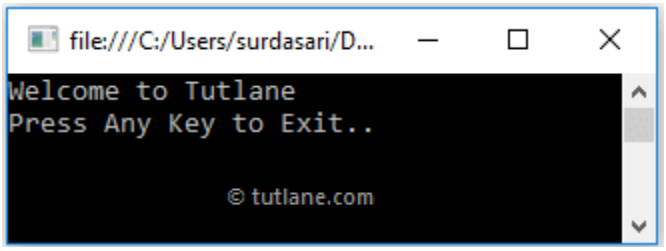
```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         /*
          Define & assign a variable and
          Send a variable to the method to display
         */
         string msg = "Welcome to Tutlane";
         GreetMessage(msg);
         Console.WriteLine("Press Enter Key to Exit..");
         Console.ReadLine();
      }
      public static void GreetMessage(string message)
      {
         Console.WriteLine(message);
      }
```

noteghar.com
tlyghar.com

```
    }
}
```

If you observe the above code, we defined multiline comments to describe the specific block or line of code in our c# application.

When we execute the above c# program, we will get the result as shown below.



This is how we can use multiline comments to provide detailed information about our c# application's functionality.

C# XML Comments

In c#, **XML Comments** are the special type of comments, and these will be added above the definition of any user-defined type or member.

In c#, the XML Comments are defined by using /// (triple forward slashes) and with XML formatted comment body.

Following is the syntax of defining the XML comments in the c# programming language.

```
///<summary>
/// This class does something.
///</summary>

public class SomeClass
{

}
```

Now, we will see how to define XML comments in c# programs with examples.

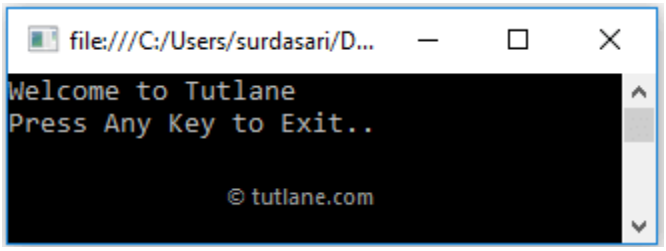
Following is the example of defining XML comments in the c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         string msg = "Welcome to Tutlane";
         GreetMessage(msg);
         Console.WriteLine("Press Enter Key to Exit..");
         Console.ReadLine();
      }
      ///<summary>
      /// Method to Display Welcome Message
      ///</summary>
      ///<param name="message"></param>
      public static void GreetMessage(string message)
      {
        Console.WriteLine(message);
      }
   }
}
```

If you observe the above code, we defined XML comments to describe the specific block or line of code in our c# application.

Here the **<summary>** tag is used to add detailed information about a type or member and **<param>** tag is used to describe the method parameters.

When we execute the above c# program, we will get the result as shown below.



This is how we can use XML comments to provide detailed information about our c# application's functionality.

noteghar.com
tlyghar.com

C# Core Programming Constructs

What is C#?

- C# is pronounced "C-Sharp".
- It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.
- C# has roots from the C family, and the language is close to other popular languages like C++ and Java.
- The first version was released in year 2002. The latest version, **C# 8**, was released in September 2019.
- It is strongly type programming language.

flyghar.com

Why Use C#?

- It is one of the most popular programming language in the world
- It is easy to learn and simple to use
- It has a huge community support
- C# is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- As C# is close to C, C++ and Java, it makes it easy for programmers to switch to C# or vice versa

Why C# is a widely used professional language?

- It is a modern, general-purpose programming language
- It is object oriented.
- It is component oriented.
- It is easy to learn.
- It is a structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- It is a part of .Net Framework.

noteghar.com

Strong Programming Features of C#

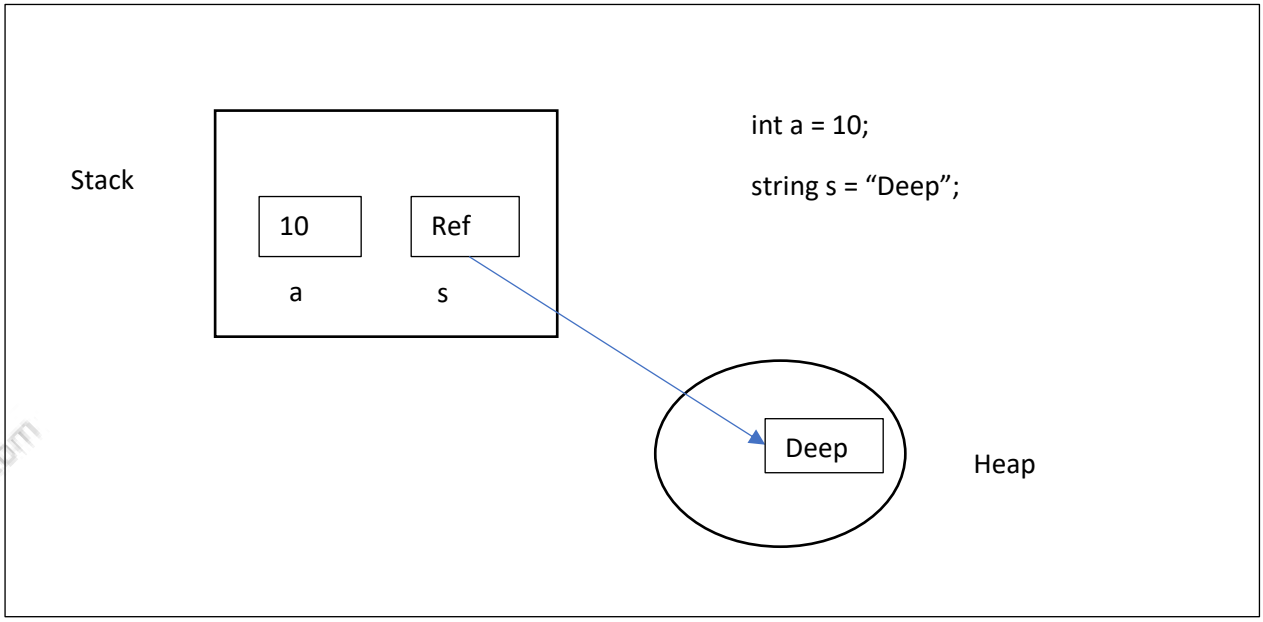
Although C# constructs closely follow traditional high-level languages, C and C++ and being an object-oriented programming language. It has strong resemblance with Java, it has numerous strong programming features that make it endearing to a number of programmers worldwide. The list of few important features of C# are given below:

- Boolean Conditions
- Automatic Garbage Collection
- Standard Library
- Assembly Versioning
- Properties and Events
- Delegates and Events Management
- Easy-to-use Generics
- Indexers
- Conditional Compilation
- Simple Multithreading
- LINQ and Lambda Expressions
- Integration with Windows

flyghar.com

Data Types

Like any programming language C# uses data types to represent a variable. The CLR provides the two categories of data types, **Value Type** and **Reference Type**. Both types are stored in different types of memory; a value type stores its value in the stack and a reference type stores a reference to the value in the managed heap. Now the question becomes, how to determine whether a type it is a value type or a reference type. For example int is a value type and a class object is a reference type.



noteghar.com

- Heap is a dynamic area which is created at the runtime it will be in action whenever it required.

- Stack is a major area where we allocate memory so, All fetching will be started from stack.
- Stack is always fixed memory area. Doesnot matter How many operation under going in a application but all the major memory allocation is done by stack.
- If you start storing value of reference type ie. String , Array , Object in a stack .they will take large space in stack and stack will be overloaded and program will be terminated.
- So, All Bulkiest type is stored in heap and stack will not get overloaded and it will maintain the performance of our program as well.

1.Value type

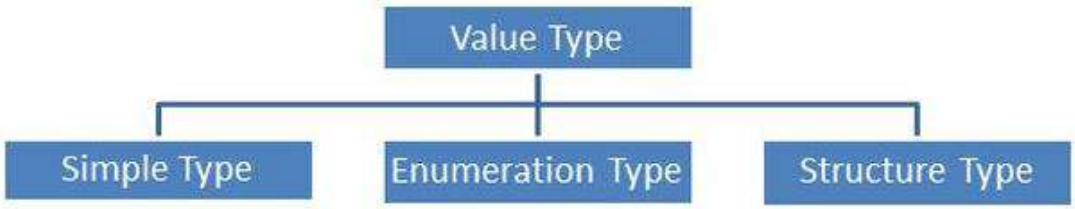


fig1.1 Value type Hierarchy

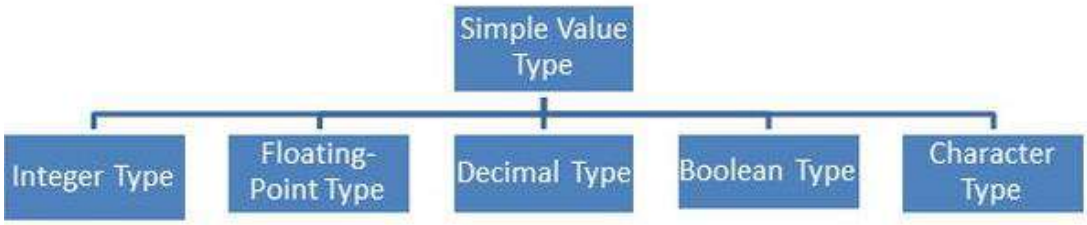
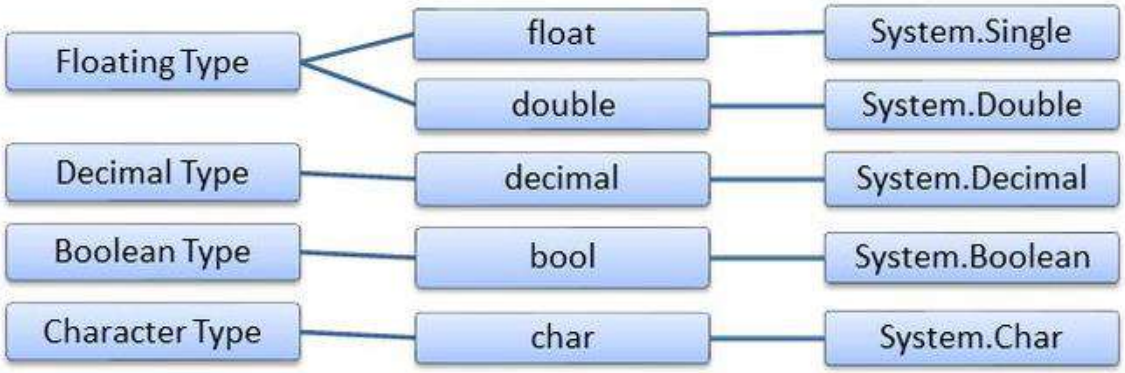


fig1.2 Value type Subcategory Hierarchy

noteghar.com
flyghar.com

fig1.3 Integer type Hierarchy



1.4 Other Value type Hierarchy

2.Reference Type

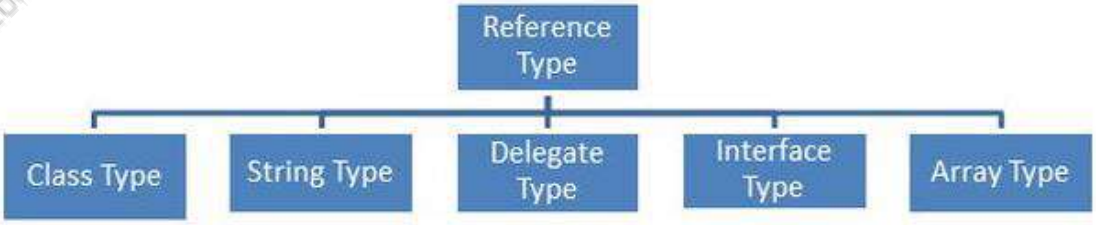


fig1.5 Reference type Hierarchy

Variables

Variables are special containers that hold some data at specific memory locations. Variables can either be numeric, string, date, character, Boolean and so on. A variable can be declared with the following syntax in C#:

Datatype identifier;

int a;

This statement declares an int type "a" variable. The CLR compiler won't allow you to use this variable until you initialize it with a value. You can initialize this variable using the equal ("=") sign as in the following:

int a=2;

You can declare and initialize multiple variables into a single statement but all of the variables will be of the same type as in the following:

```
int x , a=20 , y=10;
```

The C# compiler requires that a variable must be initialized first before use in an operation, otherwise it is flagged as a warning. If a variable is left uninitialized in a class or structure then the vigilant C# compiler sets them to zero(s).

In the case of a class object reference, they must also be initialized using a new keyword. We can assign a null value to the reference type as in the following:

```
1. Myclass obj; // not initialized
2. Obj=null;
3. Obj= new Myclass(); // initialized
```

All the intrinsic data types support a constructor that allows creation of a variable using the new keyword that automatically sets the variable to its default value as in the following:

- bool type set to false;
- Numeric set to 0;

- Object reference set to Null;
- Date set to 1/1/0001 12:00:00 AM;

Constants

A constant is a variable whose value can't be changed throughout its lifetime, unlike primitive variables. By prefixing a variable with the const keyword and later initializing it, it is designated as a constant variable.

```
const int i=10;
```

A constant variable has the following special characteristics:

- They must be initialized during declaration and then they remain intact throughout the entire program's life.
- Constant variables are always implicitly static.
- Constants make your program easier to modify.

Conditional Statement

C# conditional statements allow you to branch your code depending on a certain value of an expression or condition. C# has two constructs for branching code, the if statement and the switch statement.

1.If/else construct

The if statement allows you to test whether a specific condition is met or not. The syntax for declaring the if statement is as in the following:

```
1. if (condition)
2. Statement
3. else
4. statement
```

For example in the following program we want to determine whether or not a string is longer than zero characters:

```
1. static void Main(string[] args)
2. {
3.
```

```
4.    string str = "welcome home";
5.
6.    if (str.Length > 0)
7.       Console.WriteLine("lenght is > 0");
8.    else
9.       Console.WriteLine("lenght is < 0");
10.
11.         Console.ReadKey();
12.       }
```

flyghar.com

You can also use an if statement without the final else statement as in the following:

```
1. int data=10;
2. if (data !=5)
3.   {
4.    Console.WriteLine("value is not equal");
5.   }
```

It is not required to use curly braces on only one if statement.

If more than one statement is to be executed as part of either condition then these statements will need to be joined together into a block using curly braces ({…}) as in the following:

```
1. bool isFalse;
2. int data;
3.
4.  if (data == 0)
5.  {
6.    isFalse = true;
7.    Console.WriteLine("data is 0");
8.  }
9. else
10.     {
11.        isFalse = false;
12.        Console.WriteLine("data is not 0");
13.     }
```

noteghar.com

You can also combine an if/else statement to test multiple conditions. For example we are checking the highest number from the three numbers as in the following:

```
1. int x=10, y=20,z=22;
```

```
2. if (x >y)
3.    {
4.      Console.WriteLine("x is Highest");
5.    }
6. else if (x > z)
7.    {
8.      Console.WriteLine("x is Highest");
9.    }
10.      else if (y > z)
11.         {
12.           Console.WriteLine("y is Highest");
13.         }
14.      else
15.         {
16.           Console.WriteLine("z is Highest");
17.         }
```

Note: The expression in the if clause must evaluate to a Boolean (true / false)

2.Switch construct

The switch statement allows you to handle program flow based on predefined sets of conditions. It takes a switch argument followed by a series of case clauses. The syntax of the switch construct is as in the following:

```
1. switch(argument)
2. {
3.      case 1:
4.      // Do anything
5.      break;
6.      case 2:
7.      // Do anything
8.      break;
9.      default:
10.           // Do anything
11.           break;
12.     }
```

noteghar.com
flyghar.com

When the expression in the switch argument is evaluated, the code immediately following the case clause executes and you mark the end of statements by the break clause. If the expression evaluates to none of the other clause then you can include a default clause.

Note: The order of case doesn't matter; you can put the default case first.

flyghar.com

The following example performs some short math operations like addition, multiplication and so on. You can ask the user at run time in form of choices what operation he wants to do. Then we read two numeric values from the console and execute the operation as selected earlier.

```
1. Static void Main(string[] args)
2.         {
3.             int x,y;
4.             Console.WriteLine("Enter two Integers");
5.             x = int.Parse(Console.ReadLine());
6.             y = int.Parse(Console.ReadLine());
7.             Console.WriteLine("Operations\n------------------
   ----------------\n");
8.             Console.WriteLine("1= Addition\n2= Substraction \n
   3= Multification");
9.             Console.WriteLine("Operations\n------------------
   ----------------\n");
10.                Console.Write("Enter the operation code::");

11.                int op = int.Parse(Console.ReadLine());
12.
13.                switch (op)
14.                {
15.                    case 1:
16.                        Console.WriteLine("Add=" + (x + y));

17.                        break;
18.                    case 2:
19.                        Console.WriteLine("Subs=" + (x - y));

20.                        break;
21.                    case 3:
22.                        Console.WriteLine("Multiply=" + (x *
   y));
23.                        break;
24.                    default:
25.                        Console.WriteLine("wrong choice");
```

noteghar.com

```
                        break;
                }
                Console.ReadKey();
            }
```

Although all the case clauses are executed in a sequence, if you want to fire a specific case clause early then you can do this by the goto case clause as in the following:

```
switch (country)
    {
        case "india":
        goto case "Canada";
        case "USA":
        break;
        case "Canada":
        break;
    }
```

Loops

Loops essentially allow you to execute a block of code repeatedly until a certain condition is met. C# provides the following four constructs:

- for loop
- foreach loop
- while loop
- do/while loop

1.The for loop construct

The for loop is for iterating a block of code a fixed number of times where you can test a specific condition before you do another iteration. The for loop construct syntax is as in the following:

```
for (initializer; condition; iterator)
{
    Statements;
}
```

Here the initializer is evaluated first before the first loop is executed. The loop is executed until the condition expressions are false and the iterator is responsible for incrementing or decrementing the loop counter.

The following program typically writes the value 0 to 4 on the screen by using the for loop construct. We are declaring a local variable x and initializing it to 0 to use it as loop counter. Then we test whether or not it is less than 5 in the condition block and finally you increase the counter by one and walk through the process again as in the following:

```
1. int x;
2. for (x = 0; x < 5; x++)
3.   {
4.     Console.WriteLine(x);
5.   }
```

flyghar.com

Note we can declare the local counter variable in the for loop block also and x++ can be simply written as x=x+1. You can create a complex condition, endless loops and use goto, break and continue statements in the for loop. The following program will terminate when the counter variable 8 is reached as in the following:

```
1. for (int i = 0; i < 10; i++)
2.     {
3.        if (i == 8)
4.          {
5.             break;
6.          }
7.          Console.WriteLine(i);
8.      }
```

In some circumstance you want your loops to be executed indefinitely. C# allows you to write infinite loops using the following syntax:

```
1. for (; ;) {  }
2.
3. for (; ;)
4.  {
5.  Console.WriteLine("printing");
6. }
```

noteghar.com

2.The foreach loop construct

The foreach loop construct allows you to iterate through each item into a collection without the necessity of testing the upper limit condition. This

construct can also be applied on user-defined collections. In the following you can traverse an array of string:

```
1. string[] country = { "India", "USA", "Canada", "China" };
2. foreach (string x in country)
3.   {
4.     Console.WriteLine(x);
5.   }
```

Here, the foreach loop steps through the array element one at a time. With each element, it places the value of the element in the string x variable and then performs an iteration of the loop.

3.The while loop construct

The while loop is mostly used to repeat a statement or a block of statements for a number of times that is not known before the loop begins. The syntax of the while loop is as in the following:

```
1. while(condition)
2.     Statement;
```

The following program checks whether the value of int x is less than 10 using a while loop and displays them:

```
1. int x = 0;
2. while (x<10)
3.    {
4.       Console.WriteLine("value of x is"+ x);
5.       x++;
6.    }
```

4.The do/while loop constructs

The do/while is nearly similar to its previous construct while loop but has the slight difference that the condition is evaluated after the body of the loop has been executed. This loop is useful for the situation in which the statements

must execute at least once. The syntax for the do/while loop is as in the following:

```
do
{
  //body
} while(condition);
```

flyghar.com

The following program depicts the same objective as in a while loop but it executes the body statements first and later it checks the condition as in the following:

```
int x = 0;
do
  {
   Console.WriteLine("value of x is" + x);
   x++;
  } while (x < 10);
```

Jump Statement

C# provides special statements that allow you to jump immediately to another line in the program. **The break**, **continue** and **goto statements** are known as jump statements.

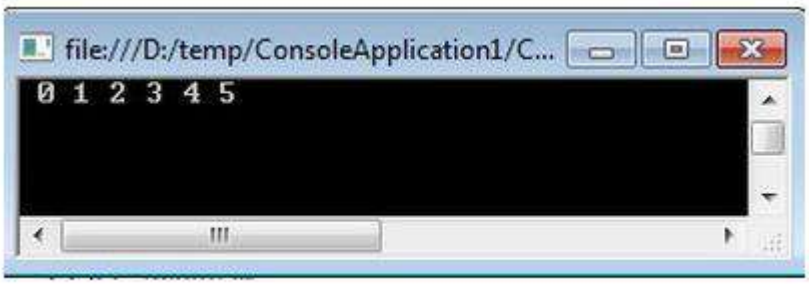
1.The break statement

The break statements are mainly used to exit the current iteration containing for, foreach, while and do..while loops. The following example shows the break statements to exit a for loop as in the following.

```
for (int i = 0; i < 10; i++)
    {
      Console.Write(" " + i);
      if (i == 5) break;
    }
```

noteghar.com

Here the for loops stops its execution when the value of i reaches 5 using the break statement. The output of this program is as in the following:



flyghar.com

fig: 1.6 Break statements Output

2.The continue statement

The continue statement is a jump statement and can be used in for, foreach, while and do..while loops. It stops the current execution and begins or restarts a new iteration, it does not exit from the current iteration as does the break statement.

```
1. for (int j = 0; j < 10; j++)
2.     {
3.        if (j == 5) continue;
4.        Console.Write(" "+j);
5.      }
```

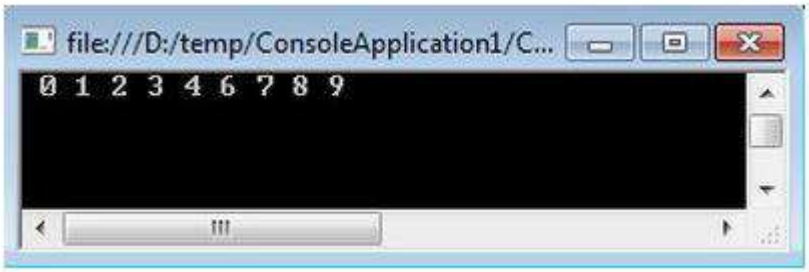


fig: 1.7 Continue Statement output

noteghar.com

3.The goto statement

The goto statements allow you to jump directly to another specific line in the program indicated by a label identifier. This statement is quite handy for jumping among cases in a switch statement. The following program declares a label identifier lableX to display the incremented value of x for each jump from the condition as in the following:

```
1. int i = 0;
2. lableX: Console.WriteLine("lable construct"+ i++);
3.
4. if (i < 5)
5. goto lableX;
```

flyghar.com

noteghar.com

C# Identifier

In C#, an **identifier** is a name assigned to a method, a variable, or any other user-defined item. Identifiers can be one or more characters long. Variable names may start with any letter of the alphabet or an underscore. Next may be a letter, a digit, or an underscore.

Remember, you can't start an **identifier** with a digit.

flyghar.com

Example:

```
public class GFG {
    static public void Main ()
    {
        int x;
    }
}
```

Here the total number of identifers present in the above example is 3 and the names of these identifiers are:

- **GFG:** Name of the class
- **Main:** Method name
- **x:** Variable name

Rules for defining identifiers in C#:

There are certain valid rules for defining a valid C# identifier. These rules should be followed, otherwise, we will get a compile-time error.

noteghar.com

- The only allowed characters for identifiers are all alphanumeric characters(**[A-Z]**, **[a-z]**, **[0-9]**), '**_**' (underscore). For example "geek@" is not a valid C# identifier as it contain '@' – special character.
- Identifiers should not start with digits([0-9]). For example "123geeks" is a not a valid in C# identifier.
- Identifiers should not contain white spaces.

- Identifiers are not allowed to use as **keyword** unless they include @ as a prefix. For example, **@as** is a valid identifier, but "**as**" is not because it is a keyword.
- C# identifers allow Unicode Characters.
- C# identifiers are case-sensitive.
- C# identifers cannot contain more than 512 characters.

flyghar.com

C# Keywords (Reserved, Contextual)

In c#, **Keywords** are the predefined set of reserved words that have special meaning for the compiler. So the keywords in c# cannot be used as identifiers such as variable name, class name, etc., in our applications.

Use Keywords as Variable Names

In c#, if you want to use Keywords as variable names (identifiers), you need to include @ as a prefix for your variable names. For example, **@switch** is a valid identifier, but the **switch** is not because it's a keyword and having a special meaning for the compiler.

Following is the example of using the reserved keywords as variable names by including @ as a prefix in c# programming language.

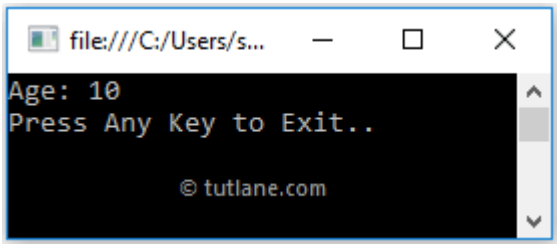
```
using System;

namespace CsharpExamples
{
   public class @class
   {
      public int age;
   }
   class Program
   {
      static void Main(string[] args)
```

noteghar.com

```
    {
        @class p1 = new @class();
        p1.age = 10;
        Console.WriteLine("Age: "+p1.age);
        Console.WriteLine("Press Enter Key to Exit..");
        Console.ReadLine();
    }
  }
}
```

If you observe the above c# example, we used a **class** keyword as a variable name (**@class**) by including `@` as a prefix.

When you execute the above program, you will get the result as shown below.



We can use keywords as variable names in the c# programming language based on our requirements.

Different Types of Keywords

In c#, Keywords are differentiated into two types those are

- Reserve Keywords
- Contextual Keywords

Reserved keywords

The following table lists the available reserved keywords in the c# programming language.

abstract	bool	continue	decimal	default
event	explicit	extern	char	checked
class	const	break	as	base
delegate	is	lock	long	num
byte	case	catch	false	finally
fixed	float	for	as	foreach
goto	if	implicit	in	int
interface	internal	do	double	else
namespace	new	null	object	operator
out	override	params	private	protected
public	readonly	sealed	short	sizeof
ref	return	sbyte	stackalloc	static
string	struct	void	volatile	while
true	try	switch	this	throw
unchecked	unsafe	ushort	using	using static

flyghar.com
noteghar.com

virtual	typeof	uint	ulong	out (generic modifier)

Contextual keywords

In c#, **Contextual** keywords can be used as an identifier in a limited program context, which can be outside of the context.

flyghar.com

The following table lists the available Contextual Keywords in the c# programming language.

add	alias	async	await	dynamic
from	get	orderby	ascending	descending
group	into	join	let	nameof
global	partial	set	remove	select
value	var	when	Where	yield

These are the keywords available in the c# programming language, and we can use them in our applications based on our

noteghar.com

C# Namespaces with Examples

In c#, **Namespace** is used to organize the multiple classes in our applications, reducing the code redundancy in our .NET applications. By using the namespace keyword, we can define the namespaces in our c# applications.

flyghar.com

Following is the syntax of defining a namespace in the c# programming language.

```
namespace Namespace_Name
{
// Body of Namespace
}
```

If you observe the above syntax, we defined a namespace in c# using the namespace keyword.

Following is the example of defining the namespace in the c# programming language.

```
namespace Tutlane
{
    class Welcome
    {
        public void GreetMessage()
        {
            // your method code
        }
    }
}
```

noteghar.com

If you observe the above code, we defined a namespace **Tutlane,** and it consists of a class **Welcome** as its member, and **GreetMessage()** is a method of **Welcome** class.

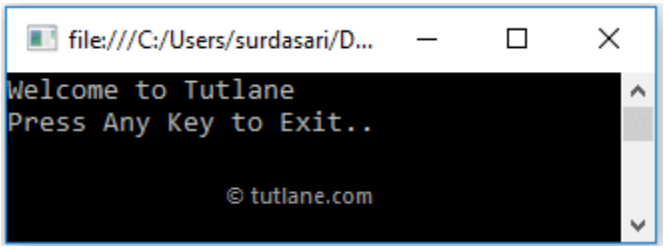
By default, **.NET Framework** provided a lot of namespaces to make the application implementation easy.

Following is the example of using the **.NET Framework** namespace "**System**" in the c# programming language.

```
namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         System.Console.WriteLine("Welcome to Tutlane");
         System.Console.WriteLine("Press Enter Key to Exit..");
         System.Console.ReadLine();
      }
   }
}
```

In the above example, **System** is a namespace, and **Console** is a class in that namespace. **WriteLine** and **ReadLine** are the methods of the **Console** class.

When we execute the above c# program, it will return the result as shown below.



noteghar.com
tlyghar.com

C# Access Namespace with using Keyword

If you observe the above example, every time we used a **System** namespace to access the **Console** class methods instead, you can import the **System** namespace in your application and use the associated classes and methods with using keyword.

In c#, using keyword is useful to import the namespaces and provide access to the associated classes and methods to use it in our .NET applications.

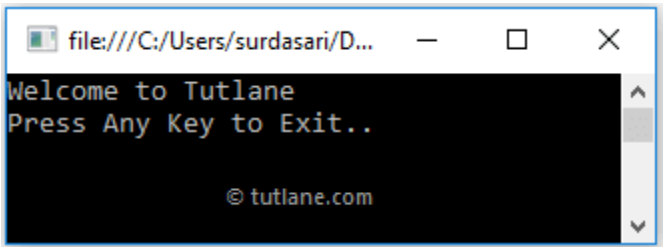
Following is the example of importing **System** namespace with using keyword in c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Welcome to Tutlane");
         Console.WriteLine("Press Any Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

If you observe the above example, we imported a "**System**" namespace with the "**using**" keyword to access Console class and associated methods (**WriteLine**, **ReadLine**) directly without using the **System** namespace every time.

To know more details about the above program, check this C# Hello World Program Example.

When we execute the above c# program, it will return the result as shown below.



Like the **System** namespace, the **.NET Framework** has various namespaces, those are

Namespace	Description
System	It contains classes that allow you to perform basic operations such as mathematical operations and data conversation.
System.IO	It contains classes to perform Input and Output operations.
System.Net	It contains classes that are useful to network protocols.
System.Data	It contains classes that are useful to work with ADO.Net architecture.
System.Collection	It contains classes that are useful to implement the collection of objects, such as lists.
System.Drawing	It contains classes that are useful to implement GUI functionalities.
System.Web	It contains classes that are helpful to perform HTTP requests.

In simple words, we can say that **Namespace** is a collection of classes, and classes are the collection of objects and methods. Using namespaces, we can easily access all the class methods just by importing the namespace in our application.

C# Define a Multiple Namespaces

In c#, we can define and access multiple namespaces in our application with using keyword.

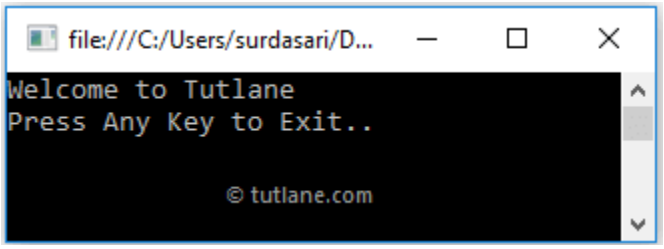
To access the custom namespace classes, we need to import the custom namespace with using keyword and need to create an instance for that classes in our application.

Following is the example of defining and accessing the multiple namespaces in the c# programming language.

```
using System;
using CustomNameSpace;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         Welcome w = new Welcome();
         w.GreetMessage();
         Console.WriteLine("Press Any Key to Exit..");
         Console.ReadLine();
      }
   }
}

namespace CustomNameSpace {
   class Welcome {
      public void GreetMessage() {
          Console.WriteLine("Welcome to Tutlane");
      }
   }
}
```

If you observe the above example, we created multiple namespaces and accessed the
namespace **CustomNamespace** classes in **Tutlane** namespace by creating the class instance.

When we execute the above c# program, it will return the result as shown below.



This is how you can create and access multiple namespaces in your c# applications.

Nested Namespaces in C#

In c#, a namespace can contain another namespace, and it is called a **Nested Namespace**. The nested namespace classes and their members can be accessed by using the **dot** (**.**) operator.

Following is the syntax of defining the nested namespace in the c# programming language.

```
namespace namespace1 {
   namespace nestednamespace {
   // Nested Namespace Code
   }
}
```

If you observe the above syntax, we defined a namespace "**nestednamespace**" within another namespace "**namespace1**".

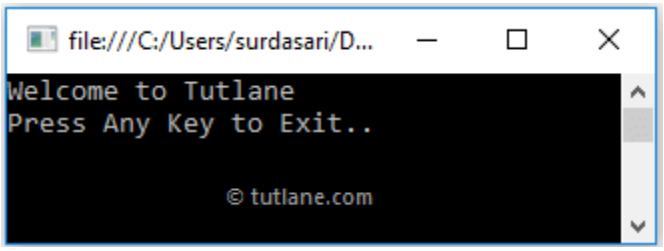
Following is the example of using nested namespace in c# programming language.

```
using System;
using CustomNameSpace.Nested;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
```

```
        {
            Welcome w = new Welcome();
            w.GreetMessage();
            Console.WriteLine("Press Any Key to Exit..");
            Console.ReadLine();
        }
    }
}
namespace CustomNameSpace {
    namespace Nested
    {
        class Welcome
        {
            public void GreetMessage()
            {
                Console.WriteLine("Welcome to Tutlane");
            }
        }
    }
}
```

If you observe the above code, we imported nested namespace with using keyword like using CustomNameSpace.Nested; and accessing the nested namespace classes by creating the instance.

When we execute the above c# program, it will return the result as shown below.



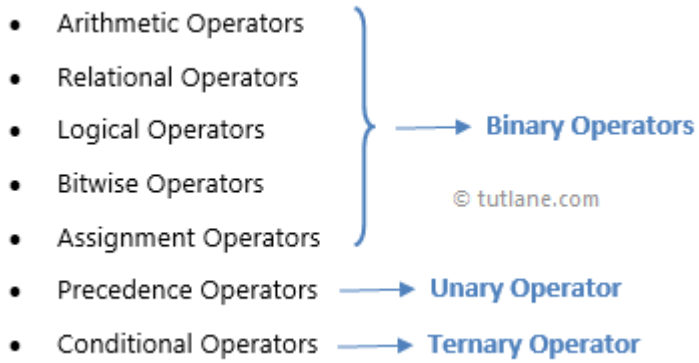
This is how we can use the nested namespaces in our c# applications based on our requirements.

C# Operators (Arithmetic, Relational, Logical, Assignment, Precedence)

In c#, the **Operator** is a programming element that specifies what kind of operation needs to perform on operands or variables. For example, an addition (+) operator in c# is used to perform the sum operation on operands.

flyghar.com

In c#, we have a different type of operators available, those are



These operators are categorized as a different group such as **Binary Operators**, **Unary Operators** and **Ternary Operators** based on taking the number operands to perform the operations.

Suppose, if the operator takes one operand to perform the operation such as precedence operator (++), then those will call as **Unary Operators**. In case, if the operator takes two operands to perform the operation such as arithmetic operator (-, +, *, /), then those will call as a **Binary Operators** and we have a Ternary Operator that takes three operands to perform the operation, such as conditional operator (**?:**).

Now we will learn each operator in a detailed manner with examples in c# programming language.

noteghar.com

C# Arithmetic Operators

In c#, **Arithmetic Operators** are useful to perform basic arithmetic calculations like addition, subtraction, division, etc. based on our requirements.

flyghar.com

C# Relational Operators

In c#, **Relational Operators** are useful to check the relation between two operands like we can determine whether two operand values equal or not, etc. based on our requirements.

C# Logical Operators

In c#, **Logical Operators** are useful to perform the logical operation between two operands like AND, OR and NOT based on our requirements and the operands must contain only Boolean values otherwise Logical Operators will throw an error.

C# Bitwise Operators

In c#, **Bitwise Operators** will work on bits and these are useful to perform the bit by bit operations such as Bitwise AND (&), Bitwise OR (|), Bitwise Exclusive OR (^), etc. on operands and we can perform bit-level operations on Boolean and integer data.

C# Assignment Operators

noteghar.com

In c#, **Assignment Operators** are useful to assign a new value to the operand and these operators will work with only one operand.

C# Operators Precedence

In c#, **Operators Precedence** is useful to define multiple operators in single expression and the evaluation of expression can be happened based on the priority of operators.

tiyeghar.com

noteghar.com

C# Arithmetic Operators with Examples

In c#, **Arithmetic Operators** are useful to perform basic arithmetic calculations like addition, subtraction, division, etc., based on our requirements.

For example, we have integer variables **x = 20**, **y = 10,** and if we apply an arithmetic operator **+** (**x + y**) to perform an addition operator. We will get the result as 30 like as shown below.

```
int result;
int x = 20, y = 10;
result = (x + y);
```

The following table lists the different types of operators available in c# arithmetic operators.

Operator	Name	Description	Example (a = 6, b = 3)
+	Addition	It adds two operands.	a + b = 9
-	Subtraction	It subtracts two operands.	a - b = 3
*	Multiplication	It multiplies two operands.	a * b = 18
/	Division	It divides numerator by de-numerator.	a / b = 2
%	Modulo	It returns a remainder as a result.	a % b = 0

noteghar.com

C# Arithmetic Operators Example

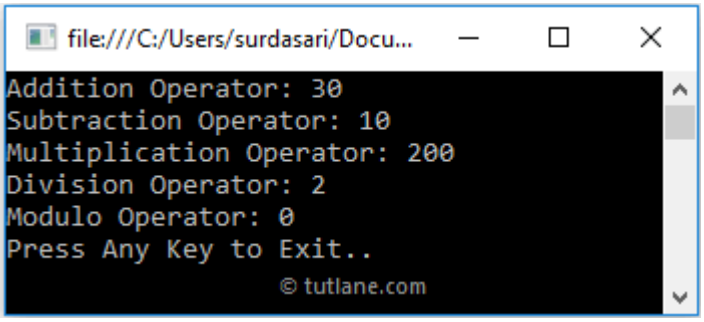
Following is the example of using the Arithmetic Operators in the c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         int result;
         int x = 20, y = 10;
         result = (x + y);
         Console.WriteLine("Addition Operator: " + result);
         result = (x - y);
         Console.WriteLine("Subtraction Operator: " + result);
         result = (x * y);
         Console.WriteLine("Multiplication Operator: "+ result);
         result = (x / y);
         Console.WriteLine("Division Operator: " + result);
         result = (x % y);
         Console.WriteLine("Modulo Operator: " + result);
         Console.WriteLine("Press Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

If you observe the above code, we used Arithmetic operators (+, -, *, /, %) to perform different operations on defined operands based on our requirements.

Output of C# Arithmetic Operators Example

When we execute the above c# program, we will get the result as shown below.



This is how we can use arithmetic operators in the c# programming language to perform the basic arithmetic calculations on operands based on our requirements.

C# Relational Operators with Examples

In c#, **Relational Operators** are useful to check the relation between two operands like we can determine whether two operand values equal or not, etc., based on our requirements.

Generally, the c# relational operators will return **true** only when the defined operands relationship becomes **true**. Otherwise, it will return **false**.

For example, we have integer variables **a = 10**, **b = 20**. If we apply a relational operator >= (**a >= b**), we will get the result **false** because the variable "**a**" contains a value that is less than variable **b**.

The following table lists the different types of operators available in c# relational operators.

Operator	Name	Description	Example (a = 6, b = 3)
==	Equal to	It compares two operands, and it returns true if both are the same.	a == b (false)
>	Greater than	It compares whether the left operand greater than the right operand or not and returns true if it is satisfied.	a > b (true)
<	Less than	It compares whether the left operand less than the right operand or not and returns true if it is satisfied.	a < b (false)
>=	Greater than or Equal to	It compares whether the left operand greater than or equal to the right operand or not and returns true if it is satisfied.	a >= b (true)
<=	Less than or Equal to	It compares whether the left operand less than or equal to the right operand or not and returns true if it is satisfied.	a <= b (false)
!=	Not Equal to	It checks whether two operand values equal or not and return true if values are not equal.	a != b (true)

noteghar.com
flyghar.com

C# Relational Operators Example

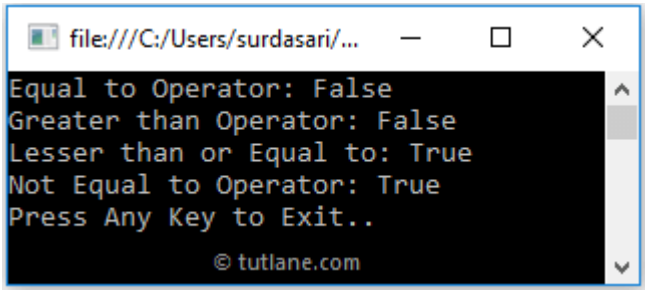
Following is the example of using the Relational Operators in c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         bool result;
         int x = 10, y = 20;
         result = (x == y);
         Console.WriteLine("Equal to Operator: " + result);
         result = (x > y);
         Console.WriteLine("Greater than Operator: " + result);
         result = (x <= y);
         Console.WriteLine("Lesser than or Equal to: "+ result);
         result = (x != y);
         Console.WriteLine("Not Equal to Operator: " + result);
         Console.WriteLine("Press Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

If you observe the above code, we used different Relational operators (<, >, ==) to perform required operations on defined operands.

Output of C# Relational Operators Example

When we execute the above c# program, we will get the result as shown below.



flyghar.com

This is how we can use the relational operators in the c# programming language to check the relationship between defined operands based on our requirements.

C# Logical Operators with Examples

In c#, **Logical Operators** are useful to perform the logical operation between two operands like AND, OR, and NOT based on our requirements. The Logical Operators will always work with Boolean expressions (**true** or **false**) and return Boolean values.

The operands in logical operators must always contain only Boolean values. Otherwise, Logical Operators will throw an error.

The following table lists the different types of operators available in c# logical operators.

Operator	Name	Description	Example (a = true, b = false)
&&	Logical AND	It returns true if both operands are non-zero.	a && b (false)

noteghar.com

Operator	Name	Description	Example (a = true, b = false)
||	Logical OR	It returns true if any one operand becomes a non-zero.	a || b (true)
!	Logical NOT	It will return the reverse of a logical state that means if both operands are non-zero, it will return false.	!(a && b) (true)

If we use Logical **AND**, **OR** operators in c# applications, those will return the result as shown below for different inputs.

Operand1	Operand2	AND	OR
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

If you observe the above table, if any one operand value becomes **false**, then the logical **AND** operator will return **false.** **The** logical **OR** operator will return **true** if any one operand value becomes **true**.

If we use the Logical **NOT** operator in our c# applications, it will return the results like as shown below for different inputs.

Operand	NOT
true	false
false	true

If you observe the above table, the Logical **NOT** operator will always return the reverse value of the operand. If the operand value is **true**, then the Logical **NOT** operator will return **false** and vice versa.

C# Logical Operators Example

Following is the example of using the Logical Operators in c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         int x = 15, y = 10;
         bool a = true, result;
         // AND operator
         result = (x <= y) && (x > 10);
         Console.WriteLine("AND Operator: " + result);
         // OR operator
         result = (x >= y) || (x < 5);
         Console.WriteLine("OR Operator: " + result);
         //NOT operator
         result = !a;
         Console.WriteLine("NOT Operator: " + result);
```

noteghar.com

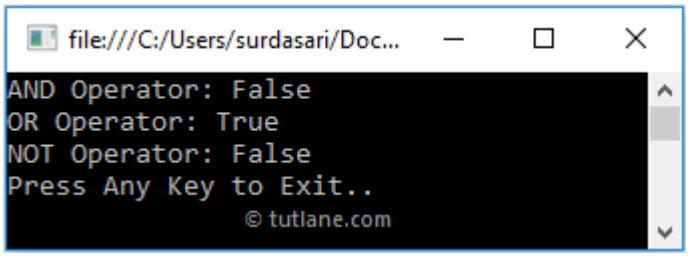
```
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above code, we used logical operators (**AND**, **OR**, **NOT**) to perform different operations on defined operands.

flyghar.com

Output of C# Logical Operators Example

When we execute the above c# program, we will get the result as shown below.



This is how we can use logical operators in the c# programming language to perform logical operations on defined operands based on our requirements.

C# Bitwise Operators with Examples

In c#, **Bitwise Operators** will work on bits, and these are useful to perform bit by bit operations such as Bitwise AND (&), Bitwise OR (|), Bitwise Exclusive OR (^), etc. on operands. We can perform bit-level operations on Boolean and integer data.

noteghar.com

For example, we have integer variables **a = 10**, **b = 20,** and the binary format of these variables will be shown below.

```
a = 10 (00001010)
b = 20 (00010100)
```

When we apply the **Bitwise OR** (|) operator on these parameters, we will get the result shown below.

```
00001010
00010100
----------
00011110 = 30 (Decimal)
```

flyghar.com

The following table lists the different types of operators available in c# bitwise operators.

Operator	Name	Description	Example (a = 0, b = 1)
&	Bitwise AND	It compares each bit of the first operand with the corresponding bit of its second operand. If both bits are 1, then the result bit will be 1; otherwise, the result will be 0.	a & b (0)
|	Bitwise OR	It compares each bit of the first operand with the corresponding bit of its second operand. If either of the bit is 1, then the result bit will be 1; otherwise, the result will be 0.	a | b (1)
^	Bitwise Exclusive OR (XOR)	It compares each bit of the first operand with the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, then	a ^ b (1)

noteghar.com

Operator	Name	Description	Example (a = 0, b = 1)
		the result bit will be 1; otherwise, the result will be 0.	
~	Bitwise Complement	It operates on only one operand, and it will invert each bit of operand. It will change bit 1 to 0 and vice versa.	~(a) (1)
<<	Bitwise Left Shift)	It shifts the number to the left based on the specified number of bits. The zeroes will be added to the least significant bits.	b << 2 (100)
>>	Bitwise Right Shift	It shifts the number to the right based on the specified number of bits. The zeroes will be added to the least significant bits.	b >> 2 (001)

C# Bitwise Operators Example

Following is the example of using the Bitwise Operators in the c# programming language.

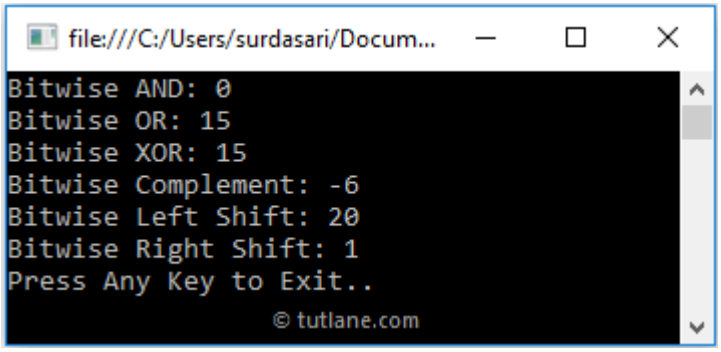
```
using System;

namespace Tutlane
{
   class Program
```

```
    {
        static void Main(string[] args)
        {
            int x = 5, y = 10, result;
            result = x & y;
            Console.WriteLine("Bitwise AND: " + result);
            result = x | y;
            Console.WriteLine("Bitwise OR: " + result);
            result = x ^ y;
            Console.WriteLine("Bitwise XOR: " + result);
            result = ~x;
            Console.WriteLine("Bitwise Complement: " + result);
            result = x << 2;
            Console.WriteLine("Bitwise Left Shift: " + result);
            result = x >> 2;
            Console.WriteLine("Bitwise Right Shift: " + result);
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above code, we used different bitwise operators (&, |, ^, ~, <<, >>) to perform different operations on defined operands.

Output of C# Bitwise Operators Example

When we execute the above c# program, we will get the result as shown below.



This is how we can use bitwise operators in the c# programming language to perform bit by bit operations on defined operands based on our requirements.

flyghar.com

C# Assignment Operators with Examples

In c#, **Assignment Operators** are useful to assign a new value to the operand, and these operators will work with only one operand.

For example, we can declare and assign a value to the variable using the assignment operator (=) like as shown below.

```
int a;
a = 10;
```

If you observe the above sample, we defined a variable called "**a**" and assigned a new value using an assignment operator (=) based on our requirements.

The following table lists the different types of operators available in c# assignment operators.

Operator	Name	Description	Example
=	Equal to	It is used to assign the values to variables.	int a; a = 10
+=	Addition Assignment	It performs the addition of left and right operands and assigns a result to the left operand.	a += 10 is equals to a = a + 10

noteghar.com

Operator	Name	Description	Example
-=	Subtraction Assignment	It performs the subtraction of left and right operands and assigns a result to the left operand.	a -= 10 is equals to a = a - 10
*=	Multiplication Assignment	It performs the multiplication of left and right operands and assigns a result to the left operand.	a *= 10 is equals to a = a * 10
/=	Division Assignment	It performs the division of left and right operands and assigns a result to the left operand.	a /= 10 is equals to a = a / 10
%=	Modulo Assignment	It performs the modulo operation on two operands and assigns a result to the left operand.	a %= 10 is equals to a = a % 10
&=	Bitwise AND Assignment	It performs the Bitwise AND operation on two operands and assigns a result to the left operand.	a &= 10 is equals to a = a & 10
|=	Bitwise OR Assignment	It performs the Bitwise OR operation on two operands and assigns a result to the left operand.	a |= 10 is equals to a = a | 10

noteghar.com
flyghar.com

Operator	Name	Description	Example
^=	Bitwise Exclusive OR Assignment	It performs the Bitwise XOR operation on two operands and assigns a result to the left operand.	a ^= 10 is equals to a = a ^ 10
>>=	Right Shift Assignment	It moves the left operand bit values to the right based on the number of positions specified by the second operand.	a >>= 2 is equals to a = a >> 2
<<=	Left Shift Assignment	It moves the left operand bit values to the left based on the number of positions specified by the second operand.	a <<= 2 is equals to a = a << 2

C# Assignment Operators Example

Following is the example of using assignment Operators in the c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         int x = 20;
```

```
            x += 10;
            Console.WriteLine("Add Assignment: " + x);
            x *= 4;
            Console.WriteLine("Multiply Assignment: " + x);
            x %= 7;
            Console.WriteLine("Modulo Assignment: " + x);
            x &= 10;
            Console.WriteLine("Bitwise And Assignment: " + x);
            x ^= 12;
            Console.WriteLine("Bitwise XOR Assignment: " + x);
            x >>= 3;
            Console.WriteLine("Right Shift Assignment: " + x);
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

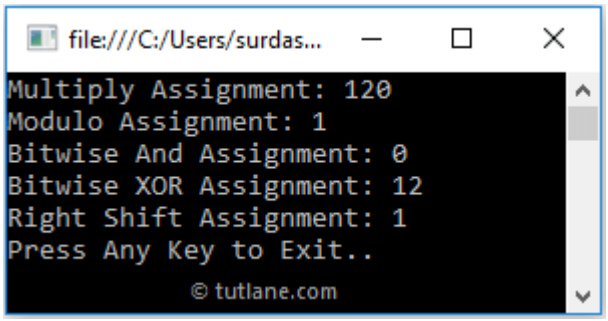
tiyeghar.com

If you observe the above example, we defined a variable or operand "**x**" and assigning new values to that variable by using assignment operators in the c# programming language.

Output of C# Assignment Operators Example

When we execute the above c# program, we will get the result as shown below.

noteghar.com



This is how we can use assignment operators in c# to assign new values to the variable based on our requirements.

C# Ternary Operator (?:) with Examples

In c#, **Ternary Operator (?:)** is a decision-making operator, and it is a substitute of if…else statement in c# programming language.

Using Ternary Operator, we can replace multiple lines of if…else statement code into a single line in c# programming language.

The Ternary operator will help you execute the statements based on the defined conditions using the decision-making operator (**?:**).

Syntax of C# Ternary Operator

In c#, the Ternary Operator will always work with **3** operands. Following is the syntax of defining a Ternary Operator in c# programming language.

```
condition_expression? first_expression : second_expression;
```

If you observe the above Ternary Operator syntax, the conditional operator (?:) will return only one value from the defined expressions, either **first_expression** or **second_expression** based on the value of a condition.

In c#, the Ternary Operator (?:) will work as follow.

- In Ternary Operator, the **condition expression** must be evaluated to either **true** or **false**. If the **condition** is **true**, the **first_expression** result is returned by the ternary operator.
- In case if the **condition** is **false**, then the **second_expression** result is returned by the operator.

As said earlier, the Ternary Operator (?:) is a substitute of if…else statement in c# programming language. For example, we can replace the following if…else statement with Ternary Operator (?:) like as shown following.

```
int x = 5, y = 20;
string result;

// if...else statement if (x > y)
{
result = "x greater than y";
}
else {
result = "x less than y";
}

//Ternary Operator (?:) statement
result = (x > y) ? "x greater than y" : "x less than y";
```

If you observe the above example, we simplified the if…else condition by replacing multiple lines of if…else condition code with Ternary Operator (?:) in c# programming language.

Now, we will see the complete example of a Ternary operator (?:) in c# programming language.

C# Ternary Operator Example

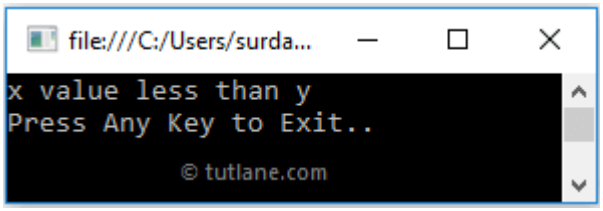
Following is the example of using a Ternary Operator (?:) in c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
```

```
        static void Main(string[] args)
        {
            int x = 5, y = 20;
            string result;
            //Ternary Operator (?:)
            result = (x > y) ? "x value greater than y" : "x value less than
y";
            Console.WriteLine(result);
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above code, we used a Ternary Operator (?:) to evaluate an expression (**x > y**) to show the result based on our requirements.

When you execute the above c# program, you will get the result as shown below.



This is how we can use Ternary Operator (?:) as a substitute for if…else statement in c# programming language.

C# Nested Ternary Operator

In c#, we can create a **Nested Ternary Operator** by including multiple conditional expressions as a second or third part of expressions in the ternary operator. These nested ternary operators will help us replace if…else if statements in c# programming language.

Following is the example of replacing if…else if statement with a nested ternary operator in c# programming language.

```
int x = 20, y = 20;
// If...else If Statement
string result;
if (x > y)
{
result = "x value greater than y";
}
else if (x < y)
{
result = "x value less than y";
}
else {
result = "x value equals to y";
}
//Nested Ternary Operator (?:)
result = (x > y) ? "x value greater than y" : (x < y) ? "x value less than y" : "x value equals to y";
```

If you observe the above code, we are able to replace multiple lines of if…else if code with a single line of the nested ternary operator based on our requirements.

In c#, the conditional operator is a right associative so the expression **a ? b : c ? d : e;** evaluated as **a ? b : (c ? d : e)**, not as **(a ? b : c) ? d : e**.

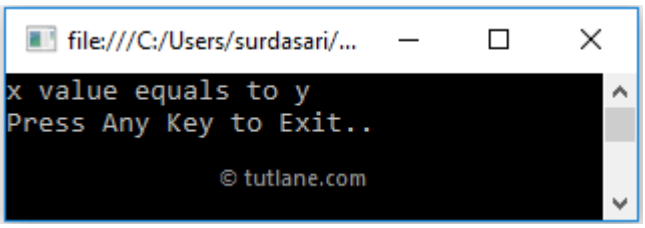
C# Nested Ternary Operator Example

Following is the example of defining a nested ternary operator in the c# programming language.

```
using System;

namespace Tutlane
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 20, y = 20;
            string result;
            //Nested Ternary Operator (?:)
            result = (x > y) ? "x value greater than y" : (x < y) ? "x value
less than y" : "x value equals to y";
            Console.WriteLine(result);
            Console.WriteLine("Press Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

When we execute the above c# program, we will get the result as shown below.



This is how we can implement a nested ternary operator in c# programming language to replace if…else if statements based on our requirements.

C# Operators Precedence with Examples

In c#, **Operators Precedence** is useful to define multiple operators in a single expression, and the evaluation of expression can be happened based on the priority of operators.

For example, the multiplication operator (*****) is having higher precedence than the addition operator (+). So if we use both multiplication (*****) and addition (+) operators in a single expression, first, it will evaluate the multiplication part and then the addition part in expression.

Following is the simple example of defining an expression with operator precedence in c#.

```
int i = 3 + 4 * 5;
```

When we execute the above statement, the multiplication part (**4 * 5**) will be evaluated first. After that, the addition part (**3 + 12**) will be executed, and the **i** value will become a **23**.

As said earlier, the multiplication (*****) operator is having higher precedence than the addition (+) operator so that the first multiplication part will be executed.

The following table lists the different types of operators available in c# relational operators.

Category	**Operator(s)**
Postfix / Prefix	++, --
Unary	+, -, !, ~
Multiplicative	*, /, %
Additive	+, -
Shift	<<, >>

Category	Operator(s)
Relational	<, <=, >, >=
Equality	==, !=
Bitwise	&, |, ^
Logical	&&, ||
Conditional	?:
Assignment	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

flyghar.com

C# Operator Precedence Example

Following is the example of implementing operator precedence in the c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         int x = 20, y = 5, z = 4;
         int result = x / y + z;
         Console.WriteLine("Result1: "+result);
         bool result2 = z <= y + x;
         Console.WriteLine("Result2: "+result2);
```

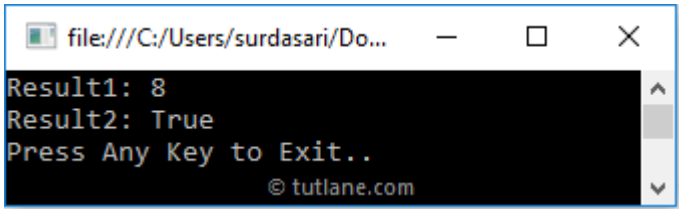
noteghar.com

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

If you observe the above example, we implemented operator precedence with multiple operators. The evaluation of expressions will be done based on operators' priority in the c# programming language.

Output of C# Operators Precedence Example

When we execute the above c# program, we will get the result as shown below.



This is how we can implement operator precedence in the c# programming language based on our requirements.

flyghar.com
noteghar.com

C# String with Examples

In c#, the **string** is a keyword that is useful to represent a sequential collection of characters called a text, and the string is an object of the **System.String** type.

In c#, we use **string** keyword to create string variables to hold the particular text, which is a sequential collection of characters.

The string variables in c# can hold any kind of text, and by using the **Length** property, we can know that the number of characters the string variable is holding based on our requirements.

C# String Declaration and Initialization

The following are the different ways of declaring and initializing string variables using **string** keyword in the c# programming language.

```
// Declare without initializing.
string str1;
// Declaring and Initializing
string str2 = "Welcome to Tutlane";
String str3 = "Hello World!";
// Initialize an empty string.
string str4 = String.Empty;
// Initialize to null.
String str5 = null;
// Creating a string from char
char[] letters = { 'A', 'B', 'C' };
string str6 = new string(letters);
```

If you observe the above code snippet, we created string variables using **string** and **String** keywords with or without initializing values based on our requirements.

C# string vs. String

If you observe the above declarations, we used two keywords called **string** and **String** to declare string variables. In c# , the **string** keyword is just an alias f or **String,** so both **string** and **String** are equivalent, and you can use whichever naming convention you prefer to define string variables.

tiyeghar.com

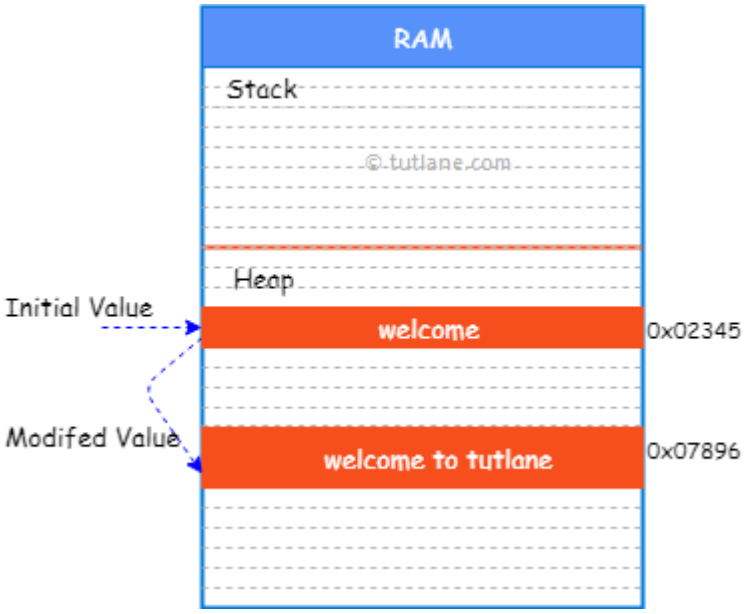
C# String Immutability

In c#, the **string** is **immutable**, which means the string object cannot be modified once it is created. If any changes are made to the string object, like adding or modifying an existing value, it will simply discard the old instance in memory and create a new instance to hold the new value.

For example, when we create a new string variable "**msg**" with the text "**welcome**", a new instance will create a heap memory to hold this value. Now, if we make any changes to the **msg** variable, like changing the text from "**welcome**" to "**welcome to tutlane**", then the old instance on heap memory will be discarded, and another instance will create on heap memory to hold the variable value instead of modifying the old instance in the memory.

Following is the pictorial representation of memory allocation for the string in the c# programming language.

noteghar.com



In c#, if we perform modifications like inserting, concatenating, removing, or replacing a value of the existing string multiple times, every time the new instance will create on heap memory to hold the new value, so automatically the performance of the application will be affected.

C# String Literals (Regular, Verbatim)

In c#, **string literal** is a sequence of characters enclosed in double quotation marks (" "). We have two kinds of string literals available in c#; those are **regular** and **verbatim**. The **regular** literals are useful when we want to embed escape characters like \n, \t, \', \", etc. in c#.

Following is the example of using regular string literals to embed escape characters in the c# programming language.

```
string names = "Suresh\nRohini\nTrishika";
Console.WriteLine(names);
/*
Output:
Suresh
Rohini
Trishika
```

```
*/
string msg = "Welcome to \"tutlane\" world";
Console.WriteLine(msg);
// Output: Welcome to "tutlane" world
```

In c#, the special character @ will serve as **verbatim** literal, and it is useful to represent a multiline string or a string with backslash characters, for example, to represent file paths.

Following is an example of using verbatim literal @ in c# programming language to represent a multiline string and a file path.

```
string path = @"C:\Users\Tutlane\Documents\";
Console.WriteLine(path);
//Output: C:\Users\Tutlane\Documents\

string msg = @"Hi Guest,
Welcome to Tutlane World
Learning Made Easy";
Console.WriteLine(msg);
/* Output:
Hi Guest,
Welcome to Tutlane World
Learning Made Easy
*/

string msg1 = @"My daughter name was ""Trishika.""";
Console.WriteLine(msg1);
//Output: My daughter name was "Trishika."
```

If you observe the above examples, we used a verbatim character @ to represent multiline strings or backslash character strings, etc., based on our requirements.

C# Format Strings

In c#, the format string is a string whose contents can be determined dynamically at runtime. We can create a format string using

the **Format** method and embedding placeholders in braces that will be replaced by other values at runtime.

Following is the example of using a format string to determine the string content dynamically at runtime in the c# programming language.

```
string name = "Suresh Dasari";
string location = "Hyderabad";
string user = string.Format("Name: {0}, Location: {1}", name, locatio
n);
Console.WriteLine(user);
// Output: Name: Suresh Dasari, Location: Hyderabad
```

If you observe the above code, we are formatting a string using the **Format** method and replacing the placeholders in braces with required values.

C# Access Individual Characters from Strings

In c#, we can access individual characters from string by using array notation with index values.

Following is the example of accessing individual characters from the string by specifying the index position.

```
string name = "Suresh Dasari";
for (int i = 0; i < name.Length; i++)
{
Console.Write(name[i]);
}
// Output: Suresh Dasari
```

If you observe the above example, we are looping through characters in a string using for loop and displaying each character of string based on the index position.

C# String Example

Following is the example declaring and initializing strings, formatting string value, and use string literals to represent data in the c# programming language.

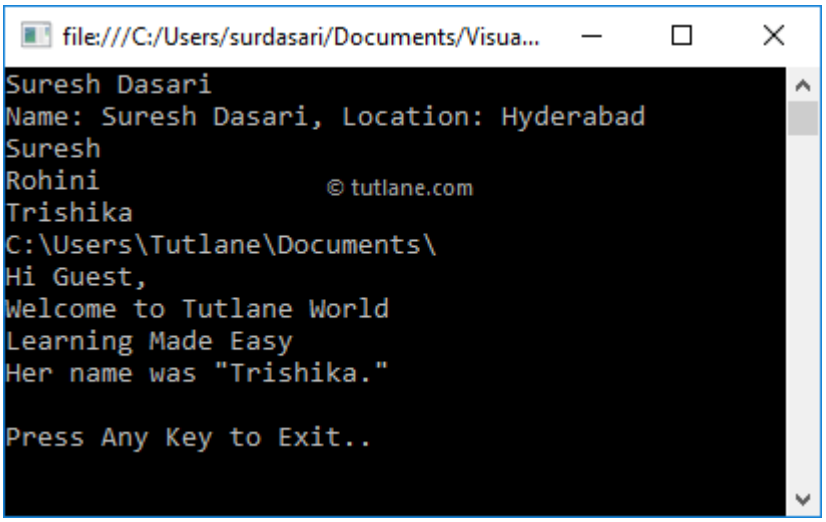
flyghar.com

```
using System;

namespace Tutlane
{
    class Program
    {
      static void Main(string[] args)
      {
          string firstname = "Suresh";
          string lastname = "Dasari";
          string location = "Hyderabad";
          string name = firstname + " " + lastname;
          Console.WriteLine(name);
          string userInfo = string.Format("Name: {0}, Location: {1}", n
ame, location);
          Console.WriteLine(userInfo);
          string names = "Suresh\nRohini\nTrishika";
          Console.WriteLine(names);
          string path = @"C:\Users\Tutlane\Documents\";
          Console.WriteLine(path);
          string msg = @"Hi Guest,
          Welcome to Tutlane World
          Learning Made Easy";
          Console.WriteLine(msg);
          string msg1 = @"Her name was ""Trishika.""";
          Console.WriteLine(msg1);
          Console.WriteLine("\nPress Enter Key to Exit..");
          Console.ReadLine();
      }
    }
}
```

noteghar.com

If you observe the above code, we declared and initialized string variables, formatted string using **Format** method, and used string literals to embed escape characters, etc., based on our requirements.

When you execute the above c# program, you will get the result as shown below.



This is how we can use strings in the c# programming language to represent a sequential collection of characters based on our requirements.

C# String Properties

The following table lists the available string properties in the c# programming language.

Property	Description
Chars	It helps us to get the characters from the current string object based on the specified position.
Length	It returns the number of characters in the current String object.

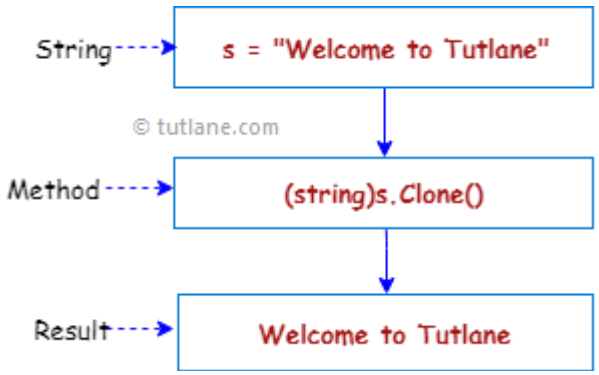
C# String Methods

In c#, the string class contains various methods to manipulate string objects based on our requirements.

1.C# String Clone Method

In c#, the string **Clone** method is used to clone and return a copy of the specified string object. The return value of the clone method is not an independent copy of the specified string because the clone method will return an existing string instance.

Following is the pictorial representation of using the string **Clone()** method to create and return a copy of the string object in the c# programming language.



If you observe the above diagram, we defined a string called "**Welcome to Tutlane**" and by using the string clone() method we cloned and returned a specified string as a result.

C# String Clone Method Syntax

Following is the syntax of defining a string **Clone** method to clone and return a copy of the string object in the c# programming language.

```
public object Clone()
```

If you observe syntax, the clone method will return an instance of a specified string as an object, and it won't accept any parameters.

C# String Clone Method Example

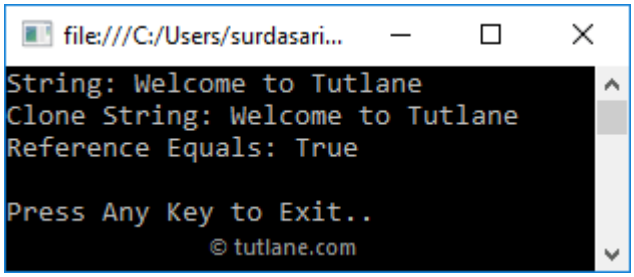
Following is the example of using a string **Clone()** method to clone a specified string object and return a copy of the same data in the c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         string msg = "Welcome to Tutlane";
         string msg1 = (string)msg.Clone();
         Console.WriteLine("String: {0}", msg);
         Console.WriteLine("Clone String: {0}", msg1);
         Console.WriteLine("Reference Equals: {0}", Object.Reference
Equals(msg, msg1));
         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

If you observe the above example, we used a string **Clone()** method to clone a specified string and return an instance of specified string as a result.

Here we used a **ReferenceEquals** method to check whether the original and cloned string objects are referring to the same instance or not.

When you execute the above c# program, you will get the result as shown below.



noteghar.com
flyghar.com

If you observe the above result, the string **Clone()** method has cloned the given string and returned the copy of the same string object.

This is how we can use Clone() method in the c# programming language based on our requirements.

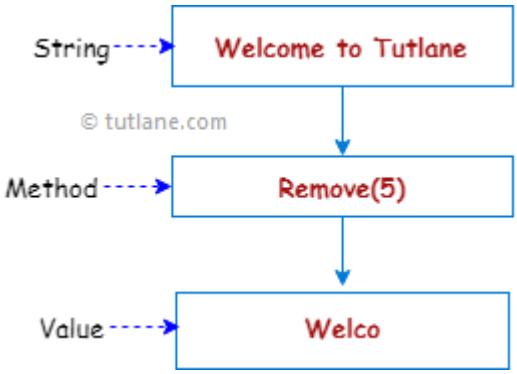
flyghar.com

2.C# String Remove Method

In c#, the string **Remove** method is used to delete characters in the string starting from the specified position and continues to the end of the string.

If we want to delete a particular length of substring, then we need to specify the starting position and length of characters to delete from the given string.

Following is the pictorial representation of using the string **Remove()** method to delete characters from the string based on the defined starting position in the c# programming language.



If you observe the above diagram, we removed all the characters from the defined string "**Welcome to Tutlane**", starting from position **5** using the **Remove** method.

noteghar.com

C# String Remove Method Syntax

Following is the syntax of defining a string **Remove** method to remove the characters starting from the defined position in a given string.

```
public string Remove(int startIndex)
public string Remove(int startIndex, int length)
```

If you observe syntaxes, the first syntax is used to delete all the string characters, starting from the defined position (**startIndex**). The second syntax is used to delete the particular length of a **substring** by defining the starting position and length of characters.

tiyghar.com

C# String Remove Method Example

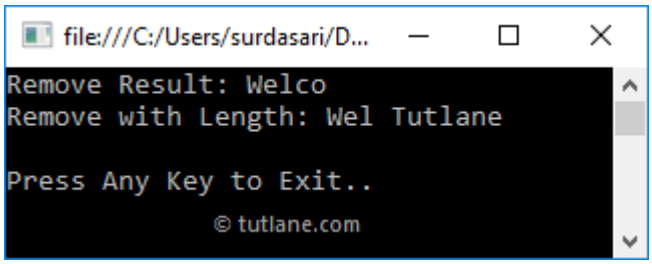
Following is the example of using a string **Remove()** method to delete characters in string based on the defined position in c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         string msg = "Welcome to Tutlane";
         Console.WriteLine("Remove Result: {0}", msg.Remove(5));
         Console.WriteLine("Remove with Length: {0}", msg.Remove(
3, 7));
         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

noteghar.com

If you observe the above example, we used a string **Remove()** method to remove characters starting from position **5** till the end of the defined string and another **Remove()** method to delete **7** characters starting from position **3**.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, the string **Remove()** method has deleted characters based on the defined positions and returned the remaining string characters.

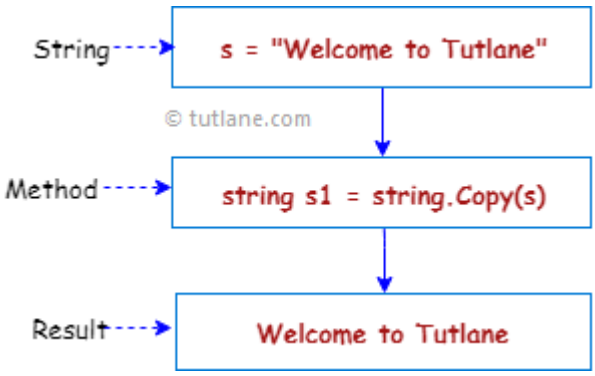
This is how you can use the string **Remove()** method to delete the required characters from the given string based on the defined position in the c# programming language.

3.C# String Copy Method

In c#, the string **Copy** method is useful to create a new instance of a string object with the same content as the specified string object. The copy method will return a string object that has the same value as the original string, but it represents a new object reference.

The main difference between the string clone() and **copy()** method is, the **copy** method will create a new instance of a specified string object with the same content, but the clone method will not create a new instance; instead, it will refer to an existing instance of the specified string object.

Following is the pictorial representation of using the string **Copy()** method to create a new string object with the same content in the c# programming language.



If you observe the above diagram, we defined a string called "**Welcome to Tutlane**" and using a string **copy()** method, We created a new string object with the same content of the specified string object.

C# String Copy Method Syntax

Following is the syntax of defining a string **Copy** method to clone and return a copy of the string object in c# programming language.

```
public static string Copy(string str)
```

If you observe syntax, the copy method will create and return a new instance of the string with the same content of specified string objects. It takes a string as a parameter to create a copy of the specified string.

C# String Copy Method Example

Following is the example of using a string **Copy()** method to create a new string object with the same content of specified string in the c# programming language.

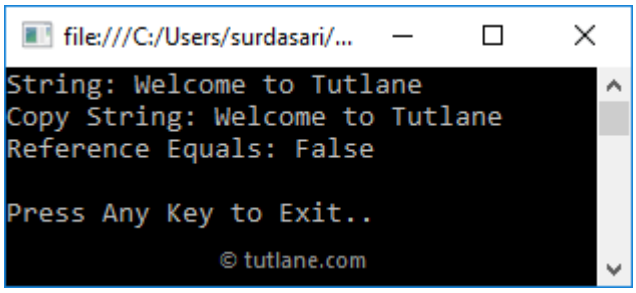
```
using System;

namespace Tutlane
{
   class Program
   {
```

```
        static void Main(string[] args)
        {
            string msg = "Welcome to Tutlane";
            string msg1 = string.Copy(msg);
            Console.WriteLine("String: {0}", msg);
            Console.WriteLine("Copy String: {0}", msg1);
            Console.WriteLine("Reference Equals: {0}", Object.Referenc
eEquals(msg, msg1));
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we used a string **Copy()** method to create a new string object with the content of the specified string object.

Here we used a **ReferenceEquals** method to check whether the original and copy method string objects are referring to the same instance or not.

When we execute the above c# program, we will get the result as shown below.



If you observe the above result, the string **Copy()** method has returned the copy of the same string object.

This is how we can use Copy() method in the c# programming language based on our requirements.

C# : CopyTo() Method

In C#, CopyTo() method is a string method. It is used to copy a specified number of characters from a specified position in the string and it copies the characters of this string into a array of Unicode characters.

Syntax:

```
public void CopyTo(int sourceIndex, char[] destination,
                   int destinationIndex, int count)
```

Explanation: CopyTo() method will copied the count characters from the sourceIndex position to the destinationIndex position of destination character array. This method accepts four parameters as follows :

1. **sourceInde**x : Index of String to be copied. And its type is System.Int32 .
2. **destination** : It is the array of Unicode characters to which characters will be copied. Its type is System.Char[].
3. **destinationIndex** : It is starting index of array from where the copy operation begins. Its type is System.Int32.
4. **count** : It is the number of characters which will copy to destination. Its type is System.Int32.

Examples :

Input : str = "GeeksForGeeks"

```
char [] Copystring = new char[15];
str.CopyTo(5, Copystring, 0, 3);
```

Output: For

Input : str2 = "GeeksForGeeks";

```
char [] Copystring = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'};
str2.CopyTo(8, Copystring, 6, 5);
```

Output: Hello Geeks

Below example programs illustrate the ToCopy() Method :

flyghar.com
noteghar.com

- Example 1:

```
// C# program to illustrate the
// ToCopy() string method
using System;

class Geeks {

    // Main Method
    public static void Main()
    {
        string str = "GeeksForGeeks";
        char[] dest = new char[15];

        // str index 5 to 5+3 has to
        // copy into Copystring
        // 3 is no. of character
        // 0 is start index of Copystring
```

tiyghar.com
noteghar.com

```
        str.CopyTo(5, dest, 0, 3);


        // Displaying String
        Console.Write("The Copied String in dest Variable is: ");
        Console.WriteLine(dest);
    }
}
```

- Output:
- The Copied String in dest Variable is: For
- Example 2:

```
// C# program to illustrate the
// ToCopy() string method
using System;

class Geeks {

    // Main Method
```

noteghar.com
tlyghar.com

```
    public static void Main()
    {
        string str2 = "GeeksForGeeks";
        char[] dest = {'H', 'e', 'l', 'l', 'o', ' ',
                       'W', 'o', 'r', 'l', 'd' };

        // str index 8 to 8 + 5 has
        // to copy into Copystring
        // 5 is no of character
        // 6 is start index of Copystring
        str2.CopyTo(8, dest, 6, 5);

        // Displaying the result
        Console.Write("String Copied in dest is: ");
        Console.WriteLine(dest);
    }
}
```

- Output:

flyghar.com
noteghar.com

- String Copied in dest is: Hello Geeks

C# String Compare Method

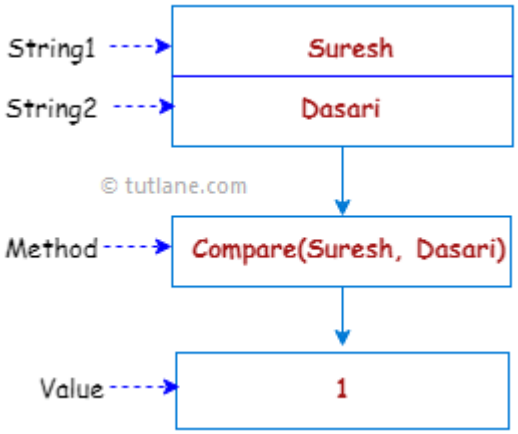
In c#, the string **Compare** method is used to compare two specified strings and return an integer value that indicates their relative position in the sort order.

flyghar.com

Here, the return integer value indicates the lexical relationship between the two specified strings. The following table lists the values that will be returned by the **Compare** method after comparing the two specified strings.

Condition	Value
strA == strB	0
strA > strB	1
strA < strB	-1

Following is the pictorial representation of comparing two strings using **Compare()** method in c# programming language.



noteghar.com

If you observe the above diagram, we are comparing two strings,"**Suresh**" and "**Dasari**" by using the **Compare** method and return an integer value "**1**".

C# String Compare Method Syntax

Following is the syntax of defining a **Compare** method to compare two specified strings in the c# programming language.

```
public int Compare(string string1, string string2)
```

If you observe syntax, the **Compare** method will compare a given two strings and return an integer value.

C# String Compare Method Example

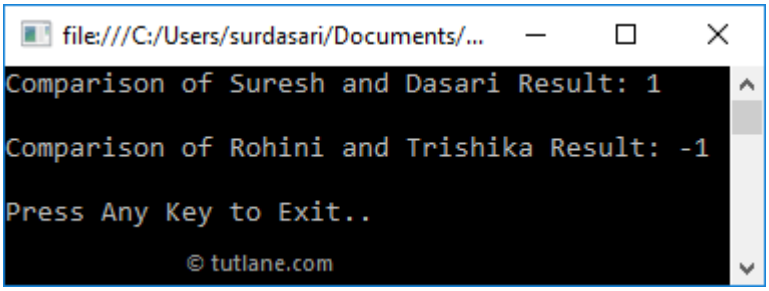
Following is the example of using the **Compare()** method to compare a given two strings and return an integer value in the c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         string str1 = "Suresh";
         string str2 = "Dasari";
         Console.WriteLine("Comparison of {0} and {1} Result: {2}",
str1, str2, string.Compare(str1, str2));

         string str3 = "Rohini";
         string str4 = "Trishika";
         Console.WriteLine("Comparison of {0} and {1} Result: {2}",
str3, str4, string.Compare(str3, str4));

         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
```

```
    }
  }
}
```

If you observe the above example, we used a Compare() method to compare two given strings and return an integer value.

When you execute the above c# program, you will get the result as shown below.



This is how you can use **Compare()** method to compare two given strings and return an integer value which indicates a lexical relationship between two strings in the c# programming language.

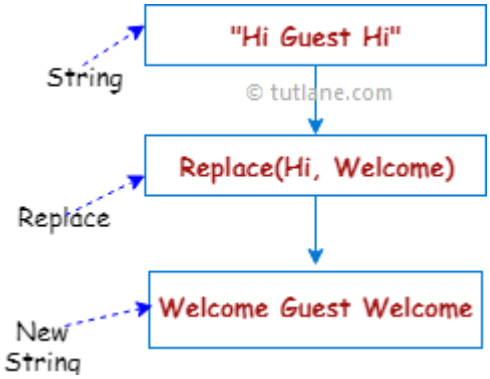
C# String Replace Method

In c#, the string **Replace** method is used to replace a specified string or a character in all occurrences of the given string.

The replace method will return a new string after replacing all occurrences of a specified string or a character.

Following is the pictorial representation of the **replace** method in the c# programming language.

noteghar.com
flyghar.com



If you observe the above diagram, we are replacing the "**Hi**" word in a given string "**Hi Guest Hi**" with "**Welcome**" using the **Replace** method. Once the replacement is done, the **Replace** method will return a new string like "**Welcome Guest Welcome**".

C# String Replace Method Syntax

Following is the syntax of defining a **Replace** method to replace a particular part of a string or a character in the c# programming language.

```
public string Replace(char oldchar, char newchar)
public string Replace(string oldstring, string newstring)
```

If you observe syntax, the Replace method will replace all occurrences of **oldchar** / **oldstring** with **newchar** / **newstring** in a given string and return it as a new string.

C# String Replace Method Example

Following is the example of using the **Replace()** method to replace a particular part of a string or a character c# programming language.

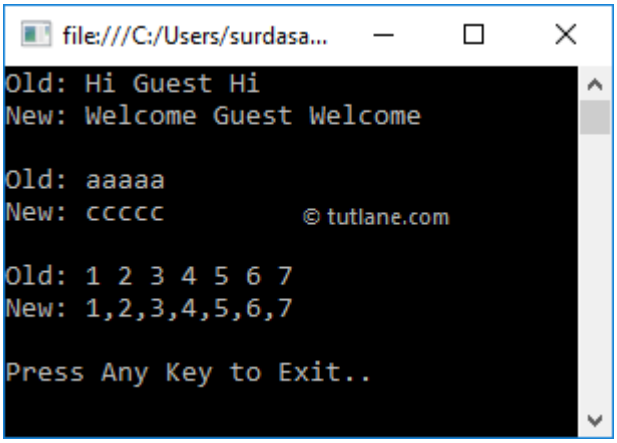
```
using System;

namespace Tutlane
{
   class Program
```

```
  {
    static void Main(string[] args)
    {
        string msg = "Hi Guest Hi";
        string nmsg = msg.Replace("Hi", "Welcome");
        Console.WriteLine("Old: {0}", msg);
        Console.WriteLine("New: {0}", nmsg);

        string x = "aaaaa";
        string nx = x.Replace("a", "b").Replace("b", "c");
        Console.WriteLine("Old: {0}", x);
        Console.WriteLine("New: {0}", nx);

        string y = "1 2 3 4 5 6 7";
        string ny = y.Replace(" ", ",");
        Console.WriteLine("Old: {0}", y);
        Console.WriteLine("New: {0}", by);

        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
  }
}
```

If you observe the above example, we used a **Replace()** method to replace a particular part of a string or a character in a given string and returning a new string.

When you execute the above c# program, you will get the result as shown below.

noteghar.com
tiyghar.com



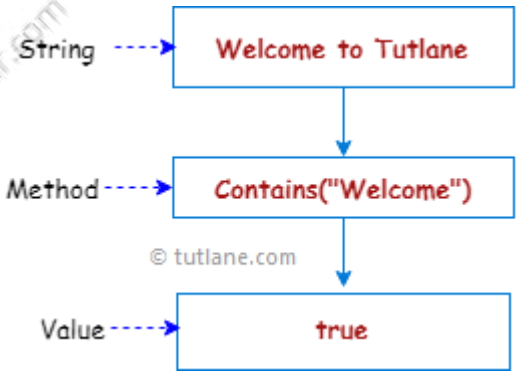
This is how we can use Replace() method to replace a particular part of a string or a character in the c# programming language.

C# String Contains Method

In c#, the string **Contains** method is used to check whether the specified substring exists in the given string or not, and it will return a boolean value.

If a substring exists in a string, then the **contains** method will return **true** otherwise, it will return **false**.

Following is the pictorial representation of checking whether the substring exists in a given string or not using the **Contains()** method in the c# programming language.



If you observe the above diagram, we are checking whether the "**Welcome**" text exists in the "**Welcome to Tutlane**" string or not using the **Contains** method, and it returns a **true** because the "**Welcome**" substring exists in the given string.

flyghar.com

C# String Contains Method Syntax

Following is the syntax of defining a **Contains** method to check whether the substring exists in a given string or not in the c# programming language.

```
public bool Contains(string value)
```

If you observe syntax, the **Contains** method will check whether the substring value exists or not, and it will return a boolean value.

C# String Contains Method Example

Following is the example of using **Contains()** method to check whether the given value occurs within the string or not in the c# programming language.

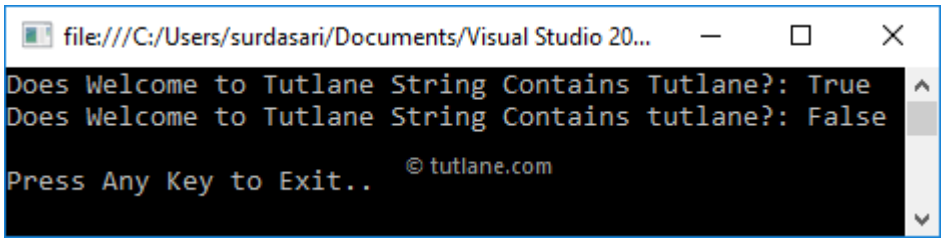
```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         string msg = "Welcome to Tutlane";
         string subtxt = "Tutlane";
         Console.WriteLine("Does {0} String Contains {1}?: {2}", msg
, subtxt, msg.Contains(subtxt));
         string subtxt1 = "tutlane";
         Console.WriteLine("Does {0} String Contains {1}?: {2}", msg
```

noteghar.com

```
, subtxt1, msg.Contains(subtxt1));
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we used a **Contains()** method to determine whether the given substring value is found in string or not and returning a bool value.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, the **Contains()** method performing a **case sensitive** operation that's the reason for the "**Tutlane**" substring it returns "**true**" and for "**tutlane**" substring, it returns **false**.

C# String Contains Case Insensitive

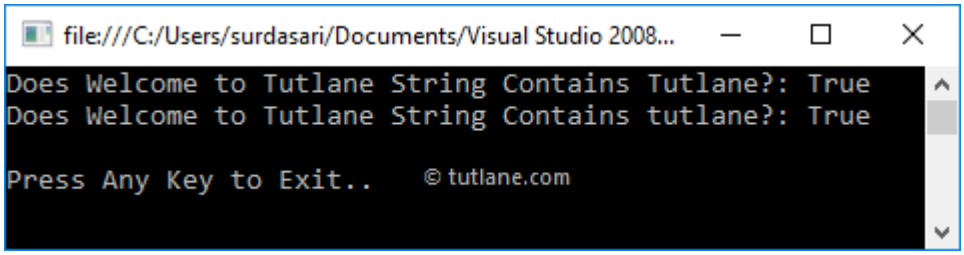
To perform a case-insensitive string comparison, you need to use the string **IndexOf** method. Following is the example of performing a case insensitive search in the c# programming language.

```
using System;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
```

```
        string msg = "Welcome to Tutlane";
        string subtxt = "Tutlane";
        Console.WriteLine("Does {0} String Contains {1}?: {2}", msg
, subtxt, msg.Contains(subtxt));
        string subtxt1 = "tutlane";
        StringComparison comp = StringComparison.OrdinalIgnoreCa
se;
        bool result = msg.IndexOf(subtxt1, comp) > 0 ? true : false;
        Console.WriteLine("Does {0} String Contains {1}?: {2}", msg
, subtxt1, result);
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
  }
}
```

If you observe the above example, we used a string **IndexOf** method to perform a case insensitive search. We will learn more about the string **IndexOf** method in the next chapters.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, for both substrings "**Tutlane**" and "**tutlane**" we got a result as **true**.

This is how we can use **Contains()** method to check whether the specified substring exists in the given string or not in the c# programming language.

C# String IndexOf Method

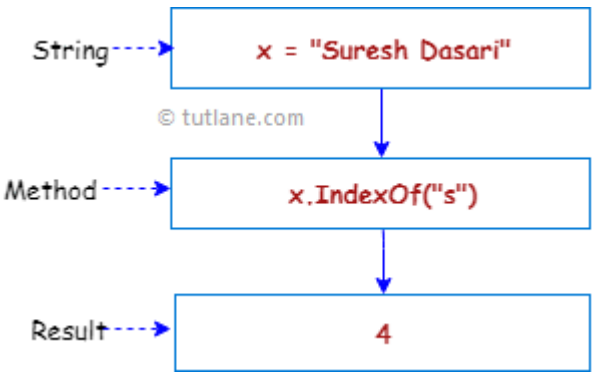
In c#, the string **IndexOf** method is useful to return an index of the first occurrence of the specified character in the given string.

Generally, in c# the string **IndexOf** method will start searching for the specified character starting from the Zero position and return the first occurrence of the specified character in the given string.

flyghar.com

If you want to change the specified character search starting position and to examine the only specified number of character positions in a string, then we can do it by specifying the required start position and number of character positions to search in the string **IndexOf** method.

Following is the pictorial representation of searching for a specified character position in the given string using the **IndexOf()** method in the c# programming language.



If you observe the above diagram, we are trying to get the character "**s**" position in the given string "**Suresh Dasari**" and it returned the position as **4**.

In c#, the string **IndexOf()** method will perform a **case-sensitive** search to get the specified character position that's the reason we got the result as **4** even the character "**S**" exists at the starting position **0**.

noteghar.com

C# String IndexOf Method Syntax

Following is the syntax of defining a string **IndexOf** method to get the specified character position in the c# programming language.

```
public int IndexOf(char ch)
public int IndexOf(char ch, int startIndex)
public int IndexOf(char ch, int startIndex, int count)
public int IndexOf(char ch, StringComparison comparisonType)
public int IndexOf(string str)
public int IndexOf(string str, int startIndex)
public int IndexOf(string str, int startIndex, int count)
public int IndexOf(string str, StringComparison comparisonType)
```

If you observe syntaxes, the **IndexOf** method will return the index position of a specified character or a string in the given string and we can change the starting position of characters search and length of characters to search based on our requirements.

The string **IndexOf** method will return an index position of a specified character if that character is found. In case the specified character is not found, then it will return **-1**.

C# String IndexOf Method Example

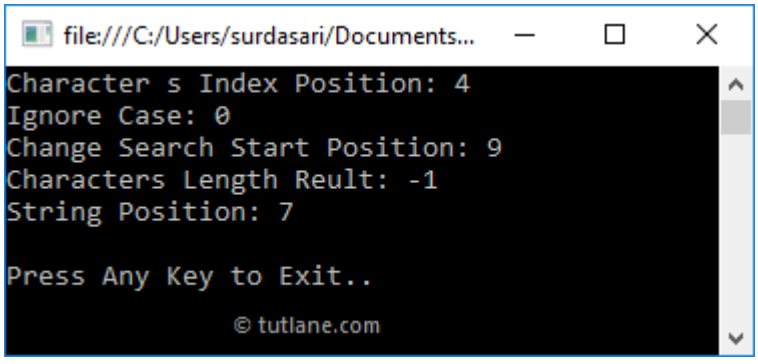
Following is the example of using the string **IndexOf()** method to return an index position of a specified character in the given string c# programming language.

```
using System;
using System.Collections.Generic;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
```

```
        {
            string name = "Suresh Dasari";
            Console.WriteLine("Character s Index Position: {0}", name.I
ndexOf("s"));
            Console.WriteLine("Ignore Case: {0}", name.IndexOf("s", St
ringComparison.OrdinalIgnoreCase));
            Console.WriteLine("Change Search Start Position: {0}", nam
e.IndexOf("s", 5));
            Console.WriteLine("Characters Length Reult: {0}", name.Ind
exOf("s", 5, 3));
            Console.WriteLine("String Position: {0}", name.IndexOf("Da
sa"));
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we used an **IndexOf()** method to find the index position of a defined character in the string "**Suresh Dasari**" with multiple conditions.

When you execute the above c# program, you will get the result as shown below.



This is how you can use the string **IndexOf()** method to get the position of a specified character in a given string in the c# programming language.

C# StringBuilder with Examples

In c#, **StringBuilder** is a class that is useful to represent a mutable string of characters, and it is an object of the **System.Text** namespace.

Like string in c#, we can use a **StringBuilder** to create variables to hold any text, a sequential collection of characters based on our requirements.

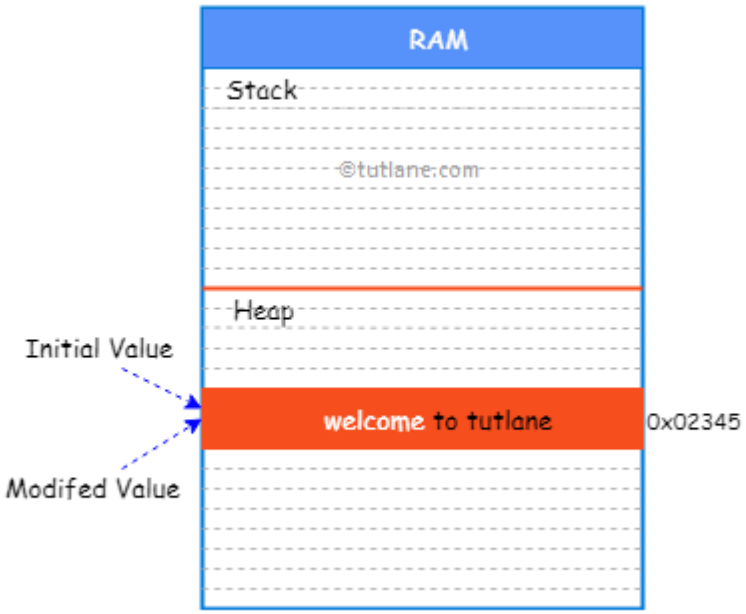
In c#, both string and StringBuilder will represent a sequence of characters and perform the same kind of operations, but the only difference is strings are **immutable,** and StringBuilder is **mutable**.

Generally, in c# the string object cannot be modified once it is created. If any changes are made to the string object, like add or modify an existing value, it will simply discard the old instance in memory and create a new instance to hold the new value. If we are doing repeated modifications on the string object, it will affect the application's performance. To know more about strings, check strings in c# with examples.

To solve this problem, c# introduced an alternative called **StringBuilder**, which is a **mutable** string class. **Mutability** means once an instance of the class is created, then the same instance will be used to perform any operations like inserting, appending, removing, or replacing the characters instead of creating a new instance for every time.

In c#, the StringBuilder is a dynamic object which will expand a memory dynamically to accommodate the modifications of string instead of creating a new instance in the memory.

Following is the pictorial representation of memory allocation for **StringBuilder** object in c# programming language.



C# StringBuilder Declaration and Initialization

As discussed, the **StringBuilder** class is an object of the **System.Text** namespace, so to use StringBuilder in our application, we need to import the **System.Text** namespace.

In c#, the StringBuilder declaration and initialization will be the same as the class. The following are the different ways of declaring and initializing a stringbuilder in the c# programming language.

```
StringBuilder sb = new StringBuilder();

//or

StringBuilder sb = new StringBuilder("Welcome to Tutlane");
```

If you observe the above code snippet, we created an instance of the **StringBuilder** class by defining our variable with overloaded constructor methods.

C# StringBuilder Capacity

As discussed, the **StringBuilder** is a dynamic object which will expand dynamically to accommodate the number of characters based on the string modifications. We can also specify an initial capacity of characters, the StringBuilder can hold by passing an int value using one of the overloaded constructors or
StringBuilder **Capacity** property.

For example, we created a StringBuilder by specifying the capacity of **25** characters and appending a string whose length is greater than the capacity of **25** characters. In this case, the new space will be allocated automatically, and the capacity of StringBuilder will be doubled.

Following is the example of specifying the initial capacity of characters the StringBuilder can hold in the c# programming language.

```
StringBuilder sb = new StringBuilder(25);

//or

StringBuilder sb = new StringBuilder("Welcome to Tutlane",25);

//or

sb.Capacity = 25;
```

In c#, whenever the defined capacity of StringBuilder is lesser than the appended string value, then the current capacity of StringBuilder automatically will increase to match the appended string value.

The default capacity of StringBuilder is **16** characters, and its maximum capacity is more than **12 billion** characters.

C# StringBuilder Methods

The following table lists the important StringBuilder methods, which we can use to modify the contents of StringBuilder.

flyghar.com

Method	Description
StringBuilder.Append	This method will append the given string value to the end of the current StringBuilder.
StringBuilder.AppendFormat	It will replace a format specifier passed in a string with formatted text.
StringBuilder.Insert	It inserts a string at the specified index of the current StringBuilder.
StringBuilder.Remove	It removes a specified number of characters from the current StringBuilder.
StringBuilder.Replace	It replaces a specified character at a specified index.

noteghar.com

C# StringBuilder Append Method

The **Append** method is used to add or append a string object at the end of the string represented by the StringBuilder.

Following is the example of initializing a StringBuilder with some text and appending a required text at the end of the string object.

```
StringBuilder sb = new StringBuilder("Suresh");
sb.Append(", Rohini");
sb.Append(", Trishika");
Console.WriteLine(sb);
// Output: Suresh, Rohini, Trishika
```

flyghar.com

C# StringBuilder AppendFormat Method

The **AppendFormat** method is used to add or append string objects by formatting them into a specified format at the end of the string represented by the StringBuilder.

Following is the example of initializing a StringBuilder with some text and appending a formatted text at the end of the string object.

```
int amount = 146;
StringBuilder sb = new StringBuilder("Total");
sb.AppendFormat(": {0:c}", amount);
Console.WriteLine(sb);
// Output is Total: $146.00
```

C# StringBuilder Insert Method

noteghar.com

The **Insert** method is used to insert a string at the specified index position of the current StringBuilder object.

Following is an example of initializing a StringBuilder with some text and inserting a string at the specified index position of the StringBuilder object.

```
StringBuilder sb = new StringBuilder("Welcome Tutlane");
sb.Insert(8, "to ");
Console.WriteLine(sb);
// Output: Welcome to Tutlane
```

flyghar.com

C# StringBuilder Remove Method

The **Remove** method is used to remove a specified number of characters from the current StringBuilder object, starting from the specified index position.

Following is the example of removing a specified number of characters from the StringBuilder object, starting from the specified index position.

```
StringBuilder sb = new StringBuilder("Welcome to Tutlane");
sb.Remove(8, 3);
Console.WriteLine(sb);
// Output: Welcome Tutlane
```

C# StringBuilder Replace Method

noteghar.com

The **Replace** method is used to replace all occurrences of specified string characters in the current StringBuilder object with a specified replacement string character.

Following is the example of replacing a specified number of characters from the StringBuilder object with specified replace characters.

```
StringBuilder sb = new StringBuilder("Welcome to Tutlane");
sb.Replace("Tutlane","C#");
Console.WriteLine(sb);
// Output: Welcome to C#
```

In c#, the StringBuilder has another method called **AppendLine(),** which is used to add a new line at the end of the string.

tiyeghar.com

C# StringBuilder Example

Following is the example of using **StringBuilder** to insert or append or replace or remove a particular string text in c# programming language.

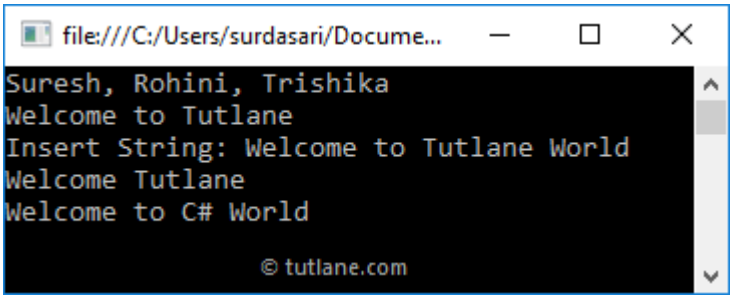
```
using System;
using System.Text;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         StringBuilder sb = new StringBuilder("Suresh");
         sb.Append(", Rohini");
         sb.Append(", Trishika");
         sb.AppendLine();
         sb.Append("Welcome to Tutlane");
         Console.WriteLine(sb);
```

noteghar.com

```
            StringBuilder sb1 = new StringBuilder("Welcome World");
            sb1.Insert(8, "to Tutlane ");
            Console.WriteLine("Insert String: " + sb1);

            StringBuilder sb2 = new StringBuilder("Welcome to Tutlane")
;
            sb2.Remove(8, 3);
            Console.WriteLine(sb2);

            StringBuilder sb3 = new StringBuilder("Welcome to Tutlane
World");
            sb3.Replace("Tutlane", "C#");
            Console.WriteLine(sb3);
            Console.ReadLine();
        }
    }
}
```

If you observe the above code, to use StringBuilder in our application, we imported a **System.Text** namespace and used different methods of StringBuilder to make required modifications based on our requirements.

When you execute the above c# program, you will get the result as shown below.



This is how we can use StringBuilder in our applications to make the required modifications to the string objects based on our requirements.

C# Convert StringBuilder to String

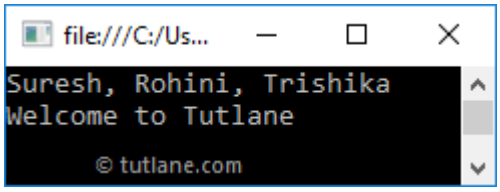
In c#, you can convert a StringBuilder object to a string by calling by **StringBuilder.ToString()** method.

Following is the example of converting a StringBuilder object to a string using the **ToString()** method in c# programming language.

```
using System;
using System.Text;

namespace Tutlane
{
   class Program
   {
      static void Main(string[] args)
      {
         StringBuilder sb = new StringBuilder("Suresh");
         sb.Append(", Rohini");
         sb.Append(", Trishika");
         sb.AppendLine();
         sb.Append("Welcome to Tutlane");
         Console.WriteLine(sb.ToString());
         Console.ReadLine();
      }
   }
}
```

If you observe the above code, we are converting StringBuilder object (**sb**) to string object using **sb.ToString()** method.

When you execute the above c# program, you will get the result as shown below.



This is how we can convert the StringBuilder object to string based on our requirements.

flyghar.com

Use StringBuilder in C#

The following are the important points we need to remember while using a StringBuilder in the c# programming language.

- We need to consider using StringBuilder only when we are unknown about the number of changes that will make to a string at design time; otherwise, StringBuilder will offer a negligible or no performance improvement over a string.
- When we perform a fixed number of concatenation operations on a string, it's better to avoid using StringBuilder because it will offer negligible performance.
- When we expect our application will make a significant number of changes to a string, we need to use StringBuilder instead of a string.

noteghar.com

Properties of Access Modifiers

Access modifiers defines the access level of a property whether a property can be accessed by any caller program, within an assembly, or just within a class. The following table describes access level modifiers.

• **public** - The type or member can be accessed by any other code in the same assembly or another assembly that references it.

• **Private** - The type or member can be accessed only by code in the same class or struct.

• **protected** - The type or member can be accessed only by code in the same class, or in a class that is derived from that class.

• **internal** - The type or member can be accessed by any code in the same assembly, but not from another assembly.

• **protected internal** - The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.

• **private protected** - The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

noteghar.com

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

noteghar.com
noteghar.com

C# Classes and Objects with Examples

In c#, **Classes and Objects** are interrelated. The **class** in c# is nothing but a collection of various data members (fields, properties, etc.) and member functions. The **object** in c# is an instance of a **class** to access the defined properties and methods.

We will now learn the classes and objects in c# and how to use them in c# applications with examples.

C# Class

In c#, **Class** is a data structure, and it will combine various types of data members such as fields, properties, member functions, and events into a single unit.

Declaring a Class in C#

In c#, classes are declared by using class keyword. Following is the declaration of class in c# programming language.

```
public class users {

// Properties, Methods, Events, etc.

}
```

If you observe the above syntax, we defined a class "**users**" using class keyword with public access modifier. Here, the public access specifier will allow the users to create objects for this class, and inside of the body class, we can create required fields, properties, methods, and events to use in our applications.

Now we will see how to create a class in c# programming language with example.

C# Class Example

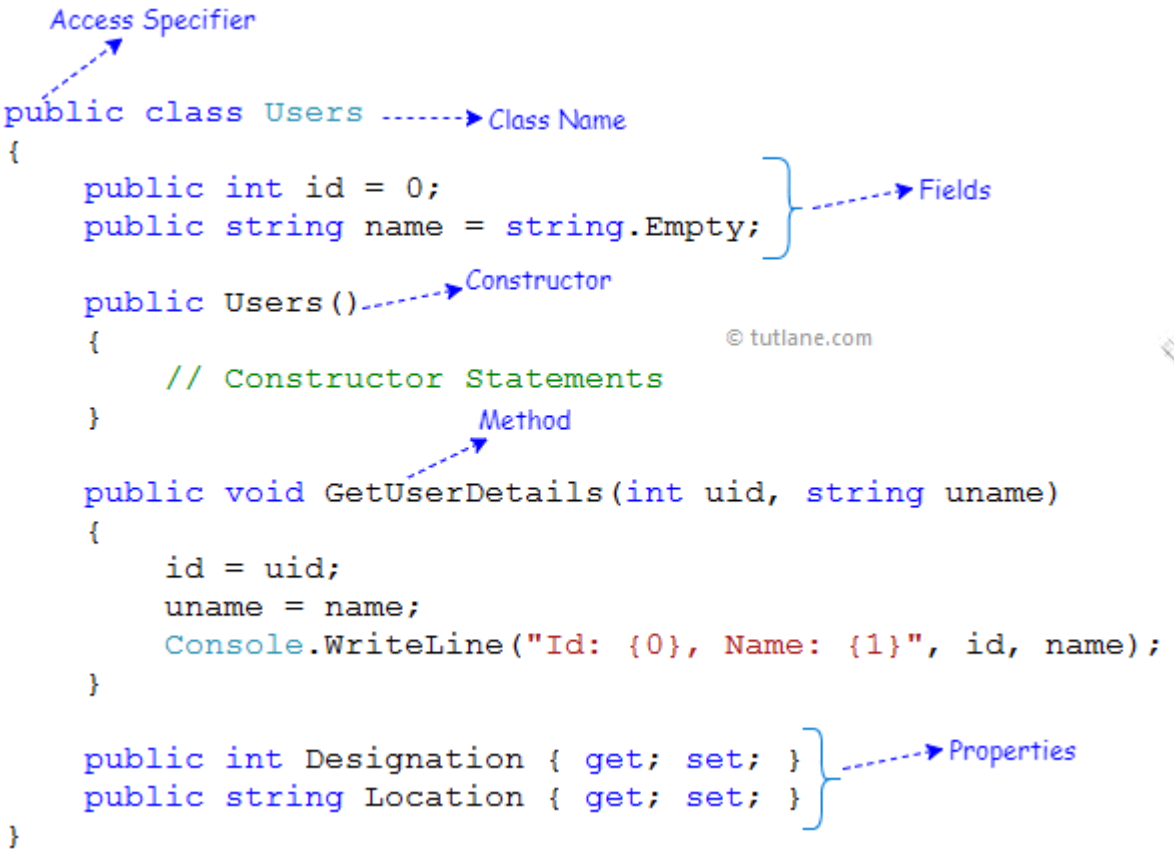
Following is the example of creating a class in c# programming language with various data members and member functions.

```
public class Users
{
    public int id = 0;
    public string name = string.Empty;
    public Users()
    {
      // Constructor Statements
    }
    public void GetUserDetails(int uid, string uname)
    {
        id = uid;
        uname = name;
        Console.WriteLine("Id: {0}, Name: {1}", id, name);
    }
    public int Designation { get; set; }
    public string Location { get; set; }
}
```

If you observe the above c# class example, we defined a class "**Users**" with various data members and member functions based on our requirements.

Following is the detailed description of various data members used in the above c# class example.

noteghar.com



If you observe the above image, we used various data members like access modifiers, fields, properties, methods, constructors, etc., in our c# class based on our requirements.

noteghar.com
flyghar.com

C# Object

In c#, **Object** is an instance of a **class** that can be used to access the data members and member functions of a **class**.

Creating Objects in C#

Generally, we can say that objects are the concrete entities of classes. In c#, we can create objects by using a **new** keyword followed by the class's name like as shown below.

```
Users user = new Users();
```

If you observe the above example, we created an instance (**user**) for the class (**Users**), which we created in the previous section. Now the instance "**user**" is a reference to an object that is based on **Users**. Using the object name "**user**" we can access all the data members and member functions of the **Users** class.

C# Objects Example

Following is the example of creating objects in the c# programming language.

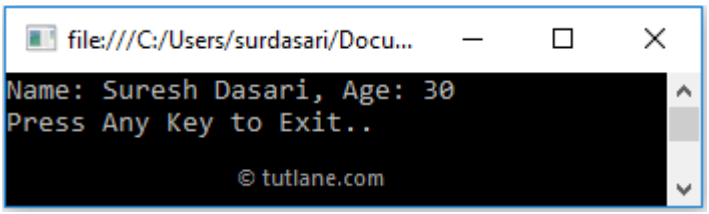
```
using System;

namespace Tutlane
{
  class Program
  {
    static void Main(string[] args)
    {
      Users user = new Users("Suresh Dasari", 30);
      user.GetUserDetails();
      Console.WriteLine("Press Enter Key to Exit..");
      Console.ReadLine();
    }
  }
```

noteghar.com

```
  public class Users
  {
     public string Name { get; set; }
     public int Age { get; set; }
     public Users(string name, int age)
     {
        Name = name;
        Age = age;
     }
     public void GetUserDetails()
     {
        Console.WriteLine("Name: {0}, Age: {1}", Name, Age);
     }
  }
}
```

If you observe the above example, we created a new class called "**Users**" with a required data members and member functions. To access **Users** class methods and properties, we created an object (**user**) for the **Users** class and performing the required operations.

When we execute the above c# program, we will get the result as shown below.



This is how we can create and use classes and objects in c# applications based on our requirements.

noteghar.com
tlyghar.com

Collections in C#

In c#, the **collection** is a class that is useful to manage a group of objects in a flexible manner to perform various operations like insert, update, delete, get, etc. on the object items in a dynamic manner based on our requirements.

Generally, while working in c# applications we will get a requirement like to create or manage a group of related objects. In that case, we have two ways to create group objects in c# i.e. by using the arrays and collections.

In the previous section, we learned about arrays in c# but those are useful only when we are working with a fixed number of strongly-typed objects. So, to solve this problem Microsoft has introduced collections in c# to work with a group of objects which can grow or shrink dynamically based on our requirements.

In c#, the collection is a class so we just need to declare an instance of the class before we perform any operations like add, delete, etc. on the defined collection and the collections are implemented by using the **IEnumerable** interface so we can access collection items by using a foreach loop.

C# Collection Types

In c#, we have a different type of collection classes are available, those are

- **Non-Generic** (System.Collections)
- **Generic** (System.Collections.Generic)
- **Concurrent** (System.Collections.Concurrent)

C# Non-Generic Collections

In c#, non-generic collection classes are useful to store elements of different data types and these are provided by **System.Collections** namespace. Now, these collection classes are legacy types so whenever possible try to use generic collections (**System.Collections.Generic**) or concurrent collections (**System.Collections.Concurrent**).

Following are the different type of non-generic collection classes which are provided by **System.Collections** namespace.

Class	Description
ArrayList	It is useful to represent an array of objects whose size is dynamically increased as required.
Queue	It is useful to represent a FIFO (First in, First Out) collection of objects.
Stack	It is useful to represent a LIFO (Last in, First Out) collection of objects.
Hashtable	It is useful to represent a collection of key/value pairs that are organized based on the hash code of the key.

C# Generic Collections

In c#, generic collections will enforce a type safety so we can store only the elements which are having the same data type and these are provided by **System.Collections.Generic** namespace.

Following are the different type of generic collection classes which are provided by **System.Collections.Generic** namespace.

Class	Description
List	It is useful to represent a list of objects that can be accessed by index.
Queue	It is useful to represent a FIFO (First in, First Out) collection of objects.
Stack	It is useful to represent a LIFO (Last in, First Out) collection of objects.
SortedList<K,V>	It is useful to represent a collection of key/value pairs that are sorted by a key.
Dictionary<K,V>	It is useful to represent a collection of key/value pairs that are organized based on the key.

C# Concurrent Collections

In c#, concurrent collections are useful to access collection items from multiple threads and these are available from .NET Framework 4 with **System.Collections.Concurrent** namespace.

In case, if we are using multiple threads to access a collection concurrently, then we need to use concurrent collections instead of **non-generic** and **generic** collections.

Following are the different type of concurrent collection classes which are provided by **System.Collections.Concurrent** namespace.

Class	Description
BlockingCollection	It is useful to provide blocking and bounding capabilities for thread-safe collections.
ConcurrentBag	It is useful to represent a thread-safe, unordered collection of objects.
ConcurrentDictionary<K,V>	It is useful to represent a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently.
ConcurrentQueue	It is useful to represent a thread-safe FIFO (First in, First Out) collection.
ConcurrentStack	It is useful to represent a thread-safe LIFO (Last in, First Out) collection.
Partitioner	It is useful to provide partitioning strategies for arrays, lists, enumerables.
OrderablePartitioner	It is useful to provide a specific way of splitting an orderable data source into multiple partitions.

noteghar.com
flyghar.com

C# Arraylist

In c#, **ArrayList** is useful to store elements of different data types and the size of ArrayList can grow or shrink dynamically based on the need of our application like adding or removing elements from ArrayList.

In c#, arraylists are same as arrays but the only difference is arrays are used to store a fixed number of same data type elements.

flyghar.com

C# ArrayList Declaration

In c#, ArrayList is a **non-generic** type of collection and it is provided by **System.Collections** namespace.

As discussed, the collection is a class so to define an arraylist, you just need to declare an instance of the arraylist class before we perform any operations like add, delete, etc. like as shown below.

```
ArrayList arrList = new ArrayList();
```

If you observe the above arraylist declaration, we created a new arraylist (**arrList**) with an instance of arraylist class without specifying any size.

C# ArrayList Properties

The following are the some of commonly used properties of an arraylist in c# programming language.

Property	Description
Capacity	It is useful to get or set the number of elements an arraylist can contain.
Count	It is useful to get the number of elements in an arraylist.
IsFixedSize	It is useful to get a value to indicate that the arraylist has a fixed size or not.
IsReadOnly	It is useful to get a value to indicate that the arraylist is read-only or not.
Item	It is used to get or set an element at the specified index.

notesghar.com

Property	Description
IsSynchronized	It is used to get a value to indicate that access to arraylist is synchronized (thread-safe) or not.

C# ArrayList Methods

flyghar.com

The following are some of the commonly used methods of an arraylist to add, search, insert, delete or sort elements of arraylist in c# programming language.

Method	Description
Add	It is useful to add an element at the end of arraylist.
AddRange	It is useful to add all the elements of the specified collection at the end of arraylist.
Clear	It will remove all the elements in arraylist.
Clone	It will create a shallow copy of arraylist.
Contains	It is useful to determine whether the specified element exists in an arraylist or not.
CopyTo	It is useful to copy all the elements of an arraylist into another compatible array.
GetRange	It is useful to return a specified range of arraylist elements starting from the specified index.
IndexOf	It is useful to search for a specified element and return a zero-based index if found otherwise return -1 if no element found.
Insert	It is useful to insert an element in arraylist at the specified index.
InsertRange	It is useful to insert all the elements of the specified collection into arraylist starting from the specified index.

noteghar.com

Method	Description
Remove	It is useful to remove the first occurrence of specified element from the arraylist.
RemoveAt	It is useful to remove an element from arraylist based on the specified index position.
RemoveRange	It is useful to remove a range of elements from an arraylist.
Reverse	It reverses the order of arraylist elements.
Sort	It sorts the elements in an arraylist.
ToArray	It copies all the elements of an arraylist to a new array object.

flyghar.com

C# Arraylist Example

Following is the example of using an arraylist in c# programming language.

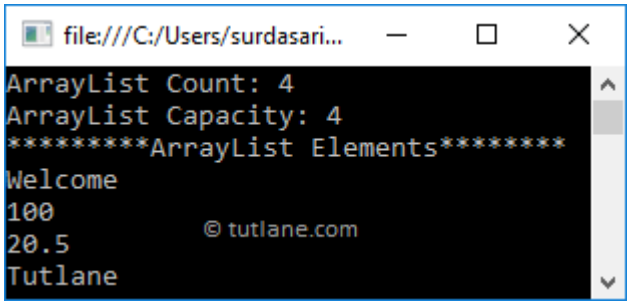
```
using System;
using System.Collections;
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrlist = new ArrayList();
            arrlist.Add("Welcome");
            arrlist.Add(100);
            arrlist.Add(20.5);
            arrlist.Add("Tutlane");
            Console.WriteLine("ArrayList Count: " + arrlist.Count);
            Console.WriteLine("ArrayList Capacity: " + arrlist.Capacity
);
            Console.WriteLine("*********ArrayList Elements********");
            foreach (var item in arrlist)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
```

noteghar.com

```
        }
    }
}
```

If you observe above example, we are able to define a new arraylist (**arrlist**) collection by using **System.Collections** namespace. Here, we are binding a different data type values to newly created arraylist (**arrlist**) by using **Add** method and with **Count** & **Capacity** properties we are able to access number of elements in an arraylist.

When you execute the above c# program, you will get the result as shown below.



This is how you can use arraylist in c# to hold a group of different data type objects based on our requirements.

The following are some of the useful examples to work with arraylist in c# programming language.

C# Add Elements to ArrayList

In c#, by using the **Add** or **AddRange** method, we can add a different data type elements to an arraylist.

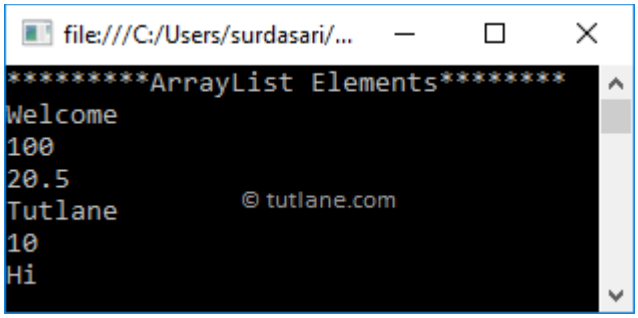
Following is the example of adding elements to arraylist using **Add** or **AddRange** methods in c#.

```
using System;
using System.Collections;
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // creating arraylist
            ArrayList arrlist = new ArrayList();
            arrlist.Add("Welcome");
            arrlist.Add(100);
            arrlist.Add(20.5);
            arrlist.Add("Tutlane");
```

```
            //creating arraylist
            ArrayList arrlist2 = new ArrayList() { 10, "Hi" };
            // adding arrlist2 to arrlist
            arrlist.AddRange(arrlist2);
            Console.WriteLine("*********ArrayList Elements********");
            foreach (var item in arrlist)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created a multiple arraylists (**arrlist**, **arrlist2**) and adding an elements to the arraylists in different ways.

When you execute the above c# program, you will get the result as shown below.



C# Access ArrayList Elements

In c#, you have different ways to access arraylist elements i.e. either by using index positions or by iterating through an arraylist using for / foreach loops.

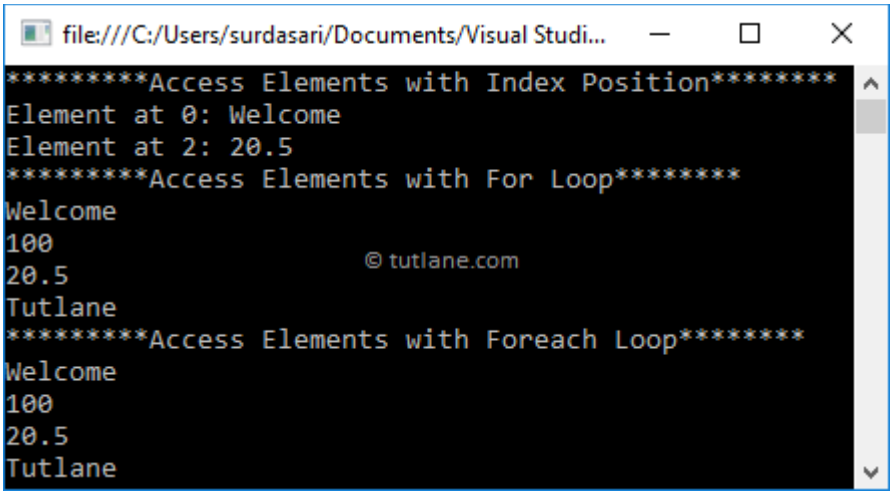
Following is the example of accessing arraylist elements in different ways.

```
using System;
using System.Collections;
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // creating arraylist
            ArrayList arrlist = new ArrayList();
            arrlist.Add("Welcome");
            arrlist.Add(100);
            arrlist.Add(20.5f);
```

```
            arrlist.Add("Tutlane");
            // accessing first element
            string msg = (string)arrlist[0];
            // accessing third element
            float num = (float)arrlist[2];
            Console.WriteLine("*********Access Elements with Index Posi
tion********");
            Console.WriteLine("Element at 0: " + msg);
            Console.WriteLine("Element at 2: " + num);
            Console.WriteLine("*********Access Elements with For Loop**
******");
            // for loop to access arraylist elements
            for (int i = 0; i < arrlist.Count; i++)
            {
                Console.WriteLine(arrlist[i]);
            }
            Console.WriteLine("*********Access Elements with Foreach Lo
op********");
            // foreach loop to access arraylist elements
            foreach (var item in arrlist)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we are accessing arraylist elements in different ways by using index positions, for and foreach loops based on our requirements.

When you execute the above c# program, you will get the result like as shown below.



noteghar.com

C# Insert Elements into ArrayList

In c#, we can insert elements into arraylist either by using **Insert()** or **InsertRange()** methods.

Following is the example of inserting elements into an arraylist by using **Insert** and **InsertRange** methods in c#.

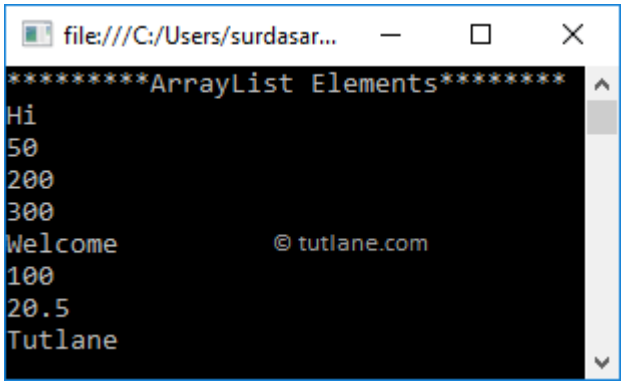
flyghar.com

```
using System;
using System.Collections;
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // creating arraylist
            ArrayList arrlist = new ArrayList();
            arrlist.Add("Welcome");
            arrlist.Add(100);
            arrlist.Add(20.5f);
            arrlist.Add("Tutlane");
            // inserting elements into arraylist
            arrlist.Insert(0, "Hi");
            arrlist.Insert(1, 50);
            ArrayList arrlist2 = new ArrayList() { 200, 300 };
            // inserting arrlist2 into arrlist at position 2
            arrlist.InsertRange(2, arrlist2);
            Console.WriteLine("*********ArrayList Elements********");
            foreach (var item in arrlist)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we inserted an elements into arraylist at different index positions (**0**, **1**) by using **Insert()** method. Same way, by using **InsertRange()** method we are inserting all the element of newly created arraylist (**arrlist2**) into **arrlist** at index position **2**.

noteghar.com

When you execute the above c# program, you will get the result as shown below.



tiyghar.com

C# Remove Elements from ArrayList

In c#, you have a different methods to remove an elements from arraylist i.e. either by
using **Remove()** or **RemoveAt()** or **RemoveRange()** methods.

Following is the example of deleting elements from arraylist in c# programming language.

```
using System;
using System.Collections;
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // creating arraylist
            ArrayList arrlist = new ArrayList();
            arrlist.Add("Welcome");
            arrlist.Add(100);
            arrlist.Add(20.5f);
            arrlist.Add("Tutlane");
            arrlist.Add(200);
            arrlist.Add(300);
            arrlist.Add(400);
            // Removing an element which is having a value 20.5f
            arrlist.Remove(20.5f);
            // Removing an element at index 0
            arrlist.RemoveAt(0);
            // Removing 2 elements starting from index 3
            arrlist.RemoveRange(3, 2);
            Console.WriteLine("*********ArrayList Elements********");
            foreach (var item in arrlist)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
```

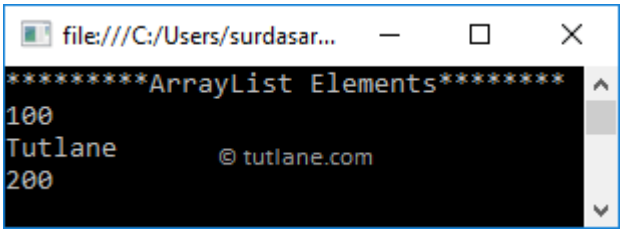
noteghar.com

```
    }
}
```

If you observe above example, we used a **Remove()** method to delete particular value of element from arraylist and used a **RemoveAt()** method to delete an element at specified index position.

Same way, we used the **RemoveRange()** method to delete the specified number of elements from arraylist starting from the specified index position.

When you execute the above c# program, you will get the result as shown below.



In case, if you want to remove all the elements from arraylist, then use the **Clear()** method.

Same way, we can use different methods like **Sort()** method to sort elements of arraylist and **Reverse()** method to reverse the order of arraylist elements.

C# ArrayList Check If Item Exists

By using **Contains()** method, we can check whether the specified element exists in arraylist or not. In case, if it exists it will return **true** otherwise **false**.

Following is the example of using **Contains()** method to check for an item that exists in arraylist or not in c#.

```
using System;
using System.Collections;
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // creating arraylist
            ArrayList arrlist = new ArrayList();
            arrlist.Add("Welcome");
            arrlist.Add(100);
            arrlist.Add(20.5f);
```

```
            arrlist.Add("Tutlane");
            // check for an item 100 exists in arraylist or not
            Console.WriteLine("Item Exists: " + arrlist.Contains(100));
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Contains()** method to check for an item (**100**) exists in arraylist (**arrlist**) or not.

When you execute the above c# program, you will get the result as shown below.

```
Item Exists: True
```

C# ArrayList Overview

The following are the important points that need to remember about arraylist in c#.

- ArrayList is used to store elements of different data types and the size of arraylist can grow or shrink dynamically by adding or removing elements.
- ArrayList is a non-generic type of collection and available with a **System.Collections** namespace.
- You can access arraylist elements either by using loops (for and foreach) or with a particular index position.
- You need to cast ArrayList items to appropriate data types before using them in your application.
- ArrayList provided different methods to perform multiple operations like add, insert, delete, sort, reverse, etc. on elements of arraylist.

C# Difference between Array and ArrayList

The following are the main difference between array and arraylist in c# programming language.

- The array is strongly typed which means it can store only the elements of the same data type but in ArrayList, you can store elements of different data types.

- The size of the array is fixed whereas the size of ArrayList will vary dynamically based on requirements.
- ArrayList provided different methods to perform operations like add, insert, remove, etc. on multiple elements but in the array, you can get or set the value of only one element at a time.
- In c#, an array can have multiple dimensions but an ArrayList will have only one dimension.
- To use arraylist items, you need to cast it to appropriate data type but in the array, you don’t need to cast elements because it’s a strongly typed.

noteghar.com
noteghar.com

C# Queue with Examples

In c#, **Queue** is useful to represent a collection of objects which stores elements in **FIFO** (First In, First Out) style i.e. the element which added first will come out first. In a queue, elements are inserted from one end and removed from another end.

Generally, queues are useful when you want to access elements from the collection in same order that is stored and you can store multiple null and duplicate values in a queue based on our requirements.

By using **Enqueue()** and **Dequeue()** methods, we can add or delete an element from the queue. Here, the **Enqueue()** method is useful to add elements at the end of the queue and the **Dequeue()** method is useful to remove elements start from the queue.

Following is the pictorial representation of the queue process flow in c# programming language.

C# Queue Declaration

Generally, c# will support both generic and non-generic types of queues. Here, we will learn about non-generic queue collections by using the **System.Collections** namespace.

As discussed, the collection is a class so to define a queue, you just need to declare an instance of the queue class before we perform any operations like add, delete, etc. like as shown below.

```
Queue que = new Queue();
```

If you observe the above queue declaration, we created a new queue (**que**) with an instance of a queue class without specifying any size.

C# Queue Properties

The following are some of the commonly used properties of a queue in c# programming language.

Property	Description
Count	It will return the total number of elements in the queue
IsSynchronized	It is used to get a value to indicate that access to the queue is synchronized (thread-safe) or not.

flyghar.com

C# Queue Methods

The following are some of the commonly used methods of a queue to perform operations like add, delete, etc. on elements of a queue in c# programming language.

Method	Description
Enqueue	It is used to add elements at the end of the queue.
Dequeue	It will remove and returns an item from the starting of a queue.
Clear	It will remove all the elements from the queue.
Clone	It will create a shallow copy of the queue.
Contains	It is used to determine whether an element exists in a queue or not.
Peek	It is used to get the first element from the queue.
TrimToSize	It is used to set the capacity of a queue to an actual number of elements in the queue.

noteghar.com

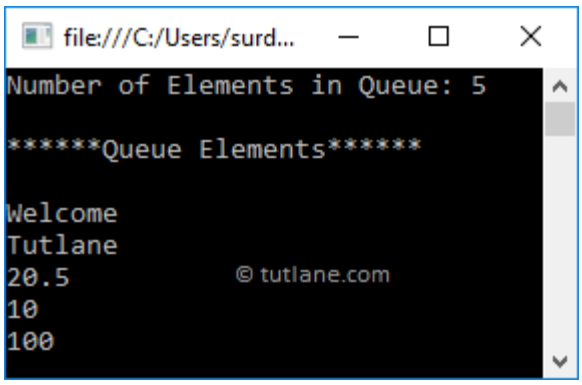
C# Queue Example

Here, we are going to use a non-generic type of queue so we can add elements of different data types to the queue using **Enqueue()** method. Following is the example of adding and accessing elements of the queue in c#.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create and initialize a queue
            Queue que = new Queue();
            que.Enqueue("Welcome");
            que.Enqueue("Tutlane");
            que.Enqueue(20.5f);
            que.Enqueue(10);
            que.Enqueue(100);
            Console.WriteLine("******Queue Example******");
            Console.WriteLine("Number of Elements in Queue: {0}", que.C
ount);
            Console.WriteLine("******Queue Elements******");
            // Access Queue Elements
            foreach (var item in que)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created a new queue (**que**) and added a different data type elements to the queue (**que**)
using **Enqueue()** method and we used a foreach loop to iterate through a queue to get an elements from it.

When you execute the above c# program, you will get the result as shown below.



noteghar.com
tiyghar.com

If you observe the above result, we got elements from the queue in same order which we added first.

C# Dequeue() Method to Access Queue Elements

As discussed, the queue **Dequeue()** method will always remove and return the first element of the queue. Following is the example of accessing a queue elements using **Dequeue()** method in c#.

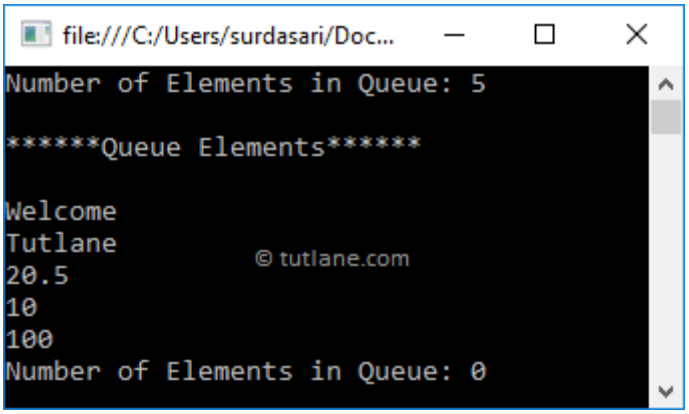
tlyghar.com

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create and initialize a queue
            Queue que = new Queue();
            que.Enqueue("Welcome");
            que.Enqueue("Tutlane");
            que.Enqueue(20.5f);
            que.Enqueue(10);
            que.Enqueue(100);
            Console.WriteLine("Number of Elements in Queue: {0}", que.C
ount);
            Console.WriteLine("******Queue Elements******");
            // Access Queue Elements
            while (que.Count > 0)
            {
                Console.WriteLine(que.Dequeue());
            }
            Console.WriteLine("Number of Elements in Queue: {0}", que.C
ount);
            Console.ReadLine();
        }
    }
}
```

noteghar.com

If you observe above example, we are accessing a queue elements by using **Dequeue()** method and calling a **Dequeue** method on empty queue will throw an exception (**InvalidOperation**) so to avoid that situation we are checking whether the count of queue elements greater than zero or not using while loop.

When you execute the above c# program, we will get the result as shown below.



tiyeghar.com

If you observe the above result, every time the **Dequeue()** method has removed and returned a first element of the queue that's the reason, in the end, we got a count of queue elements as **0**.

In case, if you want to remove all the elements of the queue, then you need to use a **clear()** method.

C# Peek() Method to Access Queue Elements

As discussed, the queue **Peek()** method will always return a first element of the queue. Following is the example of accessing queue elements using the **Peek()** method in c#.

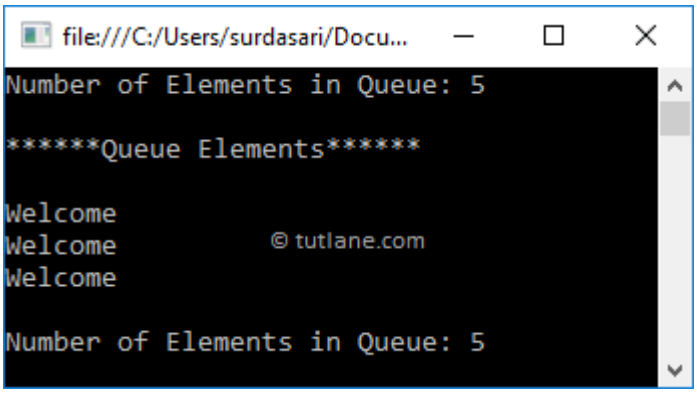
```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create and initialize a queue
            Queue que = new Queue();
            que.Enqueue("Welcome");
            que.Enqueue("Tutlane");
            que.Enqueue(20.5f);
            que.Enqueue(10);
            Console.WriteLine("Number of Elements in Queue: {0}", que.C
ount);
            Console.WriteLine("******Queue Elements******");
            // Access Queue Elements
            Console.WriteLine(que.Peek());
            Console.WriteLine(que.Peek());
            Console.WriteLine(que.Peek());
            Console.WriteLine("Number of Elements in Queue: {0}", que.C
ount);
            Console.ReadLine();
```

noteghar.com

```
        }
    }
}
```

If you observe above example, we are accessing a queue elements by using **Peek()** method.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, every time the **Peek()** method has returned a first element of the queue without removing any element that's the reason at the end also the queue elements count is same as the starting of the queue.

C# Queue Contains() Method

In c#, by using the queue **Contains()** method you can check whether an element exists in a queue or not. In case, if the element found in the queue, then it will return **true** otherwise **false**.

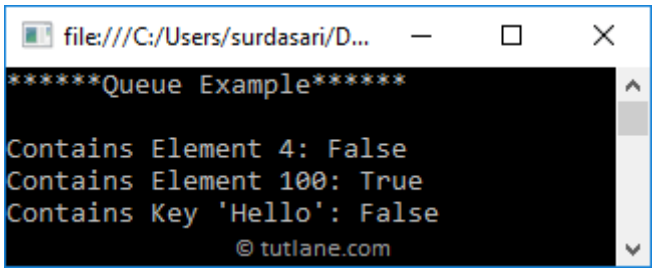
Following is the example of using queue **Contains()** method to check whether an item exists in queue or not in c#.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create and initialize a queue
            Queue que = new Queue();
            que.Enqueue("Welcome");
            que.Enqueue("Tutlane");
            que.Enqueue(20.5f);
```

```
            que.Enqueue(10);
            que.Enqueue(100);
            Console.WriteLine("******Queue Example******");
            Console.WriteLine("Contains Element 4: {0}", que.Contains(4
));
            Console.WriteLine("Contains Element 100: {0}", que.Contains
(100));
            Console.WriteLine("Contains Key 'Hello': {0}", que.Contains
("Hello"));
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Contains()** method to check for a particular elements exists in queue (**que**) or not.

When you execute the above c# program, you will get the result as shown below.



C# Queue Overview

The following are the important points that need to remember about the queue in c#.

- In c#, queues are used to store a collection of objects in a **FIFO** (First in, First out) style i.e. the element which added first will come out first.
- By using **Enqueue()** method, we can add elements at the end of the queue.
- The **Dequeue()** method will remove and return the oldest (first) element from the queue.
- The queue **Peek()** method will always return the first element of the queue and it won't delete any element from the queue.

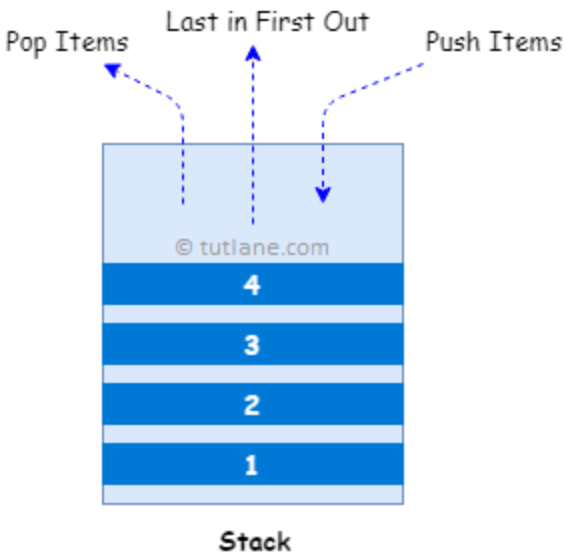
C# Stack with Examples

In c#, **Stack** is useful to represent a collection of objects which store elements in **LIFO** (Last in, First out) style i.e. the element which added last will be the first to come out.

Generally, stacks are useful when you want to access elements from the collection in a last-in-first-out style and we can store multiple null and duplicate values in a stack based on our requirements.

By using **Push()** and **Pop()** / **Peek()** methods, we can add or retrieve an elements from the stack. Here, the **Push()** method is useful to add elements to the stack and the **Pop()** / **Peek()** method is useful to retrieve elements from the stack.

Following is the pictorial representation of the stack process flow in c# programming language.



C# Stack Declaration

Generally, c# will support both generic and non-generic types of stacks. Here, we will learn about non-generic queue collections by using the **System.Collections** namespace so you can add elements of different data types.

As discussed, the collection is a class so to define a stack, you just need to declare an instance of the stack class before we perform any operations like add, delete, etc. like as shown below.

```
Stack stk = new Stack();
```

If you observe the above stack declaration, we created a new stack (**stk**) with an instance of stack class without specifying any size.

C# Stack Properties

The following are some of the commonly used properties of a stack in c# programming language.

Property	Description
Count	It will return the total number of elements in a stack.
IsSynchronized	It is used to get a value to indicate that access to stack is synchronized (thread-safe) or not.

C# Stack Methods

The following are some of the commonly used methods of the stack to perform operations like add, delete, etc. on elements of a stack in c# programming language.

Method	Description
Push	It is used to insert an object at the top of a stack.
Pop	It will remove and return an object at the top of the stack.
Clear	It will remove all the elements from the stack.
Clone	It will create a shallow copy of the stack.
Contains	It is used to determine whether an element exists in a stack or not.
Peek	It is used to return a top element from the stack.

C# Stack Example

Here, we are going to use a non-generic collection of the stack so we can add elements of different data types to the stack using **Push()** method. Following is the example of adding and accessing elements of the stack in c#.

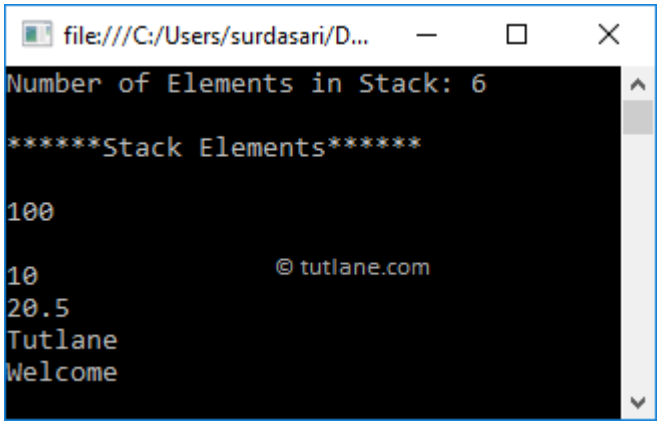
flyghar.com

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create and initialize a stack
            Stack stk = new Stack();
            stk.Push("Welcome");
            stk.Push("Tutlane");
            stk.Push(20.5f);
            stk.Push(10);
            stk.Push(null);
            stk.Push(100);
            Console.WriteLine("Number of Elements in Stack: {0}", stk.C
ount);
            Console.WriteLine("******Stack Elements******");
            // Access Stack Elements
            foreach (var item in stk)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created a new stack (**stk**) and added a different data type elements to the stack (**stk**) using **Push()** method and we used a foreach loop to iterate through the stack to get an elements from it.

noteghar.com

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, we got elements from the stack in **LIFO** (last in, first out) order.

C# Stack Pop() Method to Access Elements

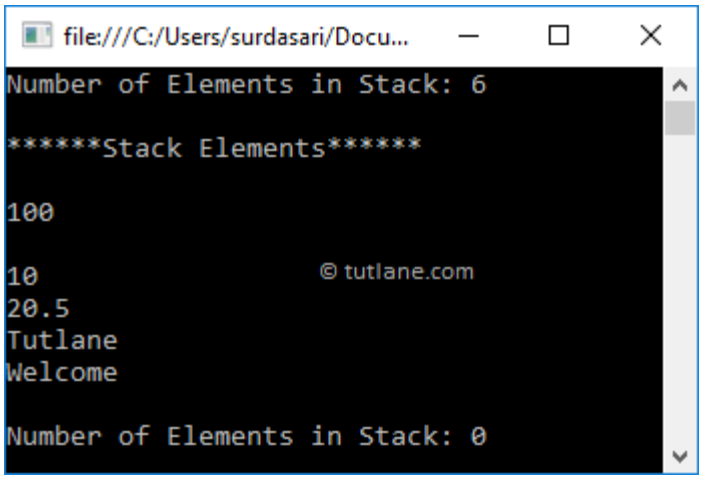
As discussed, the stack **Pop()** method will always remove and return a top element of the queue. Following is the example of accessing stack elements using **Pop()** method in c#.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create and initialize a stack
            Stack stk = new Stack();
            stk.Push("Welcome");
            stk.Push("Tutlane");
            stk.Push(20.5f);
            stk.Push(10);
            stk.Push(null);
            stk.Push(100);
            Console.WriteLine("Number of Elements in Stack: {0}", stk.C
ount);
            Console.WriteLine("******Stack Elements******");
            // Access Stack Elements
            while (stk.Count > 0)
            {
                Console.WriteLine(stk.Pop());
            }
            Console.WriteLine("Number of Elements in Stack: {0}", stk.C
ount);
            Console.ReadLine();
        }
```

```
    }
}
```

If you observe above example, we are accessing a stack elements by using **Pop()** method and calling a **Pop** method on empty stack will throw an exception (**InvalidOperation**) so to avoid that situation we are checking whether the count of stack elements greater than zero or not using while loop.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, every time the **Pop()** method has removed and returned a top element of the stack that's the reason at the end we got a count of stack elements as **0**.

In case, if you want to remove all the elements of the stack, then you need to use a **clear()** method.

C# Stack Peek() Method to Access Elements

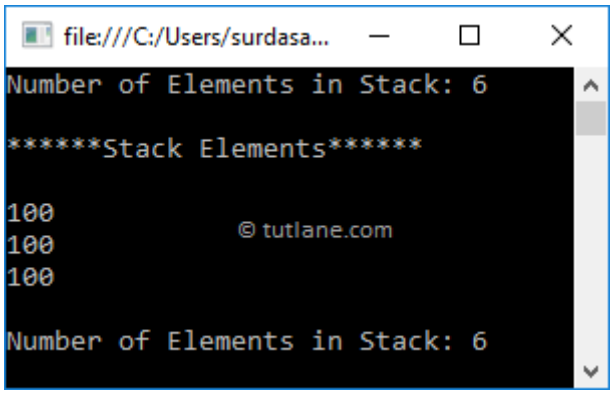
As discussed, the stack **Peek()** method will always return a last (**top-most**) inserted element of the stack. Following is the example of accessing stack elements using the **Peek()** method in c#.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create and initialize a stack
```

```
            Stack stk = new Stack();
            stk.Push("Welcome");
            stk.Push("Tutlane");
            stk.Push(20.5f);
            stk.Push(10);
            stk.Push(null);
            stk.Push(100);
            Console.WriteLine("Number of Elements in Stack: {0}", stk.C
ount);
            Console.WriteLine("******Stack Elements******");
            // Access Stack Elements
            Console.WriteLine(stk.Peek());
            Console.WriteLine(stk.Peek());
            Console.WriteLine(stk.Peek());
            Console.WriteLine("Number of Elements in Stack: {0}", stk.C
ount);
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we are accessing a stack elements by using **Peek()** method.

When you execute the above c# program, we will get the result as shown below.



If you observe the above result, every time the **Peek()** method has returned a last (top-most) element of the queue without removing any element that's the reason at the end also the stack elements count is same as the starting of the stack.

C# Stack Contains() Method

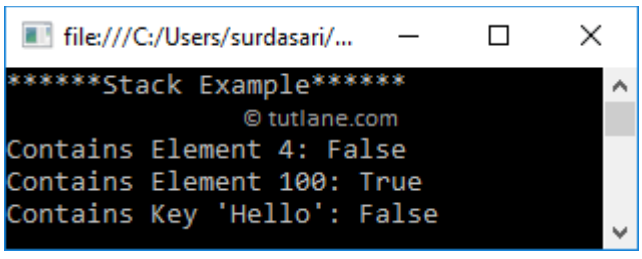
In c#, by using stack **Contains()** method we can check whether an element exists in stack or not. In case, if the element found in stack, then it will return **true** otherwise **false**.

Following is the example of using stack **Contains()** method to check whether an item exists in a stack or not in c#.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create and initialize a stack
            Stack stk = new Stack();
            stk.Push("Welcome");
            stk.Push("Tutlane");
            stk.Push(20.5f);
            stk.Push(10);
            stk.Push(null);
            stk.Push(100);
            Console.WriteLine("******Stack Example******");
            Console.WriteLine("Contains Element 4: {0}", stk.Contains(4
));
            Console.WriteLine("Contains Element 100: {0}", stk.Contains
(100));
            Console.WriteLine("Contains Key 'Hello': {0}", stk.Contains
("Hello"));
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Contains()** method to check for a particular elements exists in stack (**stk**) or not.

When we execute the above c# program, we will get the result as shown below.



noteghar.com
flyghar.com

C# Stack Overview

The following are important points that need to remember about the stack in c#.

- In c#, stacks are used to store a collection of objects in a LIFO (Last in, First out) style i.e. the element which added last will come out first.
- By using the **Push()** method, you can add elements to the stack.
- The **Pop()** method will remove and return a topmost element from the stack.
- The stack **Peek()** method will always return a last (**top-most**) inserted element of the stack and it won't delete any element from the stack.

flyghar.com

noteghar.com

C# HashTable

In c#, **Hashtable** is used to store a collection of key/value pairs of different data types and those are organized based on the hash code of the key.

Generally, the hashtable object will contain buckets to store elements of the collection. Here, the bucket is a virtual subgroup of elements within the hashtable and each bucket is associated with a hash code, which is generated based on the key of an element.

In c#, hashtable is same as a dictionary object but the only difference is the dictionary object is used to store a key-value pair of same data type elements.

When compared with dictionary object, the hashtable will provide a lower performance because the hashtable elements are of object type so the boxing and unboxing process will occur when we store or retrieve values from the hashtable.

C# HashTable Declaration

In c#, hashtable is a non-generic type of collection so we can store a key/value pair elements of different data types and it is provided by **System.Collections** namespace.

As discussed, the collection is a class so to define a hashtable, you just need to declare an instance of the hashtable class before we perform any operations like add, delete, etc. like as shown below.

```
Hashtable htbl = new Hashtable();
```

If you observe the above hashtable declaration, we created a new hashtable (hshtbl) with an instance of hashtable class without specifying any size.

C# HashTable Properties

The following are some of the commonly used properties of hashtable in c# programming language.

Property	Description
Count	It is used to get the number of key/value pair elements in the hashtable.

Property	Description
IsFixedSize	It is used to get a value to indicate that the hashtable has fixed size or not.
IsReadOnly	It is used to get a value to indicate that the hashtable is read-only or not.
Item	It is used to get or set the value associated with the specified key.
IsSynchronized	It is used to get a value to indicate that access to hashtable is synchronized (thread-safe) or not.
Keys	It is used to get the collection of keys in the hashtable.
Values	It is used to get the collection of values in the hashtable.

flyghar.com

C# HashTable Methods

The following are some of the commonly used methods of hashtable to perform operations like add, delete, etc. on elements of hashtable in c# programming language.

Method	Description
Add	It is used to add an element with a specified key and value in a hashtable.
Clear	It will remove all the elements from the hashtable.
Clone	It will create a shallow copy of hashtable.
Contains	It is useful to determine whether the hashtable contains a specific key or not.
ContainsKey	It is useful to determine whether the hashtable contains a specific key or not.

noteghar.com

ContainsValue	It is useful to determine whether the hashtable contains a specific value or not.
Remove	It is useful to remove an element with a specified key from the hashtable.
GetHash	It is useful to get the hash code for the specified key.

flyghar.com

C# Add Elements to HashTable

In c#, while adding elements to hashtable you need to make sure that there will not be any duplicate keys because hashtable will allow us to store duplicate values but keys must be unique to identify the values in the hashtable.

In hashtable, we can add a key/value pair elements of different data types. Following is the example of adding elements to hashtable in c#.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable htbl = new Hashtable();
            htbl.Add("msg", "Welcome");
            htbl.Add("site", "Tutlane");
            htbl.Add(1, 20.5);
            htbl.Add(2, null);
            // Another way to add elements. If key not exist, then that
key adds a new key/value pair.
            htbl[3] = "Tutorials";
            // Add method will throws an exception if key already exist
s in hash table
            try
            {
                htbl.Add(2, 100);
            }
            catch
            {
                Console.WriteLine("An element with Key = '2' already ex
ists.");
            }
```

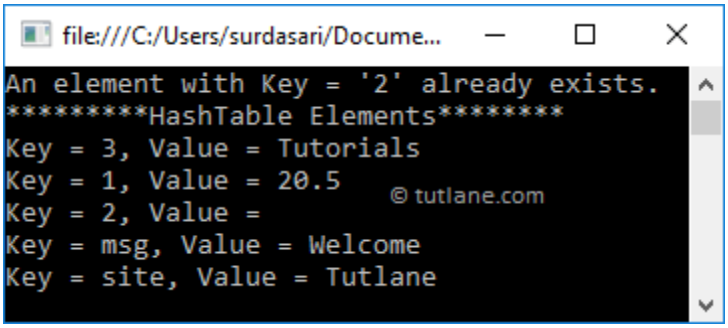
noteghar.com

```
            Console.WriteLine("*********HashTable Elements********");
            // It will return elements as KeyValuePair objects.
            foreach (DictionaryEntry item in htbl)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.ReadLine();
        }
    }
}
```

tiyeghar.com

If you observe above example, we created a new hastable (**htbl**) and adding a different data type elements (key/value) to hashtable (**htbl**) in different ways. As discussed, **Add** method will throw an exception in case if we try to add a key (**2**) which is already existing so to handle that exception we used a try-catch block.

When you execute the above c# program, we will get the result as shown below.



If you observe the above result, we got an exception when we tried to add a key (**2**) which is already existing and added a key (**3**) which is not existing in the hashtable.

In c#, we can also assign a **key/value** pairs to hashtable at the time of initialization like as shown below.

```
Hashtable htbl = new Hashtable(){
                                {"msg", "Welcome"},
                                {"site", "Tutlane"},
                                {1, 20.5},
                                {2, null}
                            };
```

noteghar.com

C# Access HashTable Elements

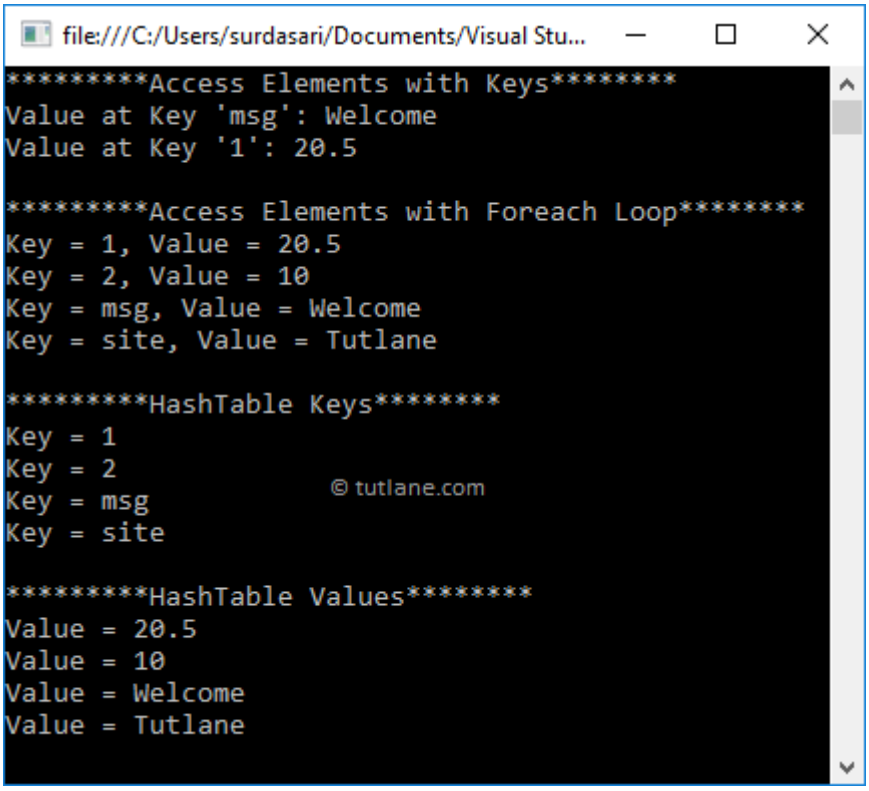
In c#, we have a different ways to access hashtable elements i.e. either by using keys or by iterating through a hashtable using foreach loop.

Following is the example of accessing hashtable elements in different ways.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable htbl = new Hashtable();
            htbl.Add("msg", "Welcome");
            htbl.Add("site", "Tutlane");
            htbl.Add(1, 20.5f);
            htbl.Add(2, 10);
            string msg = (string)htbl["msg"];
            float num = (float)htbl[1];
            Console.WriteLine("*********Access Elements with Keys******
**");
            Console.WriteLine("Value at Key 'msg': " + msg);
            Console.WriteLine("Value at Key '1': " + num);
            Console.WriteLine("*********Access Elements with Foreach Lo
op********");
            foreach (DictionaryEntry item in htbl)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.WriteLine("*********HashTable Keys********");
            foreach (var item in htbl.Keys)
            {
                Console.WriteLine("Key = {0}", item);
            }
            Console.WriteLine("*********HashTable Values********");
            foreach (var item in htbl.Values)
            {
                Console.WriteLine("Value = {0}", item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we are accessing hashtable elements in different ways by using keys, foreach loop and with available properties (**Keys**, **Values**) based on our requirements.

When you execute the above c# program, we will get the result as shown below.



C# Remove Elements from HashTable

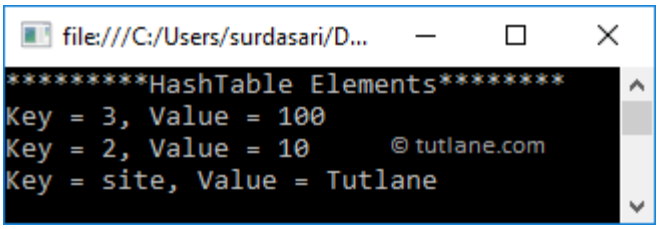
In c#, you can delete elements from hashtable by using the **Remove()** method. Following is the example of deleting elements from hashtable in c# programming language.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable htbl = new Hashtable();
            htbl.Add("msg", "Welcome");
            htbl.Add("site", "Tutlane");
            htbl.Add(1, 20.5f);
            htbl.Add(2, 10);
            htbl.Add(3, 100);
            // Removing hashtable elements with keys
            htbl.Remove(1);
            htbl.Remove("msg");
```

```
            Console.WriteLine("*********HashTable Elements********");
            foreach (DictionaryEntry item in htbl)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Remove()** method to delete particular value of element from hashtable using keys.

When you execute the above c# program, we will get the result as shown below.



In case, if you want to remove all the elements from a hashtable, then use the Clear() method.

C# HashTable Check If Element Exists

By using **Contains()**, **ContainsKey()** and **ContainsValue()** methods, we can check whether the specified element exists in hashtable or not. In case, if it exists these methods will return **true** otherwise **false**.

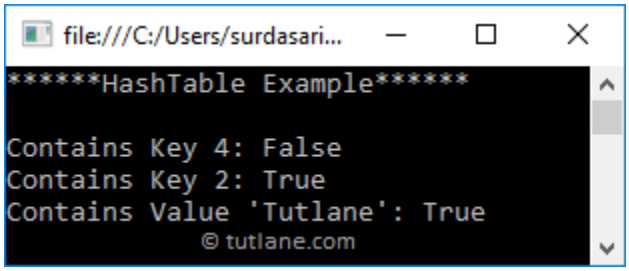
Following is the example of using **Contains()**, **ContainsKey()** and **ContainsValue()** methods to check for an item that exists in a hashtable or not in c#.

```
using System;
using System.Collections;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable htbl = new Hashtable();
            htbl.Add("msg", "Welcome");
            htbl.Add("site", "Tutlane");
```

```
            htbl.Add(1, 20.5f);
            htbl.Add(2, 10);
            htbl.Add(3, 100);
            Console.WriteLine("Contains Key 4: {0}", htbl.Contains(4));
            Console.WriteLine("Contains Key 2: {0}", htbl.ContainsKey(2
));
            Console.WriteLine("Contains Value 'Tutlane': {0}", htbl.Con
tainsValue("Tutlane"));
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Contains()**, **ContainsKey()** and **ContainsValue()** methods to check for a particular keys and values exists in hashtable (htbl) or not.

When you execute the above c# program, we will get the result as shown below.



C# HashTable Overview

The following are the important points that need to remember about hashtable in c#.

- Hashtable is used to store a collection of key/value pairs of different data types and those are organized based on the hash code of the key.
- Hashtable is a non-generic type of collection so we need to cast hashtable items to appropriate data type before we use it in our application.
- Hashtable will allow us to store duplicate values but keys must be unique to identify the values in the hashtable.
- In hashtable, the key cannot be null but the value can be null.

- You can access hashtable elements either by using keys or with a foreach loop. In the foreach loop, you need to use **DictionaryEntry** to get a key/value pairs from the hashtable.

flyghar.com

noteghar.com

What is a constructor in C#?

A special method of the class that is automatically invoked when an instance of the class is created is called a constructor. The main use of constructors is to initialize the private fields of the class while creating an instance for the class. When you have not created a constructor in the class, the compiler will automatically create a default constructor of the class. The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.

flyghar.com

Some of the key points regarding constructor are:

- A class can have any number of constructors.
- A constructor doesn't have any return type, not even void.
- A static constructor cannot be a parametrized constructor.
- Within a class, you can create one static constructor only.

In C#, constructors can be divided into 5 types:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

1.Default Constructor in C#

A constructor without any parameters is called a default constructor; in other words, this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class with different values. The default constructor initializes:

noteghar.com

1. All numeric fields in the class to zero.
2. All string and object fields to null.

Example

```
1. using System;
2. namespace DefaultConstractor
3. {
4.     class addition
```

```
5.     {
6.         int a, b;
7.         public addition()   //default contructor
8.         {
9.             a = 100;
10.                 b = 175;
11.             }
12.
13.             public static void Main()
14.             {
15.                 addition obj = new addition(); //an object is
    created , constructor is called
16.                 Console.WriteLine(obj.a);
17.                 Console.WriteLine(obj.b);
18.                 Console.Read();
19.             }
20.         }
21.     }
```

Now run the application, the output will be as following:

noteghar.com
flyghar.com

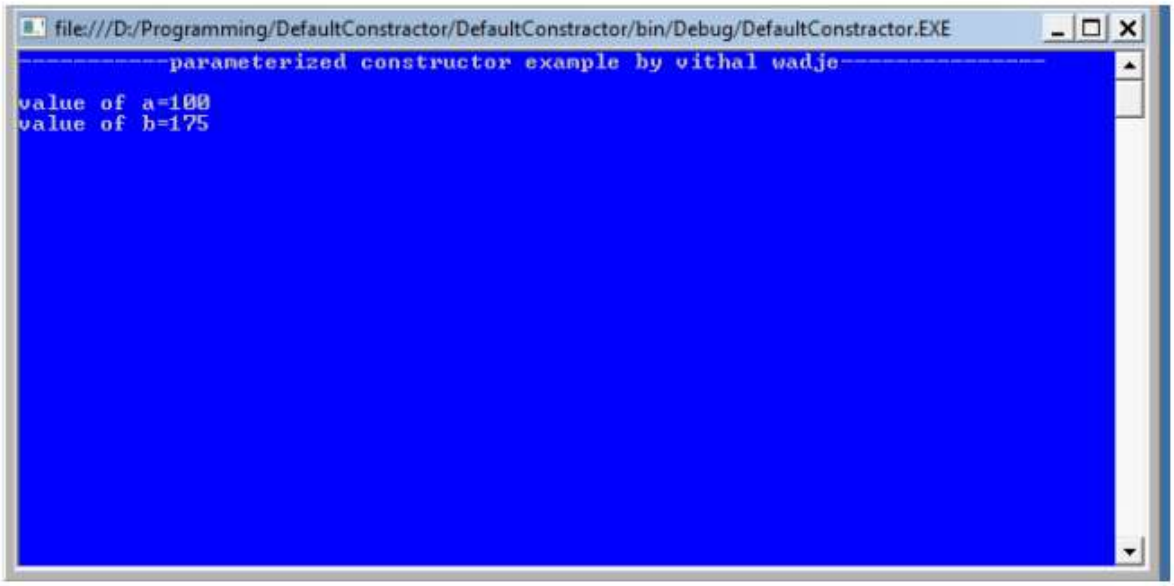
2.Parameterized Constructor in C#

A constructor with at least one parameter is called a parameterized constructor. The advantage of a parameterized constructor is that you can initialize each instance of the class with a different value.

Example

```
1. using System;
2. namespace Constructor
3. {
4.     class paraconstrctor
5.     {
6.       public  int a, b;
7.       public paraconstrctor(int x, int y)  // decalaring Parem
   etrized Constructor with ing x,y parameter
8.         {
9.             a = x;
10.                 b = y;
11.             }
12.         }
13.          class MainClass
14.          {
15.              static void Main()
16.              {
17.                  paraconstrctor v = new paraconstrctor(100, 17
   5);   // Creating object of Parameterized Constructor and ing
   values
18.                  Console.WriteLine("-----------
   parameterized constructor example---------------");
19.                  Console.WriteLine("\t");
20.                  Console.WriteLine("value of a=" + v.a );
21.                  Console.WriteLine("value of b=" + v.b);
22.                  Console.Read();
23.              }
24.          }
25.     }
```

Now run the application, the output will be as following:



noteghar.com
flyghar.com

3.Copy Constructor in C#

The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

Syntax

```
public employee(employee emp)
{
    name=emp.name;
    age=emp.age;
}
```

The copy constructor is invoked by instantiating an object of type employee and bypassing it the object to be copied.

Example

```
employee emp1=new  employee (emp2);
```

Now, emp1 is a copy of emp2.

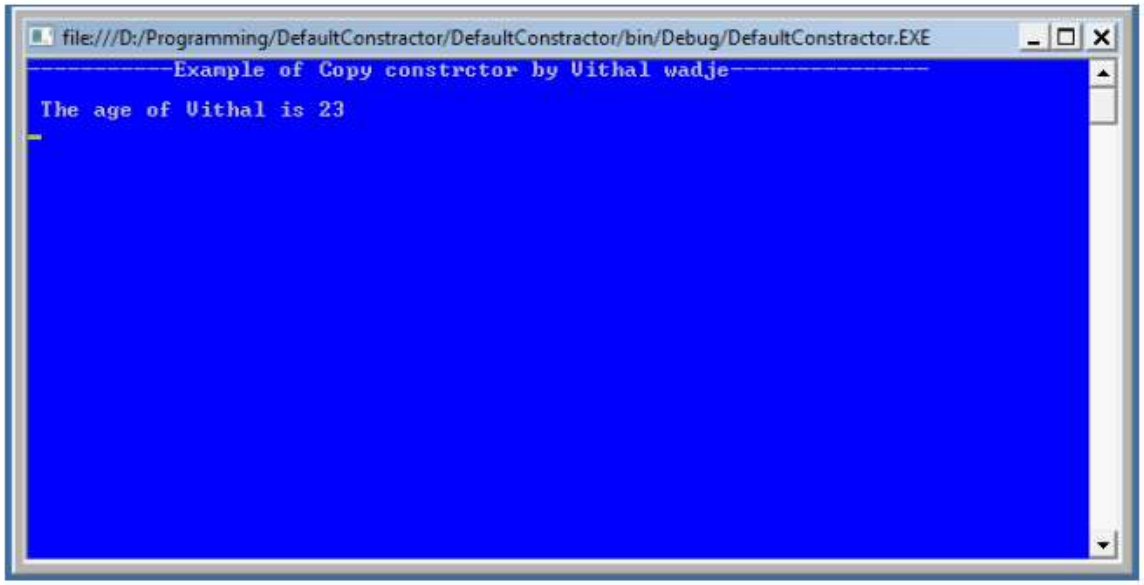
Let's see its practical implementation.

```
using System;
namespace copyConstractor
{
    class employee
    {
        private string name;
        private int age;
        public employee(employee emp)   // declaring Copy constructor.
        {
                name = emp.name;
                age = emp.age;
            }
            public employee(string name, int age)  // Instance constructor.
            {
                this.name = name;
                this.age = age;
            }
            public string Details     // Get deatils of employee
            {
                get
                {
```

noteghar.com
flyghar.com

```
22.                     return  " The age of " + name +" is "+ ag
   e.ToString();
23.                 }
24.             }
25.         }
26.         class empdetail
27.         {
28.             static void Main()
29.             {
30.                 employee emp1 = new employee("Vithal", 23);
   // Create a new employee object.
31.                 employee emp2 = new employee(emp1);
   // here is emp1 details is copied to emp2.
32.                 Console.WriteLine(emp2.Details);
33.                 Console.ReadLine();
34.             }
35.         }
36.     }
```

Now run the program, the output will be as follows:



noteghar.com

4.Static Constructor in C#

When a constructor is created using a static keyword, it will be invoked only once for all of the instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Some key points of a static constructor are:

1. A static constructor does not take access modifiers or have parameters.
2. A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
3. A static constructor cannot be called directly.
4. The user has no control over when the static constructor is executed in the program.
5. A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

Syntax

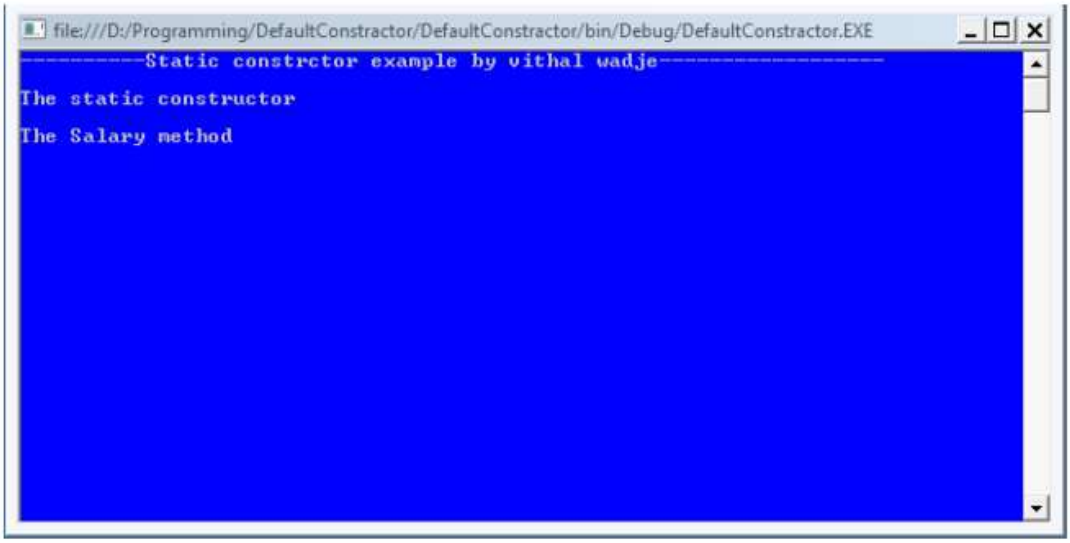
```
class employee
 {// Static constructor
  static employee(){}
 }
```

Now let's see it practically.

```
using System;
namespace staticConstractor
{
    public class employee
    {
        static employee() // Static constructor
        declaration{Console.WriteLine("The static constructor
");
    }
    public static void Salary()
         {
             Console.WriteLine();
             Console.WriteLine("The Salary method");
         }
     }
     class details
     {
         static void Main()
         {
             Console.WriteLine("----------
Static constrctor example  ------------------");
             Console.WriteLine();
             employee.Salary();
             Console.ReadLine();
         }
       }
    }
```

noteghar.com

Now run the program the output will be look as in the following:



flyghar.com

5.Private Constructor in C#

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class. They are usually used in classes that contain static members only. Some key points of a private constructor are:

1. One use of a private constructor is when we have only static members.
2. It provides an implementation of a singleton class pattern
3. Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.

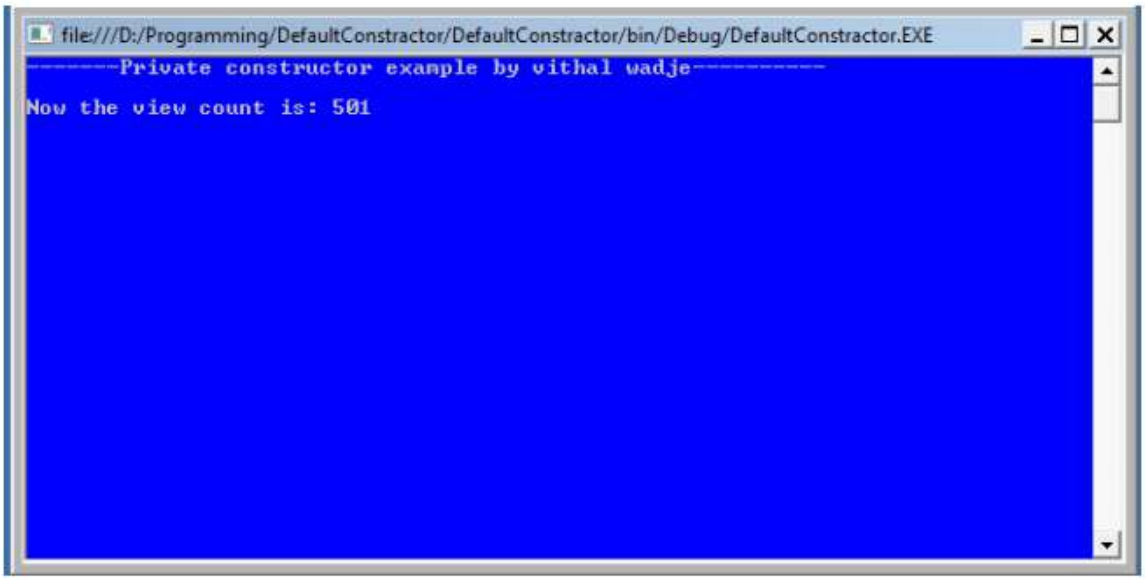
Now let's see it practically.

```
1. using System;
2. namespace defaultConstractor
3. {
4.     public class Counter
5.     {
6.         private Counter()   //private constrctor declaration
7.         {
8.         }
9.         public static int currentview;
10.              public static int visitedCount()
11.              {
12.                  return ++ currentview;
13.              }
14.          }
```

noteghar.com

```
15.         class viewCountedetails
16.         {
17.             static void Main()
18.             {
19.                 // Counter aCounter = new Counter(); // Error

20.                 Console.WriteLine("-------
   Private constructor example ----------");
21.                 Console.WriteLine();
22.                 Counter.currentview = 500;
23.                 Counter.visitedCount();
24.                 Console.WriteLine("Now the view count is: {0}
   ", Counter.currentview);
25.                 Console.ReadLine();
26.             }
27.         }
28.     }
```

Now run the application. The output is as follows.



If you uncomment the preceding statement that is commented in the above program then it will generate an error because the constructor is inaccessible (private).

Properties of Access Modifiers

Access modifiers defines the access level of a property whether a property can be accessed by any caller program, within an assembly, or just within a class.

The following table describes access level modifiers.

- **public** - The type or member can be accessed by any other code in the same assembly or another assembly that references it.
- **Private** - The type or member can be accessed only by code in the same class or struct.
- **protected** - The type or member can be accessed only by code in the same class, or in a class that is derived from that class.
- **internal** - The type or member can be accessed by any code in the same assembly, but not from another assembly.
- **protected internal** - The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.
- **private protected** - The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

C# Properties (GET, SET)

In c#, **Property** is an extension of the class variable and it provides a mechanism to read, write or change the value of the class variable without affecting the external way of accessing it in our applications.

In c#, properties can contain one or two code blocks called **accessors** and those are called a get accessor and set accessor. By using get and set accessors, we can change the internal implementation of class variables and expose it without effecting the external way of accessing it based on our requirements.

Generally, in object-oriented programming languages like c# you need to define fields as **private**, and then use properties to access their values in a **public** way with get and set accessors.

Following is the syntax of defining a **property** with get and set accessor in c# programming language.

noteghar.com
flyghar.com

```
<access_modifier> <return_type> <property_name>
{
    get
    {
        // return property value
    }
    set
    {
        // set a new value
    }
}
```

If you observe the above syntax, we used an access modifier and **return type** to define a property along with get and set accessors to make required modifications to the class variables based on our requirements.

Here, the get accessor code block will be executed whenever the property is read and the code block of set accessor will be executed whenever the property is assigned to a new value.

In c#, the properties are categorized as three types, those are.

Type	Description
Read-Write	A property which contains a both `get` and `set` accessors, then we will call it as read-write property.
Read-Only	A property which contains only `get` accessor, then we will call it as a read-only property.
Write-Only	A property which contains only `set` accessor, then we will call it as write-only property.

In c#, Properties won't accept any parameters and we should not pass a property as a ref or out parameter in our application.

Following is the simple example of defining a private variable and a property in c# programming language.

```
class User
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

If you observe the above example, we defined a property called "**Name**" and we used a get accessor to return a property value and set accessors

to set a new value. Here, the **value** keyword in set accessor is used to define a value that is being assigned by set accessor.

In c#, the get accessor needs to be used only to return the field value or to compute it and return it but we should not use it for changing the state of an object.

As discussed, we can extend the behavior of class variables using properties get and set accessors. Following is the example of extending the behavior of **private** variable in property using get and set accessors in c# programming language.

flyghar.com

```
class User
{
    private string name = "Suresh Dasari";
    public string Name
    {
        get
        {
            return name.ToUpper();
        }
        set
        {
            if (value == "Suresh")
                name = value;
        }
    }
}
```

If you observe the above example, we are extending the behavior of **private** variable **name** using a property called **Name** with get and set accessors by performing some validations like to make sure **Name** value equals to only "**Suresh**" using set accessor and converting property text to uppercase with get accessor.

Here the field "**name**" is marked as **private** so if you want to make any changes to this field then we can do it only by calling the property (**Name**).

noteghar.com

In c# properties, the get accessor will be invoked while reading the value of a property and when we assign a new value to the property, then the set accessor will be invoked by using an argument that provides the new value.

Following is the example of invoking get and set accessors of properties in c# programming language.

```
User u = new User();
u.Name = "Rohini"; // set accessor will invoke
Console.WriteLine(u.Name); // get accessor will invoke
```

In the above example, when we assign a new value to the property, then the set accessor will be invoked and the get accessor will be invoked when we try to read the value from property.

tiyeghar.com

C# Create Read and Write Property

As discussed, if a property contains the get and set accessor, then we will call it as Read and Write property. Following is the example of creating Read and Write property in c# programming language.

Following is the exampleof defining **properties** with get and set accessors to implement required validations without effecting the external way of using it in c# programming language.

```
using System;

namespace Tutlane
{
    class User
    {
        private string location;
        private string name = "Suresh Dasari";
        public string Location
        {
            get { return location; }
            set { location = value; }
        }
        public string Name
        {
            get
            {
                return name.ToUpper();
            }
            set
            {
                if (value == "Suresh")
                    name = value;
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
```

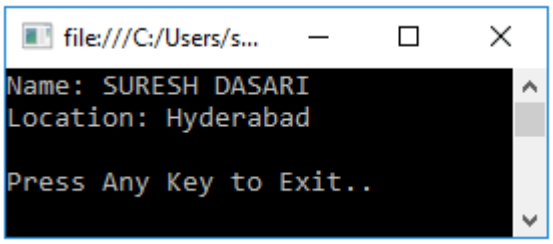
noteghar.com

```
        {
            User u = new User();
            // set accessor will invoke
            u.Name = "Rohini";
            // set accessor will invoke
            u.Location = "Hyderabad";
            // get accessor will invoke
            Console.WriteLine("Name: " + u.Name);
            // get accessor will invoke
            Console.WriteLine("Location: " + u.Location);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

tiyeghar.com

If you observe above example, we are extending the behavior of private variables (**name**, **location**) using properties (**Name**, **Location**) with get and set accessors by performing some validations like to make sure **Name** value is equals to only "**Suresh**" using set accessor and converting property text to uppercase with get accessor.

When you execute the above c# program, we will get the result as shown below.



If you observe the above example, our variable text converted to upper case and even after we set variable text as "**Rohini**", it displayed text as "**Suresh Dasari**" because of the set accessor validation fails in the property.

noteghar.com

C# Create Read-Only Property

As discussed, if a property contains the only get accessor, then we will call it as a read-only property. Following is the example of creating read-only properties in c# programming language.

```
using System;

namespace Tutlane
{
    class User
    {
        private string name;
        private string location;
        public User(string a, string b)
        {
            name = a;
            location = b;
        }
        public string Name
        {
            get
            {
                return name;
            }
        }
        public string Location
        {
            get
            {
                return location;
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User("Suresh Dasari", "Hyderabad");
            // compile error
            // u.Name = "Rohini";
            // get accessor will invoke
            Console.WriteLine("Name: " + u.Name);
            // get accessor will invoke
            Console.WriteLine("Location: " + u.Location);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

tiyeghar.com

noteghar.com

```
using System;
```

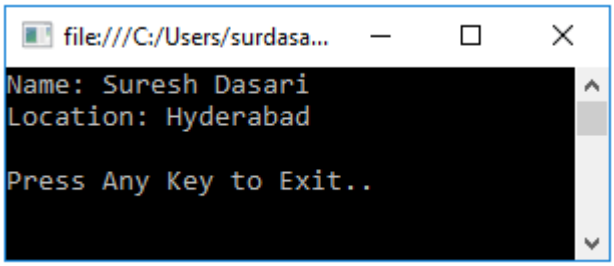
```
namespace Tutlane
{
    class User
    {
        private string name;
        private string location;
        public User(string a, string b)
        {
            name = a;
            location = b;
        }
        public string Name
        {
            get
            {
                return name;
            }
        }
        public string Location
        {
            get
            {
                return location;
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User("Suresh Dasari", "Hyderabad");
            // compile error
            // u.Name = "Rohini";
            // get accessor will invoke
            Console.WriteLine("Name: " + u.Name);
            // get accessor will invoke
            Console.WriteLine("Location: " + u.Location);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

noteghar.com
tlyghar.com

If you observe above example, we created a properties using only get accessor to make the properties are read-only based on our requirements.

In case, if we uncomment the commented code then we will get a compile error because our **Name** property doesn't contain any set accessor to set a new value. It's a just read-only property.

When you execute the above c# program, you will get a result like as shown below.



This is how we can create read-only properties in c# applications based on our requirements.

C# Create Write Only Property

As discussed, if a property contains the only set accessor, then we will call it as **write-only** property. Following is the example of creating write-only properties in c# programming language.

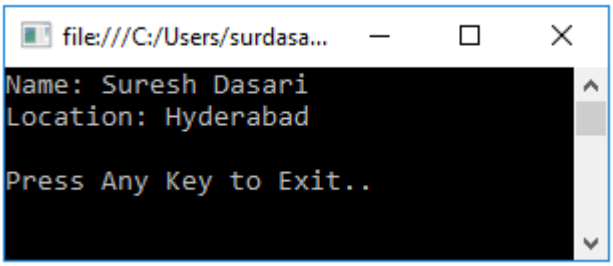
```
using System;

namespace Tutlane
{
    class User
    {
        private string name;
        public string Name
        {
            set
            {
                name = value;
            }
        }
        private string location;
        public string Location
        {
            set
            {
                location = value;
            }
        }
        public void GetUserDetails()
        {
```

```
            Console.WriteLine("Name: " + name);
            Console.WriteLine("Location: " + location);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User();
            u.Name = "Suresh Dasari";
            u.Location = "Hyderabad";
            // compile error
            //Console.WriteLine(u.Name);
            u.GetUserDetails();
            Console.WriteLine("\nPress Enter Key to Exit...");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created a properties using only set accessor to make the properties are **write-only** based on our requirements.

In case, if we uncomment the commented code then we will get a compile error because our **Name** property doesn't contain any get accessor to return a value. It's a just write-only property.

When you execute the above c# program, you will get a result like as shown below.



This is how we can create **write-only** properties in c# applications based on our requirements.

C# Auto Implemented Properties

In c#, a property is called as an auto-implemented property when it contains accessors (**get**, **set**) without having any logic implementation.

Generally, the auto-implemented properties are useful whenever there is no logic implementation required in property accessors.

Following is the example of creating auto-implemented properties in c# programming language.

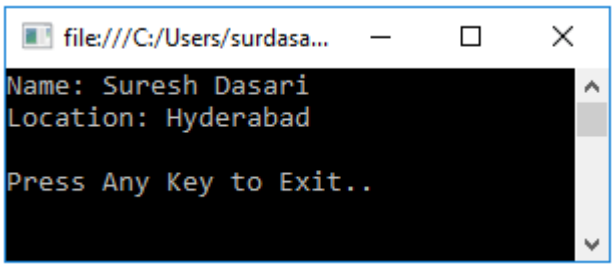
flyghar.com

```
using System;

namespace Tutlane
{
    class User
    {
        public string Name { get; set; }
        public string Location { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User();
            u.Name = "Suresh Dasari";
            u.Location = "Hyderabad";
            Console.WriteLine("Name: " + u.Name);
            Console.WriteLine("Location: " + u.Location);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created a properties with `get` and `set` accessors without having any logic implementation.

When you execute the above c# program, we will get a result like as shown below.

noteghar.com



This is how we can create **auto-implemented** properties in c# applications based on our requirements.

C# Properties Overview

The following are the important points which we need to remember about properties in c# programming language.

- In c#, properties will enable class variables to expose in public way using get and set accessors by hiding implementation details.
- In properties, a get accessor is used to return a property value and a set accessor is used to assign a new value.
- The **value** keyword in set accessor is used to define a value which is going to be assigned by the set accessor.
- In c#, the properties are categorized as read-write, read-only or write-only.

noteghar.com

Static Properties

C# also supports static properties, which belongs to the class rather than to the objects of the class. All the rules applicable to a static member are applicable to static properties also.

The following program shows a class with a static property.

```
1. //C# : static Property
2. //Author: rajeshvs@msn.com
3. using System;
4. class MyClass
5. {
6.     private static int x;
7.     public static int X
8.     {
9.         get
10.             {
11.                 return x;
12.             }
13.             set
14.             {
15.                 x = value;
16.             }
17.         }
18.     }
19.     class MyClient
```

```
    {
        public static void Main()
        {
            MyClass.X = 10;
            int xVal = MyClass.X;
            Console.WriteLine(xVal);//Displays 10
        }
    }
```

Remember that set/get accessor of static property can access only other static members of the class. Also , static properties are invoking by using the class name.

Properties & Inheritance

The properties of a Base class can be inherited to a Derived class.

```
//C# : Property : Inheritance
//Author: rajeshvs@msn.com
using System;
class Base
{
    public int X
    {
        get
        {
                Console.Write("Base GET");
                return 10;
            }
            set
            {
                Console.Write("Base SET");
            }
        }
    }
    class Derived : Base
    {
    }
    class MyClient
    {
        public static void Main()
        {
            Derived d1 = new Derived();
            d1.X = 10;
            Console.WriteLine(d1.X);
        }
    }
```

noteghar.com

The output from above listing is shown below.

The above program is very straightforward. The inheritance of properties is just like inheritance any other member.

Properties & Polymorphism

A Base class property can be polymorphically overridden in a Derived class. But remember that the modifiers like virtual, override etc are using at property level, not at accessor level.

```
1. //C# : Property : Polymorphism
2. //Author: rajeshvs@msn.com
3. using System;
4. class Base
5. {
6.     public virtual int X
7.     {
8.         get
9.         {
10.                 Console.Write("Base GET");
```

```
                return 10;
            }
            set
            {
                Console.Write("Base SET");
            }
        }
    }
    class Derived : Base
    {
        public override int X
        {
            get
            {
                Console.Write("Derived GET");
                return 10;
            }
            set
            {
                Console.Write("Derived SET");
            }
        }
    }
    class MyClient
    {
        public static void Main()
        {
            Base b1 = new Derived();
            b1.X = 10;
            Console.WriteLine(b1.X);
        }
    }
```

The output from above listing is shown below.

noteghar.com
noteghar.com

Abstract Properties

A property inside a class can be declared as abstract by using the keyword abstract. Remember that an abstract property in a class carries no code at all. The get/set accessors are simply represented with a semicolon. In the derived class we must implement both set and get assessors.

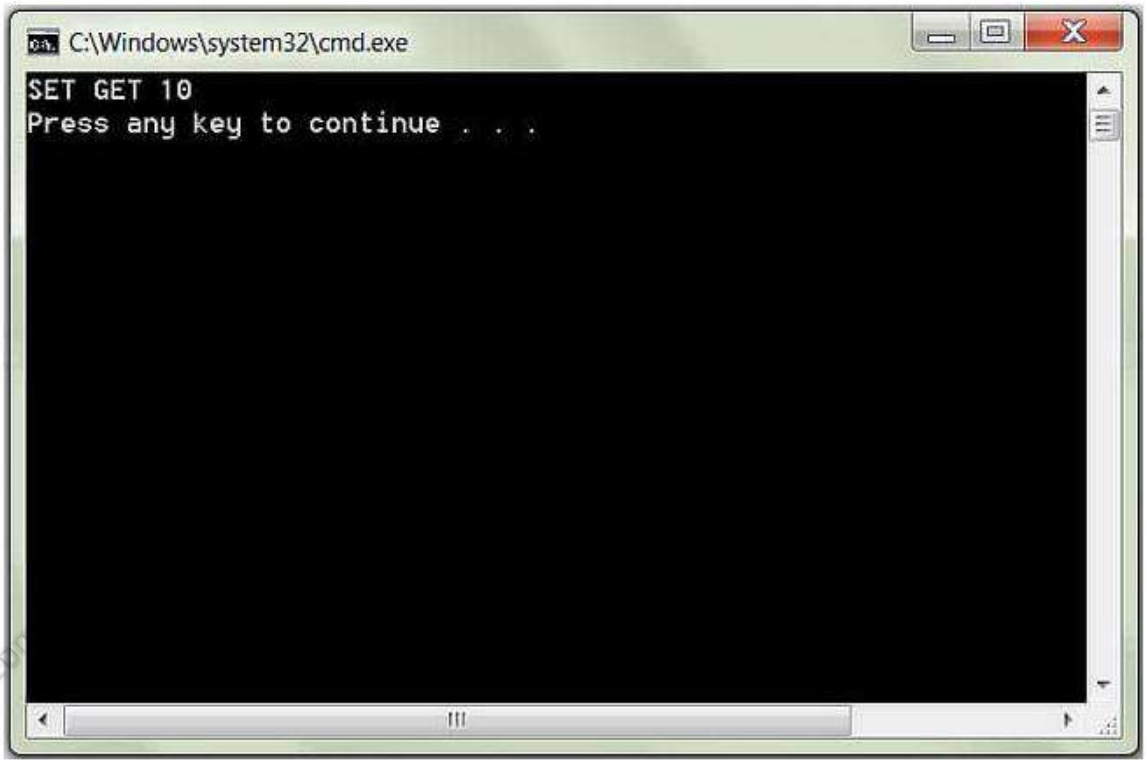
If the abstract class contains only set accessor, we can implement only set in the derived class.

The following program shows an abstract property in action.

```
1. //C# : Property : Abstract
2. //Author: rajeshvs@msn.com
3. using System;
4. abstract class Abstract
5. {
6.     public abstract int X
7.     {
8.         get;
9.         set;
10.          }
11.     }
12.     class Concrete : Abstract
13.     {
14.         public override int X
15.         {
16.             get
```

```
17.             {
18.                 Console.Write(" GET");
19.                 return 10;
20.             }
21.             set
22.             {
23.                 Console.Write(" SET");
24.             }
25.         }
26.     }
27.     class MyClient
28.     {
29.         public static void Main()
30.         {
31.             Concrete c1 = new Concrete();
32.             c1.X = 10;
33.             Console.WriteLine(c1.X);
34.         }
35.     }
```

The output from above listing is shown below.



The properties are important features added in language level inside C#. They are very useful in GUI programming. Remember that the compiler actually generates the appropriate getter and setter methods when it parses the C# property syntax.

flyghar.com
noteghar.com

C# Constructors with Examples

In c#, **Constructor** is a method that will invoke automatically whenever an instance of class or **struct** is created. The constructor will have the same name as the class or **struct,** and it useful to initialize and set default values for the data members of the new object.

tiyghar.com

If we create a class without having any constructor, then the compiler will automatically create a one default constructor for that class. So, there is always one constructor that will exist in every class.

In c#, a class can contain more than one constructor with different types of arguments. The constructors will never return anything, so we don't need to use any return type, not even **void,** while defining the constructor method in the class.

C# Constructor Syntax

As discussed, the constructor is a method, and it won't contain any return type. If you want to create a constructor in c#, then you need to create a method with the class name.

Following is the syntax of creating a constructor in the c# programming language.

```
public class User
{
// Constructor
public User()
{
// Your Custom Code
}
}
```

noteghar.com

If you observe the above syntax, we created a class called "User" and a method whose name is same as the class name. Here the method User() will become a constructor of our class.

C# Constructor Types

In c#, we have different types of constructors available; those are

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Static Constructor
- Private Constructor

Now we will learn about each constructor in a detailed manner with examples in the c# programming language.

C# Default Constructor

In c#, if we create a constructor without having any parameters, then we will call it a **default constructor,** and every instance of the class will be initialized without any parameter values.

Following is the example of defining the default constructor in the c# programming language.

```
using System;

namespace Tutlane
{
    class User
    {
        public string name, location;
        // Default Constructor
        public User()
        {
            name = "Suresh Dasari";
            location = "Hyderabad";
        }
    }
    class Program
```

```
    {
        static void Main(string[] args)
        {
            // The constructor will be called automatically once the instanc
e of the class created
            User user = new User();
            Console.WriteLine(user.name);
            Console.WriteLine(user.location);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we created a class called "**User**" and the constructor method "**User()**" without having any parameters. When we create an instance of our class (**User**), automatically our constructor method will be called.

When you execute the above c# program, you will get the result as shown below.

If you observe the above result, our constructor method has called automatically and initialized the parameter values after creating an instance of our class.

C# Parameterized Constructor

In c#, if we create a constructor with at least one parameter, then we will call it a **parameterized constructor,** and every instance of the class will be initialized with parameter values.

Following is the example of defining the parameterized constructor in the c# programming language.

```
using System;

namespace Tutlane
{
   class User
   {
      public string name, location;
      // Parameterized Constructor
      public User(string a, string b)
      {
        name = a;
        location = b;
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         // The constructor will be called automatically once the instanc
e of the class created
         User user = new User("Suresh Dasari", "Hyderabad");
         Console.WriteLine(user.name);
         Console.WriteLine(user.location);
         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

If you observe the above example, we created a class called "**User**" and the constructor method "**User(string, string)**" with parameters. When we create an instance of our class (**User**) with the required parameters, automatically our constructor method will be called.

When you execute the above c# program, you will get the result as shown below.

tiyghar.com

If you observe the above result, our constructor method has called automatically and initialized the parameter values after creating an instance of our class with the required parameters.

In the next chapters, you will learn about copy constructor, static constructor, and private constructor in c# programming language with examples.

C# Constructor Overloading

In c#, we can **overload** the constructor by creating another constructor with the same method name but with different parameters.

Following is the example of implementing a constructor overloading in the c# programming language.

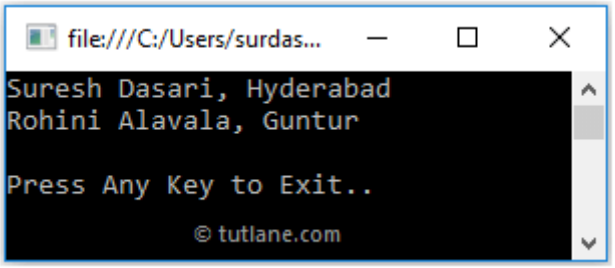
```
using System;

namespace Tutlane
{
   class User
   {
      public string name, location;
      // Default Constructor
      public User() {
          name = "Suresh Dasari";
          location = "Hyderabad";
      }
```

noteghar.com

```
        // Parameterized Constructor
        public User(string a, string b)
        {
            name = a;
            location = b;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User user = new User(); // Default Constructor will be called
            User user1 = new User("Rohini Alavala", "Guntur"); // Parame
terized Constructor will be called
            Console.WriteLine(user.name + ", " + user.location);
            Console.WriteLine(user1.name + ", " + user1.location);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we created a class called "**User**" and overloaded a constructor "**User()**" by creating another constructor "**User(string, string)**" with the same name but with different parameters.

When you execute the above c# program, you will get the result as shown below.



noteghar.com
flyghar.com

If you observe the above result, the respective constructor methods will be called automatically when we create an instance of our class with or without parameters based on our requirements.

C# Constructor Chaining

In c#, **Constructor Chaining** is an approach to invoke one constructor from another constructor. To achieve constructor chaining, we need to use this keyword after our constructor definition.

Following is the example of implementing a **constructor chaining** in c# programming language.

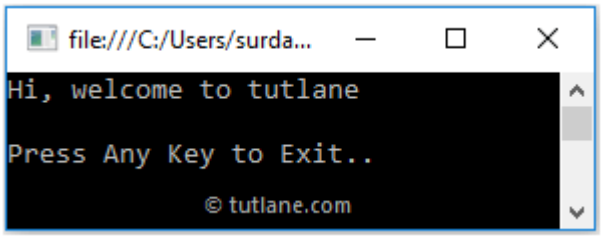
```
using System;

namespace Tutlane
{
    class User
    {
        public User()
        {
            Console.Write("Hi, ");
        }
        public User(string a): this()
        {
            Console.Write(a);
        }
        public User(string a, string b): this("welcome")
        {
            Console.Write(a + " " + b);
        }
    }
    class Program
    {
      static void Main(string[] args)
      {
          User user1 = new User(" to", "tutlane");
          Console.WriteLine();
```

noteghar.com
flyghar.com

```
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we created different constructors with different parameters, and we are calling one constructor from another constructor using this keyword.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, we are able to call one constructor from another constructor to achieve constructor chaining in the c# programming language.

This is how we can achieve constructor chaining in our applications using the c# programming language.

C# Copy Constructor with Examples

In c#, **Copy Constructor** is a parameterized constructor that contains a parameter of the same class type. The copy constructor in c# is useful whenever we want to initialize a new instance to the values of an existing instance.

In simple words, we can say a copy constructor is a constructor that copies one object's data into another object. Generally, c# won't

provide a copy constructor for objects, but we can implement ourselves based on our requirements.

C# Copy Constructor Syntax

Following is the syntax of defining a copy constructor in the c# programming language.

flyghar.com

```
class User
{
    // Parameterized Constructor
    public User(string a, string b)
    {
        // your code
    }
    // Copy Constructor
    public User(User user) {
       // your code
    }
}
```

If you observe the above syntax, we created a copy constructor with a parameter of the same class type, and it helps us initialize a new instance to the values of an existing instance.

C# Copy Constructor Example

Following is the example of creating a copy constructor to initialize a new instance to the values of an existing instance in the c# programming language.

noteghar.com

```
using System;

namespace Tutlane
{
    class User
    {
```

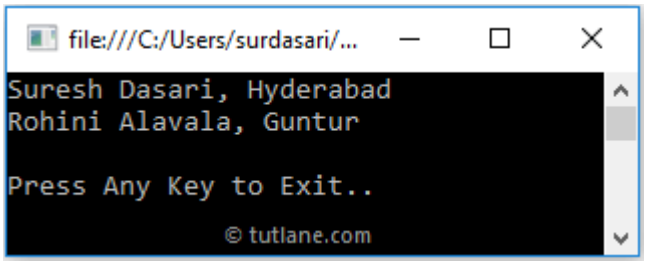
```
    public string name, location;
    // Parameterized Constructor
    public User(string a, string b)
    {
        name = a;
        location = b;
    }
    // Copy Constructor
    public User(User user)
    {
        name = user.name;
        location = user.location;
    }
}
class Program
{
    static void Main(string[] args)
    {
        // User object with Parameterized constructor
        User user = new User("Suresh Dasari", "Hyderabad");
        // Another User object (user1) by copying user details
        User user1 = new User(user);
        user1.name = "Rohini Alavala";
        user1.location = "Guntur";
        Console.WriteLine(user.name + ", " + user.location);
        Console.WriteLine(user1.name + ", " + user1.location);
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
}
```

noteghar.com
tlyghar.com

If you observe the above example, we created an instance of copy constructor (**user1**) and using an instance of **user** object as a parameter type. So the properties of the **user** object will be sent to the **user1** object, and we are changing the property values of

the **user1** object, but those will not affect the **user** object property values.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, we initialized a new instance to the values of an existing instance, but those changes not effected the existing instance values.

This is how you can use a copy constructor in the c# programming language to create a constructor with the parameter of the same class type based on our requirements.

C# Static Constructor with Examples

In c#, **Static Constructor** is useful to perform a particular action only once throughout the application. If we declare a constructor as **static**, it will be invoked only once, irrespective of the number of class instances. 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 the 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.
- CLR will invoke the static constructor, so we don’t have control over the static constructor execution order in c#.
- In c#, only one static constructor is allowed to create.

tiyeghar.com

C# Static Constructor Syntax

Following is the syntax of defining a static constructor in the c# programming language.

```
class User
{
   // Static Constructor
   static User()
   {
      // Your Custom Code
   }
}
```

If you observe the above syntax, we created a static constructor without having any parameters and access specifiers.

noteghar.com

C# Static Constructor Example

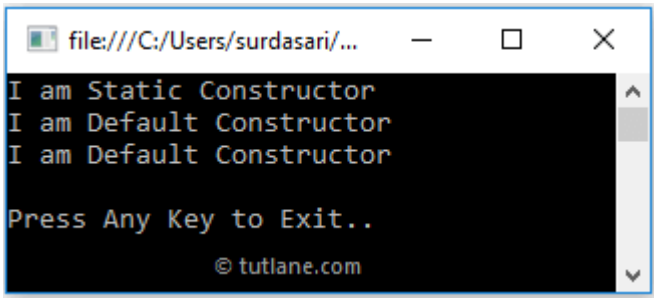
Following is the example of creating a static constructor in c# programming language to invoke the particular action only once throughout the program.

```
using System;

namespace Tutlane
{
    class User
    {
      // Static Constructor
      static User()
      {
        Console.WriteLine("I am Static Constructor");
      }
      // Default Constructor
      public User()
      {
        Console.WriteLine("I am Default Constructor");
      }
    }
    class Program
    {
       static void Main(string[] args)
       {
          // Both Static and Default constructors will invoke for first ins
tance
          User user = new User();
          // Only Default constructor will invoke
          User user1 = new User();
          Console.WriteLine("\nPress Enter Key to Exit..");
          Console.ReadLine();
       }
    }
}
```

noteghar.com

If you observe the above example, we created a **static constructor** and default constructor. Here the static constructor will be invoked only once for the first instance of a class.

When you execute the above c# program, we will get the result as shown below.



If you observe the above result, for the first instance of a class, both static and default constructors execute. For the second instance of a class, only the default constructor has been executed.

This is how you can use a static constructor in the c# programming language to execute particular action only once throughout the application based on your requirements.

C# Private Constructor with Examples

In c#, **Private Constructor** is a special instance constructor, and it is useful in classes that contain only **static** members. If a class contains one or more private constructors and no public constructors, then the other classes are not allowed to create an instance for that particular class except nested classes.

C# Private Constructor Syntax

Following is the syntax of defining a private constructor in the c# programming language.

```
class User
{
  // Private Constructor
   private User()
   {
      // Your Custom Code
   }
}
```

If you observe the above syntax, we created a private constructor without any parameters to prevent an automatic generation of the default constructor. If we didn't use any **access modifier** to define the constructor, then by default, it will treat it as **private**.

C# Private Constructor Example

Following is the example of creating a private constructor in the c# programming language to prevent other classes from creating an instance of a particular class.

```
using System;

namespace Tutlane
{
   class User
   {
      // private Constructor
      private User()
      {
         Console.WriteLine("I am Private Constructor");
      }
      public static string name, location;
      // Default Constructor
      public User(string a, string b)
      {
         name = a;
         location = b;
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         // The following comment line will throw an error because cons
tructor is inaccessible
```

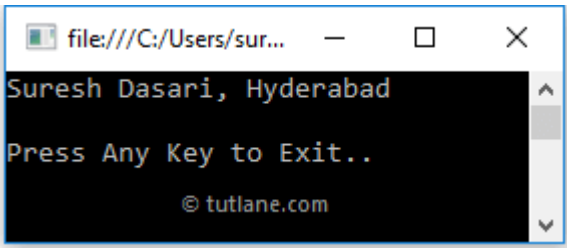
noteghar.com
tlyghar.com

```
        //User user = new User();

        // Only Default constructor with parameters will invoke
        User user1 = new User("Suresh Dasari", "Hyderabad");
        Console.WriteLine(User.name + ", " + User.location);
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
  }
}
```

If you observe the above example, we created a class with a **private constructor** and a default constructor with parameters. If you uncomment the commented line (User user = new User();), then it will throw an error because the constructor is private, so it won't allow you to create an instance for that class.

Here we are accessing class properties directly with the class name because those are **static** properties, so it won't allow you to access the instance name.

When you execute the above c# program, you will get the result as shown below.



This is how you can use a private constructor in the c# programming language to prevent creating an instance of a particular class based on our requirements.

Generics in C#

In c#, **generic** is a type that is used to define a class, structure, interface or method with **placeholders** (type parameters) to indicate that they can store or use one or more of the types. In c#, the compiler will replace placeholders with the specified type at compile time.

In c#, mostly we will use a generics with collections and the methods that operate on them to specify a type of objects to be stored in a collection. The generics are introduced in .NET Framework 2.0 with a new namespace called **System.Collections.Generic**.

In c#, generics are useful to improve the code reusability, type safety, and the performance when compared with the non-generic types such as arraylist.

C# Generics Declaration

To define a class or method as generic, then we need to use a type parameter as a placeholder with an angle (**<>**) brackets.

Following is the example of defining a generic class with type parameter (**T**) as a placeholder with an angle (**<>**) brackets.

```
public class GenericClass<T>
{
    public T msg;
    public void genericMethod(T name, T location)
    {
        Console.WriteLine("{0}", msg);
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("Location: {0}", location);
    }
}
```

If you observe above class, we created a class (**GenericClass**) with one parameter (**msg**) and method (genericMethod) using type parameter (**T**) as placeholder with an angle (**<>**) brackets.

Here, the angle (**<>**) brackets will indicate a **GenericClass** is generic and type parameter (**T**) is used to accept a requested type. The type parameter name can be anything like **X** or **U** or etc. based on our requirements.

Generally, while creating an instance of the class we need to specify an actual type, then the compiler will replace all the type parameters such as **T** or **U** or **X**, etc. with specified actual type. In c#, the following is the example of creating an instance of a generic class.

```
// Instantiate Generic Class, string is the type argument
GenericClass<string> gclass = new GenericClass<string>();
gclass.msg = "Welcome to Tutlane";
gclass.genericMethod("Suresh Dasari", "Hyderabad");
```

flygehar.com

If you observe the above code, we are sending a type as "**string**" so the compiler will substitute all the type parameters (**T**) with defined type "**string**" and our class (**GenericClass**) will be like as shown below.

```
public class GenericClass
{
    public string msg;
    public void genericMethod(string name, string location)
    {
        Console.WriteLine("{0}", msg);
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("Location: {0}", location);
    }
}
```

In c#, we can also create our own custom generic types and methods to provide our own generalized solutions that are type-safe and efficient.

C# Generic Class Example

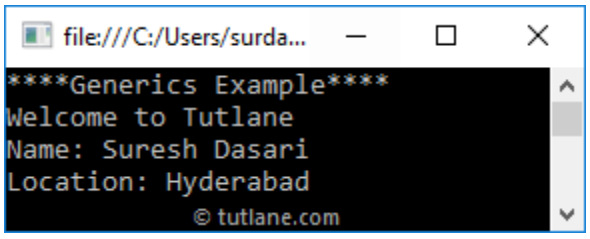
Following is the example of creating a generic class using type parameter (**T**) with angle (**<>**) brackets in c# programming language.

```
using System;
namespace Tutlane
{
    public class GenericClass<T>
    {
        public T msg;
        public void genericMethod(T name, T location)
        {
            Console.WriteLine("{0}", msg);
            Console.WriteLine("Name: {0}", name);
            Console.WriteLine("Location: {0}", location);
        }
    }
    class Program
    {
```

noteghar.com

```
        static void Main(string[] args)
        {
            Console.WriteLine("****Generics Example****");
            // Instantiate Generic Class, string is the type argument
            GenericClass<string> gclass = new GenericClass<string>();
            gclass.msg = "Welcome to Tutlane";
            gclass.genericMethod("Suresh Dasari", "Hyderabad");
            Console.ReadLine();
        }
    }
}
```

When you execute the above c# program, you will get the result as shown below.



This is how we can use generics in c# to create generic classes or methods based on our requirements.

C# Generic Class as Base / Derived Class

In c#, you can use the generic class as a base class, but we need to provide a type instead of type parameter for the base class because there is no way to send a required type argument to instantiate a base class at run time.

Following is the example of using a generic class as a base class in c# programming language.

```
// No Error
class DClass1 : GenericClass<string> {
// implementation
}
// Compile Time Error
//class DClass2 : GenericClass<T> {
// implementation
//}
```

In case, if derived class is generic, then we don't need to specify a type for generic base class instead we can use type parameter (**T**).

Following is the example of defining a generic derived class in c# programming language.

```
// No Error
class DClass1 : GenericClass<string> {
// implementation
}
// No Error
class DClass2<T> : GenericClass<T>{
// implementation
}
```

tiyeghar.com

C# Generic Methods

In c#, if we define a method with type parameter, then it is called a **generic method**. Following is the example of defining a generic method with type parameter using angle (**<>**) brackets.

```
public void genericMethod<T>(T a, T b)
{
     // Implementation
}
```

This generic method can be called either by specifying the type of argument or without an argument like as shown below.

```
genericMethod<int>(1, 2);

//or

genericMethod(1, 2);
```

C# Generic Method Example

In c#, you can call a generic method by passing any type of arguments. Following is the example of defining a generic method in c# programming language.

noteghar.com

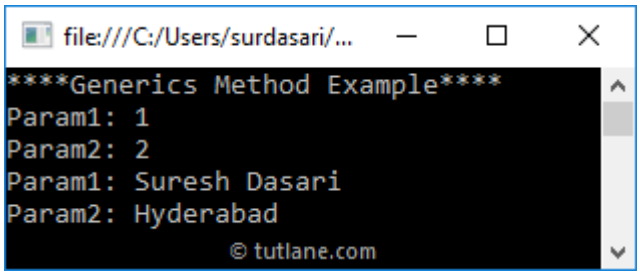
```
using System;
namespace Tutlane
{
    public class SampleClass
    {
        public void GMethod<T>(T a, T b)
        {
            Console.WriteLine("Param1: {0}", a);
            Console.WriteLine("Param2: {0}", b);
        }
    }
    class Program
    {
```

```
        static void Main(string[] args)
        {
            Console.WriteLine("****Generics Method Example****");
            SampleClass s = new SampleClass();
            s.GMethod<int>(1, 2);
            s.GMethod("Suresh Dasari", "Hyderabad");
            Console.ReadLine();
        }
    }
}
```

flyghar.com

If you observe above code, we are calling our generic method (**GMethod**) with or without type parameter and sending a different type of arguments based on our requirements.

When you execute the above c# program, you will get the result as shown below.



This is how you can define generic methods in c# programming language based on our requirements.

C# Generic Delegates

In c#, a generic delegate will be same as a normal delegate but the only difference is a generic delegate will have a generic type with angle (<>) brackets.

Following is the example of defining a generic delegate in c# programming language.

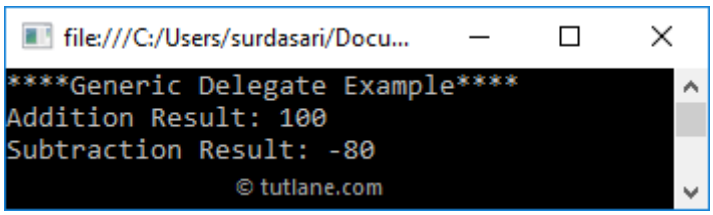
noteghar.com

```
using System;

namespace Tutlane
{
    // Declare Generic Delegate
    public delegate T SampleDelegate<T>(T a, T b);
    class MathOperations
    {
        public int Add(int a, int b)
        {
```

```
            return a + b;
        }
        public int Subtract(int x, int y)
        {
            return x - y;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("****Generic Delegate Example****");
            MathOperations m = new MathOperations();
            // Instantiate delegate with add method
            SampleDelegate<int> dlgt = new SampleDelegate<int>(m.Add);
            Console.WriteLine("Addition Result: " + dlgt(10, 90));
            // Instantiate delegate with subtract method
            dlgt = m.Subtract;
            Console.WriteLine("Subtraction Result: " + dlgt(10, 90));
            Console.ReadLine();
        }
    }
}
```

If you observe above code, we defined a delegate (**SampleDelegate**) with generic type parameter (**T**) using angle (**<>**) brackets and accessing it by creating an instance of delegate with required argument (**int**).

When you execute the above c# program, you will get a result as shown below.



This is how we can use generics with delegates based on our requirements in c# programming language.

C# Generics Overview

The following are the important properties of generics in c# programming language.

- In c#, generics are represented by using angle bracket **<>**.

- To define a class or method as generic, then we need to use a type parameter as a placeholder with an angle (**<>**) brackets.
- The compiler will replace all the placeholders with the specified type at compile time.
- In c#, generics are useful to improve the code reusability, type safety, and the performance when compared with the non-generic types such as arraylist.
- In c#, you can use generics with interfaces, classes, methods, properties, delegates, events, and operators.

tiyeghar.com

noteghar.com

Generic Constraints in C#

In c#, generics are used to define a class or structure or methods with **placeholders** (type parameters) to indicate that they can use any of the types.

Following is the example of defining a generic class with type parameter (**T**) as a placeholder with an angle (**<>**) brackets.

flyghar.com

```
public class GenericClass<T>
{
    public T msg;
    public void genericMethod(T name, T location)
    {
        Console.WriteLine("{0}", msg);
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("Location: {0}", location);
    }
}
```

If you observe above class, we created a class (**GenericClass**) with one parameter (**msg**) and method (**genericMethod**) using type parameter (**T**) as placeholder with an angle (**<>**) brackets. Here the defined **GenericClass** doesn't know anything about the defined placeholder and it will accept any type either string or int or class, etc. based on our requirements.

In case, if we want to restrict a generic class to accept only the particular type of placeholder, then we need to use **Constraints**. By using **Constraints**, we can specify what type of placeholder the generic class can accept and the compiler will throw a compile-time error. In case, if we try to instantiate a generic class with the placeholder type that is not allowed by a constraint.

In c#, constraints are specified by using **where** keyword. The following table lists a different type of constraints available in c#.

notebar.com

Constraint	Description
where T : struct	The type argument must be a value type.
where T : unmanaged	The type of argument must not be a reference type.

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

flyghar.com

C# Generic Class with Constraints Example

Following is the example of defining a generic class with constraint using **where** contextual keyword in c# programming language.

```
public class GenericClass<T> where T: class
{
    public T msg;
    public void genericMethod(T name, T location)
    {
        Console.WriteLine("{0}", msg);
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("Location: {0}", location);
    }
}
```

If you observe above code, we defined a class (**GenericClass**) with where T: class constraint so the **GenericClass** will accept only reference type arguments.

noteghar.com

Following is the example of instantiating a generic class with valid reference type arguments such as string or class in c#.

```
// Instantiate Generic Class with Constraint
GenericClass<string> gclass = new GenericClass<string>();
GenericClass<User> gclass1 = new GenericClass<User>();
// Compile Time Error
//GenericClass<int> gclass11 = new GenericClass<int>();
```

If you observe above code, we created an instance of **GenericClass** using reference type arguments such as string and class. In case, if we uncomment the commented code, then we will get a compile time error because int is a value type, not a reference type.

C# Multiple Generic Constraints

In c#, you can apply multiple constraints on generic classes based on our requirements. Following is the code snippet of adding multiple constraints to generic class.

```
public class GenericClass<T, X> where T : class where X : struct
{
    // Your Implementation
}
```

If you observe the above code snippet, we added multiple constraints on generic class (**GenericClass**) using **where** keyword.

C# Generic Methods with Constraints

In c#, you can also apply constraints on generic methods based on our requirements. Following is the example of adding constraints on generic methods using **where** keyword.

```
public class GenericClass<T> where T : class
{
    public T msg;
    public void genericMethod<X>(T name, T location) where X : class
    {
        Console.WriteLine("{0}", msg);
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("Location: {0}", location);
    }
}
```

C# Generic Constraints Overview

Following are the important points which we need to remember about generic constraints in c# programming language.

- In c#, constraints are used to restrict a generics to accept only the particular type of placeholders.
- By using **where** keyword, we can apply constraints on generics.

- In c#, you can apply multiple constraints on generic classes or methods based on your requirements.
- In c#, you have a different type of constraints available, those are class, structure, unmanaged, new(), etc.

tiyeghar.com

Generic Collections in C#

In c#, generic collections will enforce a type safety so you can store only the elements which are having the same data type and these are provided by **System.Collections.Generic** namespace.

In the previous chapter, we learned about collections in c# and these are useful to store elements of different data types but the problem with collections is while retrieving an elements we need to perform a typecasting (boxing and unboxing) and it will affect application performance so to solve this problem c# introduced a generic collections with **System.Collections.Generic** namespace.

In c#, generic collections are the strongly-typed objects so it will allow only the same data type elements to store and it doesn't require any typecasting (boxing and unboxing) while storing and retrieving elements so performance will be improved.

Following are the different type of generic collection classes which are provided by **System.Collections.Generic** namespace.

Class	Description
List	It is useful to represent a list of objects that can be accessed by index.
Queue	It is useful to represent a **FIFO** (First In, First Out) collection of objects.
Stack	It is useful to represent a **LIFO** (Last In, First Out) collection of objects.
SortedList<K,V>	It is useful to represent a collection of key/value pairs that are sorted by a key.
Dictionary<K,V>	It is useful to represent a collection of key/value pairs that are organized based on the key.

Class	Description
HashSet	It is useful to store non-duplicate elements.

flyghar.com

C# List (List<T>)

In c#, **List** is a generic type of collection so it will allow storing only strongly typed objects i.e. elements of the same data type and the size of the list will vary dynamically based on our application requirements like adding or removing elements from the list.

In c#, the list is same as an ArrayList but the only difference is ArrayList is a non-generic type of collection so it will allow storing elements of different data types.

C# List Declaration

In c#, the list is a generic type of collection and it is provided by **System.Collections.Generic** namespace.

As discussed, the collection is a class so to define a list, you need to declare an instance of the list class before we perform any operations like add, delete, etc. like as shown below.

```
List<T> lst = new List<T>();
```

If you observe the above list declaration, we created a generic list (**lst**) with an instance of list class using type parameter (**T**) as a placeholder with an angle (**<>**) brackets.

Here, the angle (**<>**) brackets will indicate that the list is a **generic** type and type parameter (**T**) is used to represent the type of elements to be accepted by the list.

In c#, the generic list (**List<T>**) is an implementation of **IList<T>** interface so we can also use **IList<T>** interface to create an object of the generic list (**List<T>**) like as shown below.

```
IList<T> lst = new List<T>();
```

noteghar.com

C# List Initialization

Following is the example of initializing a generic list by specifying a required type to accept values to store.

```
List<string> lst = new List<string>();
```

If you observe above example, we defined a list (**lst**) with **string** data type to store only string elements.

C# List Properties

The following are some of the commonly used properties of the generic list in c# programming language.

Property	Description
Capacity	It is used to get or set the number of elements a list can contain.
Count	It is used to get the number of elements in the list.
Item	It is used get or set an element at the specified index.

C# List Methods

The following are some of the commonly used methods of the generic list to perform add, search, insert, delete or sort operations in c# programming language.

Method	Description
Add	It is used to add an element at the end of the List.
AddRange	It is used to add all the elements of the specified collection at the end of the List.
Clear	It will remove all the elements from the List.
Contains	It is used determine whether the specified element exists in the List or not.
CopyTo	It is used to copy the entire List to a compatible one-dimensional array.
Find	It is used to search for an element that matches the conditions defined by the specified predicate and

Method	Description
	returns the first occurrence of the List.
FindAll	It is used to retrieve all the elements that match the conditions defined by the specified predicate.
ForEach	It is used to iterate through the List to access elements.
Insert	It is used to insert an element into the List at the specified index.
InsertRange	It is used to insert all the elements of the specified collection into List starting from the specified index.
Remove	It is used to remove the first occurrence of a specified element from the List.
RemoveAt	It is used to remove an element from the List based on the specified index position.
RemoveRange	It is used to remove a range of elements from the List.
Reverse	It reverses the order of List elements.
Sort	It sorts the elements in the List.
ToArray	It will copy the elements of the List to new array objects.

flyghar.com

C# Generic List (List<T>) Example

Following is the example of using the generic list (**List<T>**) in the c# programming language.

noteghar.com

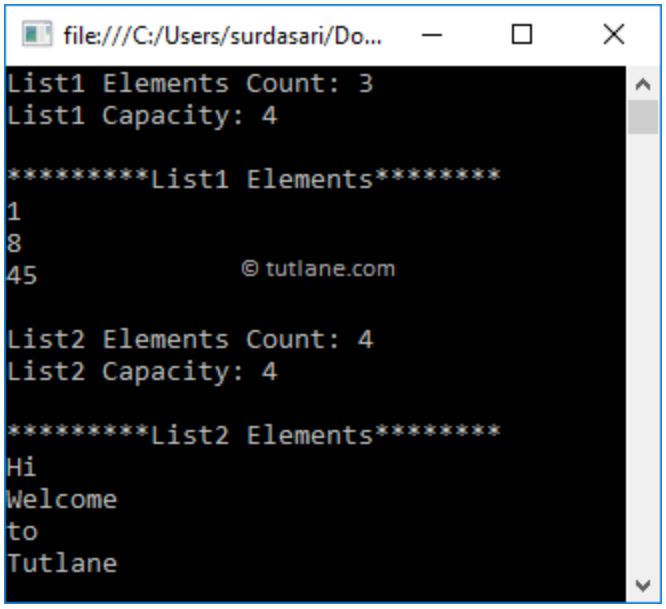
```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            // Creating and initializing list
            List<int> lst = new List<int>();
            lst.Add(1);
            lst.Add(8);
            lst.Add(45);
            List<string> lst2 = new List<string>();
            lst2.Add("Hi");
            lst2.Add("Welcome");
            lst2.Add("to");
            lst2.Add("Tutlane");
            Console.WriteLine("List1 Elements Count: " + lst.Count);
            Console.WriteLine("List1 Capacity: " + lst.Capacity);
            Console.WriteLine("*********List1 Elements********");
            // Accessing list elements
            foreach (var item in lst)
            {
                Console.WriteLine(item);
            }
            Console.WriteLine("List2 Elements Count: " + lst2.Count);
            Console.WriteLine("List2 Capacity: " + lst2.Capacity);
            Console.WriteLine("*********List2 Elements********");
            foreach (var item in lst2)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we are able to define a new generic lists (**lst**, **lst2**) collection by using **System.Collections.Generic** namespace. Here, we added only the defined data type (**string**) values to the newly created lists (**lst**, **lst2**) by using **Add** method and accessing those generic list (**lst**, **lst2**) elements by using foreach loop.

When you execute the above c# program, we will get the result as shown below.

noteghar.com



This is how you can use a generic list in c# to store the group of defined type elements based on our requirements.

The following are some of the useful examples to work with a generic list in c# programming language.

C# Add Elements to List

In c#, you can add elements to the list either by using **Add** / **AddRange** methods or at the time of initialization based on our requirements.

Following is the example of adding elements to the list using **Add** or **AddRange** methods in c#.

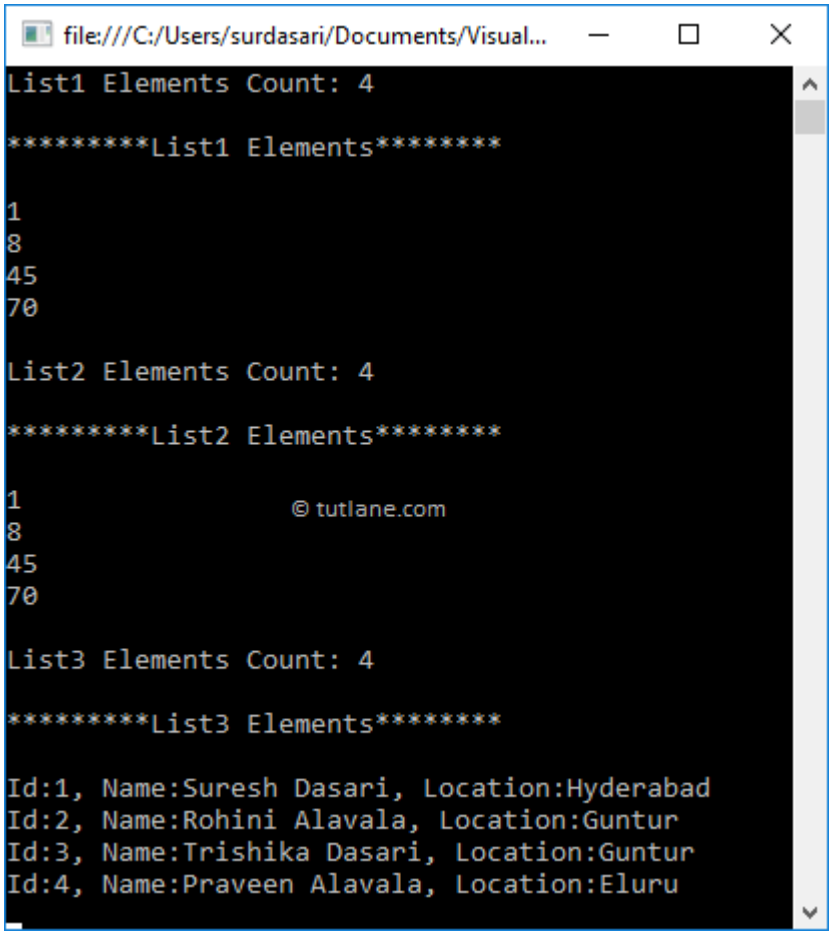
```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Location { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Creating and initializing list
            List<int> lst = new List<int>() { 1, 8, 45, 70 };
```

```
            Console.WriteLine("List1 Elements Count: " + lst.Count);
            Console.WriteLine("*********List1 Elements********");
            // Accessing list elements
            foreach (var item in lst)
            {
                Console.WriteLine(item);
            }
            //AddRange method
            List<int> lst2 = new List<int>();
            lst2.AddRange(lst);
            Console.WriteLine("List2 Elements Count: " + lst2.Count);
            Console.WriteLine("*********List2 Elements********");
            foreach (var item in lst2)
            {
                Console.WriteLine(item);
            }
            List<User> users = new List<User>(){
            new User{Id= 1, Name= "Suresh Dasari", Location= "Hyderabad
"},
            new User{Id = 2, Name = "Rohini Alavala", Location = "Guntu
r"}
            };
            users.Add(new User { Id = 3, Name = "Trishika Dasari", Loca
tion = "Guntur" });
            users.Add(new User { Id = 4, Name = "Praveen Alavala", Loca
tion = "Eluru" });
            Console.WriteLine("List3 Elements Count: " + users.Count);
            Console.WriteLine("*********List3 Elements********");
            foreach (User u in users)
            {
                Console.WriteLine("Id:{0}, Name:{1}, Location:{2}", u.I
d, u.Name, u.Location);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created a multiple lists (**lst**, **lst2**, **users**) and adding an elements to the lists in different ways like added some of the elements during initialization time and some other elements by using **Add** & **AddRange** methods.

When you execute the above c# program, you will get the result as shown below.

noteghar.com flyghar.com



flyghar.com

C# Access List Elements

In c#, we have different ways to access list elements i.e. either by using index positions or by iterating through the list using for / foreach loops.

Following is the example of accessing list elements in different ways.

noteghar.com

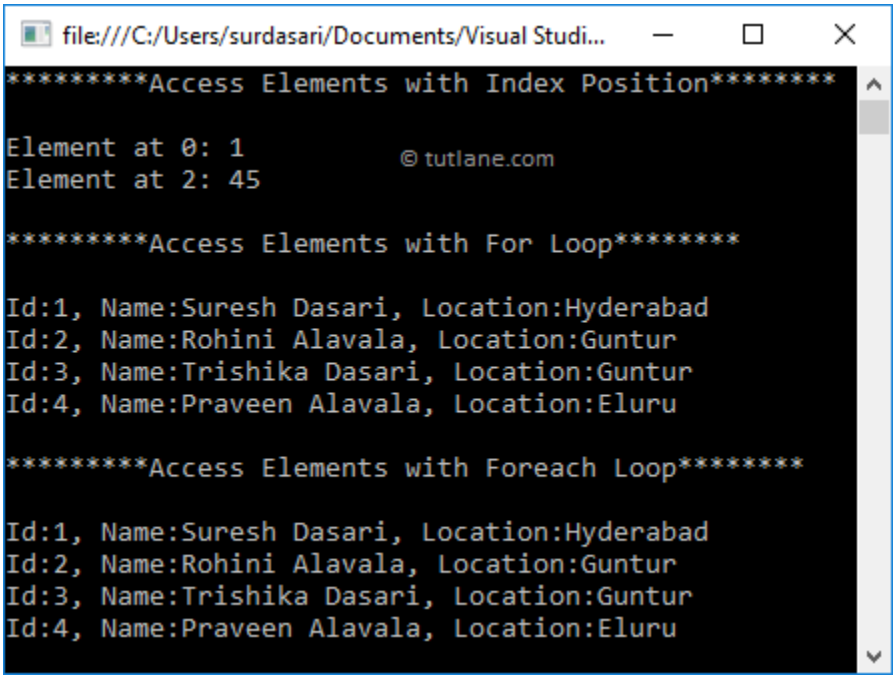
```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Location { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Creating and initializing list
            List<int> lst = new List<int>() { 1, 8, 45, 70 };
```

```
            Console.WriteLine("*********Access Elements with Index Posi
tion********");
            Console.WriteLine("Element at 0: " + lst[0]);
            Console.WriteLine("Element at 2: " + lst[2]);
            // Creating List
            List<User> users = new List<User>(){
            new User{Id= 1, Name= "Suresh Dasari", Location= "Hyderabad
"},
            new User{Id = 2, Name = "Rohini Alavala", Location = "Guntu
r"},
            new User { Id = 3, Name = "Trishika Dasari", Location = "Gu
ntur" },
            new User { Id = 4, Name = "Praveen Alavala", Location = "El
uru" }
            };
            Console.WriteLine("*********Access Elements with For Loop**
******");
            // for loop to access list elements
            for (int i = 0; i < users.Count; i++)
            {
                Console.WriteLine("Id:{0}, Name:{1}, Location:{2}", use
rs[i].Id, users[i].Name, users[i].Location);
            }
            Console.WriteLine("*********Access Elements with Foreach Lo
op********");
            // foreach loop to access list elements
            foreach (User u in users)
            {
                Console.WriteLine("Id:{0}, Name:{1}, Location:{2}", u.I
d, u.Name, u.Location);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we are accessing generic list elements in different ways by using index positions, for and foreach loops based on our requirements.

When you execute the above c# program, you will get the result as shown below.

noteghar.com
flyghar.com



C# Insert Elements into List

In c#, we can insert elements into the list either by using **Insert()** or **InsertRange()** methods.

Following is the example of inserting elements into the list by using **Insert** and **InsertRange** methods in c#.

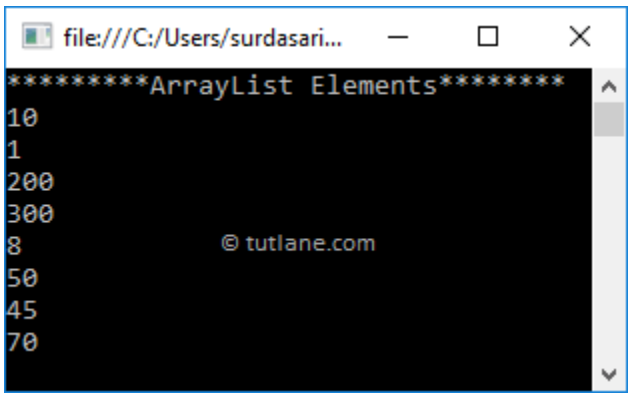
```
using System;
using System.Collections.Generic;
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Creating and initializing list
            List<int> lst = new List<int>() { 1, 8, 45, 70 };
            // inserting elements into list
            lst.Insert(0, 10);
            lst.Insert(3, 50);
            List<int> lst2 = new List<int>() { 200, 300 };
            // inserting lst2 into lst at position 2
            lst.InsertRange(2, lst2);
            Console.WriteLine("*********ArrayList Elements********");
            foreach (var item in lst)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
```

noteghar.com
tlyghar.com

```
        }
    }
}
```

If you observe above example, we inserted an elements into list at different index positions (**0**, **3**) by using **Insert()** method. Same way, by using **InsertRange()** method we are inserting all the element of newly created list (**lst2**) into **lst** at index position **2**.

When you execute the above c# program, you will get the result as shown below.



C# Remove Elements from List

In c#, we have different methods to remove elements from the list i.e. either by
using **Remove()** or **RemoveAt()** or **RemoveRange()** methods.

Following is the example of deleting the elements from the list in c# programming language.

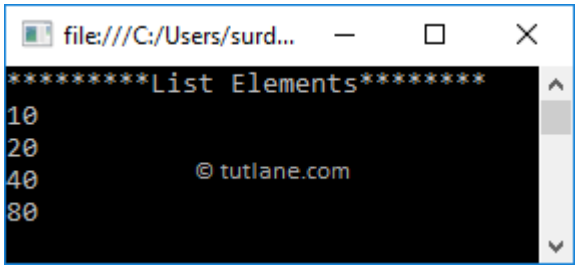
```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Creating and initializing list
            List<int> lst = new List<int>() { 10, 20, 30, 40, 50, 60, 7
0, 80 };
            // Removing an element which is having a value 50
            lst.Remove(50);
            // Removing an element at index 2
            lst.RemoveAt(2);
            // Removing 2 elements starting from index 3
            lst.RemoveRange(3, 2);
```

```
            Console.WriteLine("*********List Elements********");
            foreach (var item in lst)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Remove()** method to delete particular value of element from list and used a **RemoveAt()** method to delete an element at specified index position.

Same way, we used the **RemoveRange()** method to delete the specified number of elements from the list starting from the specified index position.

When you execute the above c# program, we will get the result as shown below.



In case, if you want to remove all the elements from the list, then use the **Clear()** method.

Same way, you can use different methods like **Find()** method to find elements of list and **Reverse()** method to reverse the order of list elements, etc. based on our requirements.

C# List Check If Item Exists

By using **Contains()** method, we can check whether the specified element exists in the list or not. In case, if it exists it will return **true** otherwise **false**.

Following is the example of using **Contains()** method to check for an item that exists in the list or not in c#.

```
using System;
using System.Collections.Generic;
```

```
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            // Creating and initializing list
            List<int> lst = new List<int>() { 10, 20, 30, 40, 50, 60, 7
0, 80 };
            // Check for an item 50 exists in list or not
            Console.WriteLine("Item Exists: " + lst.Contains(50));
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Contains()** method to check for an item (**50**) exists in list (**lst**) or not.

When you execute the above c# program, we will get the result as shown below.

```
Item Exists: True
```

C# List (List<T>) Overview

The following are the important points that need to remember about a list in c#.

- The list is used to store elements of the same data type and the size of the list can grow or shrink dynamically by adding or removing elements.
- The list is a generic type of collection and available with the **System.Collections.Generic** namespace.
- We can access the list elements either by using loops (for and foreach) or with a particular index position.
- **List<T>** can be assigned to **IList<T>** or **List<T>** type of a variable and you can store multiple null and duplicate values in the list based on your requirement.
- The list provided different methods to perform multiple operations like add, insert, delete, sort, reverse, etc. on elements of arraylist.

noteghar.com
flyghar.com

C# SortedList (SortedList<TKey,TValue>)

In c#, **SortedList** is a generic type of collection and it is used to store a collection of key/value pairs that are sorted by key based on the associated **IComparer<T>** implementation.

By default, the sortedlist will sort a key/value pairs in ascending order of the key and the sortedlist will allow storing only the strongly-typed objects i.e. the key/value pairs of the specified data type.

In c#, the sortedlist will allow us to store duplicate values but the keys must be unique and cannot be null to identify the values in sortedlist and the size of sortedlist will vary dynamically so you can add or remove elements from the sortedlist based on our application requirements.

C# SortedList Declaration

In c#, sortedlist is a generic type of collection and it is provided by **System.Collections.Generic** namespace.

As discussed, the collection is a class so to define a sortedlist you need to declare an instance of the sortedlist class before we perform any operations like add, delete, etc. like as shown below.

```
SortedList<TKey, TValue> slist = new SortedList<TKey, TValue>();
```

If you observe above sortedlist declaration, we created a generic sortedlist (**slst**) with an instance of sortedlist class using type parameters (**TKey**, **TValue**) as placeholders with an angle (**<>**) brackets.

Here, the angle (**<>**) brackets will indicate that the sortedlist is a **generic** type and type parameter **TKey** is to represent a type of keys to be accepted by sortedlist and **TValue** is used to represent a type of values to be accepted by sortedlist.

C# SortedList Initialization

Following is the example of initializing a generic sortedlist by specifying a required type for key and value.

```
SortedList<string, int> slist = new SortedList<string, int>();
```

If you observe the above example, we defined a sortedlist (**slist**) with the required key and value types to store. Here, the sortedlist will store a key of string type and value of **int** type.

Internally, the sortedlist will maintain a two object[] arrays, one for **keys** and another one for **values** so when we add key/value pair, it will perform a binary search using the key to find an appropriate index to store a key and value in respective arrays.

C# SortedList Properties

The following are some of the commonly used properties of generic sortedlist in c# programming language.

tiyeghar.com

Property	Description
Capacity	It is used to get or set the number of elements a sortedlist can contain.
Count	It is used to get the number of key/value pair elements in the list.
Item[TKey]	It is used to get or set the value associated with the specified key.
Keys	It is used to get a collection of keys in sortedlist.
Values	It is used to get a collection of values in sortedlist.

C# SortedList Methods

The following are some of the commonly used methods of generic sortedlist to perform add, search, insert, delete or sort operations in c# programming language.

Method	Description
Add	It is used to add elements with specified key and value in SortedList<TKey, TValue>.
Clear	It will remove all the elements from SortedList<TKey, TValue>.
ContainsKey	It is used to determine whether the specified key exists in the SortedList<TKey, TValue> or not.

tiyeghar.com

Method	Description
ContainsValue	It is used to determine whether the specified value exists in SortedList<TKey, TValue> or not.
IndexOfKey	It is used to get the index value of the specified key stored in the SortedList<TKey, TValue>.
IndexOfValue	It is used to get the index value of specified value stored in the SortedList<TKey, TValue>.
Remove	It is used to remove an element from SortedList<TKey, TValue> with the specified key.
RemoveAt	It is used to remove an element at the specified index position from SortedList<TKey, TValue>.
TryGetValue	It is used to get the value associated with the specified key.

flyghar.com

C# Add Elements to SortedList

In c#, while adding elements to sortedlist we need to make sure that there will not be any duplicate keys because sortedlist will allow us to store duplicate values but keys must be unique to identify the values in the list.

Following is the example of adding a key/value pair elements to sorted list in different ways.

noteghar.com

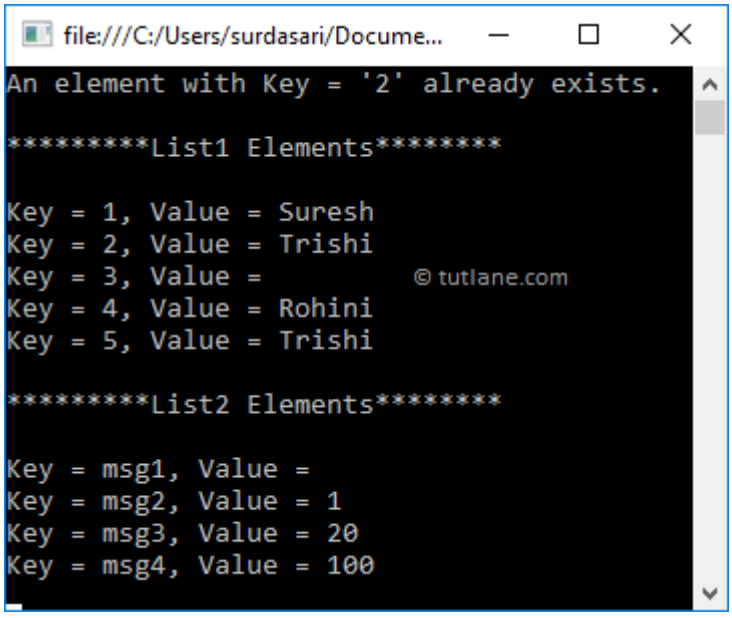
```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create a new sorted list with int keys and string values.
            SortedList<int, string> slst = new SortedList<int, string>(
);
            // Add elements to the list. No duplicate keys allowed but
values can be duplicate
            slst.Add(1, "Suresh");
            slst.Add(4, "Rohini");
```

```
            slst.Add(2, "Trishi");
            slst.Add(3, null);
            // Another way to add elements. If key not exist, then that
key adds a new key/value pair.
            slst[5] = "Trishi";
            // Add method throws exception if key already in the list
            try
            {
                slst.Add(2, "Praveen");
            }
            catch (ArgumentException)
            {
                Console.WriteLine("An element with Key = '2' already ex
ists.");
            }
            Console.WriteLine("*********List1 Elements********");
            // Accessing elements as KeyValuePair objects.
            foreach (KeyValuePair<int, string> item in slst)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }

            // Creating and initializing list
            SortedList<string, int?> slst2 = new SortedList<string, int
?> {
                                                {"msg2", 1},
                                                {"msg3", 20},
                                                {"msg4", 100},
                                                {"msg1", null}
                                            };
            Console.WriteLine("*********List2 Elements********");
            // Accessing elements as KeyValuePair objects.
            foreach (KeyValuePair<string, int?> item in slst2)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we are able to define a new generic sortedlist (**slst**, **slst2**) collections by using **System.Collections.Generic** namespace. Here, we added only the defined data type keys and values to the newly created lists (**slst**, **slst2**) in different ways.

As discussed, the **Add** method will throw an exception in case if we try to add a key (**2**) which is already existing so to handle that exception we used a try-catch block.

When you execute the above c# program, we will get the result as shown below.



If you observe the above result, by default all the list elements are sorted in ascending order based on the key and we got an exception when we tried to add a key (**2**) which is already existing and added a key (**5**) which is not existing in the list.

C# Access SortedList Elements

In c#, we have different ways to access sorted list elements i.e. either by using index positions or by iterating through the list using for / foreach loops.

Following is the example of accessing the sortedlist elements in different ways.

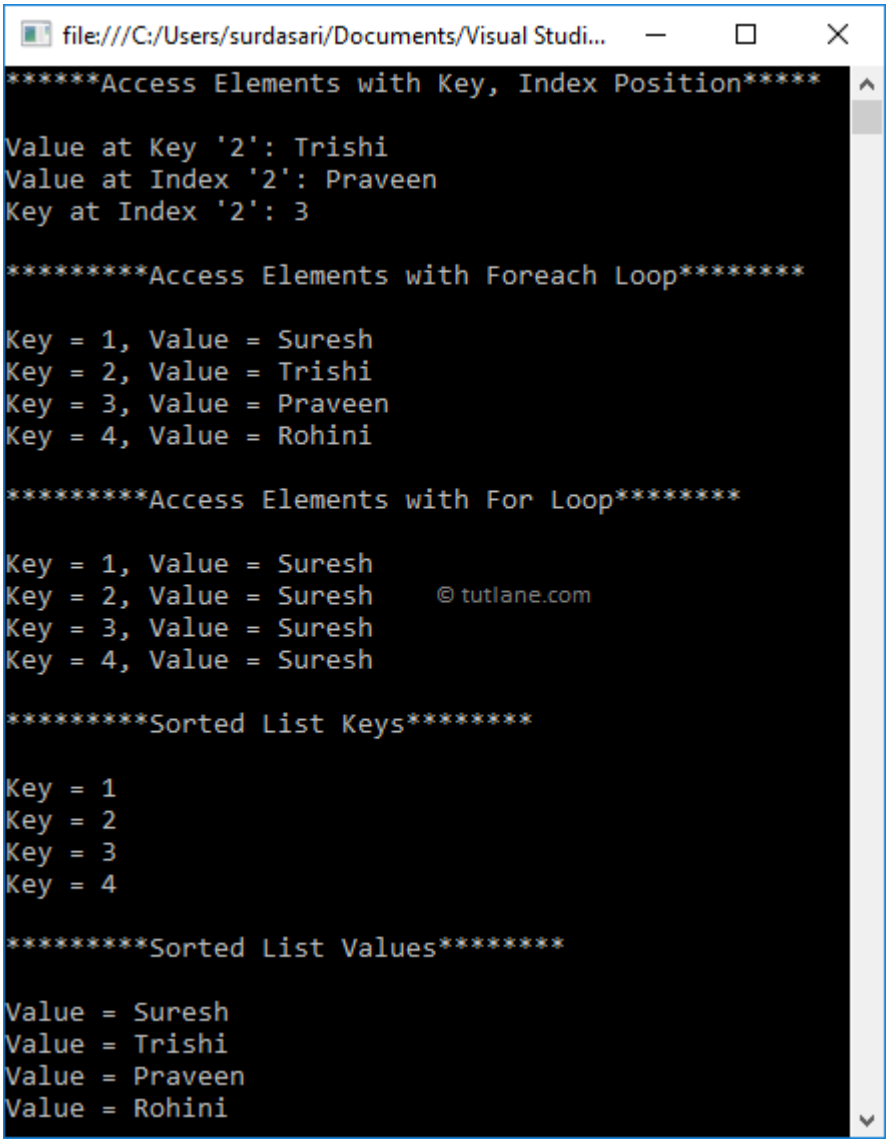
```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create a new sorted list
```

```
            SortedList<int, string> slst = new SortedList<int, string>(
);
            slst.Add(1, "Suresh");
            slst.Add(4, "Rohini");
            slst.Add(2, "Trishi");
            slst.Add(3, "Praveen");
            // Access value with key
            string val1 = slst[2];
            // Access value with index position
            string val2 = slst.Values[2];
            // Access key with index position
            int val3 = slst.Keys[2];
            Console.WriteLine("******Access Elements with Key, Index Po
sition*****");
            Console.WriteLine("Value at Key '2': " + val1);
            Console.WriteLine("Value at Index '2': " + val2);
            Console.WriteLine("Key at Index '2': " + val3);
            Console.WriteLine("*********Access Elements with Foreach Lo
op********");
            foreach (KeyValuePair<int, string> item in slst)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.WriteLine("*********Access Elements with For Loop**
******");
            for (int i = 0; i < slst.Count; i++)
            {
                Console.WriteLine("Key = {0}, Value = {1}", slst.Keys[i
], slst.Values[0]);
            }
            Console.WriteLine("*********Sorted List Keys********");
            foreach (var item in slst.Keys)
            {
                Console.WriteLine("Key = {0}", item);
            }
            Console.WriteLine("*********Sorted List Values********");
            foreach (var item in slst.Values)
            {
                Console.WriteLine("Value = {0}", item);
            }
            Console.ReadLine();
        }
    }
}
```

noteghar.com
flyghar.com

If you observe above example, we are accessing generic list elements in different ways by using index positions, for and foreach loops based on our requirements.

When you execute the above c# program, we will get the result as shown below.



© tutlane.com

C# Remove Elements from SortedList

In c#, we have different methods to remove elements from the sorted list i.e. either by using **Remove()** or **RemoveAt()** methods.

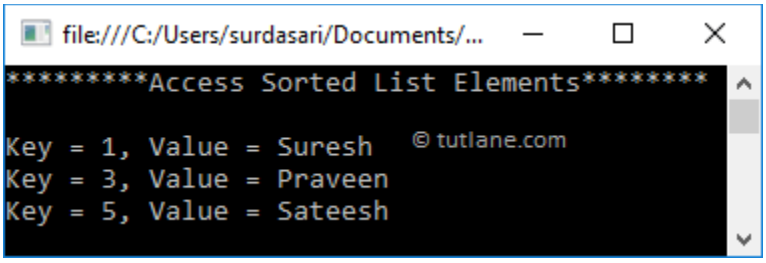
Following is the example of deleting elements from the sorted list in c# programming language.

```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            //Create a new sorted list
            SortedList<int, string> slst = new SortedList<int, string>(
);
            slst.Add(1, "Suresh");
            slst.Add(4, "Rohini");
            slst.Add(2, "Trishi");
            slst.Add(3, "Praveen");
            slst.Add(5, "Sateesh");
            // Remove element with key 2
            slst.Remove(2);
            // Remove element at index position 2
            slst.RemoveAt(2);
            Console.WriteLine("*********Access Sorted List Elements****
****");
            Console.WriteLine();
            foreach (KeyValuePair<int, string> item in slst)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Remove()** method to delete a particular key of element from the list and used a **RemoveAt()** method to delete an element at specified index position.

When you execute the above c# program, you will get the result as shown below.



In case, if you want to remove all the elements from the sorted list, then use the **Clear()** method.

C# SortedList Check If Item Exists

By using **ContainsKey()** and **ContainsValue()** methods, you can check whether the specified element exists in a sorted list or not. In case, if it exists these methods will return **true** otherwise **false**.

Following is the example of
using **ContainsKey()** and **ContainsValue()** methods to check for an item that exists in a sorted list or not in c#.

```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create a new sorted list
            SortedList<int, string> slst = new SortedList<int, string>(
);
            slst.Add(1, "Suresh");
            slst.Add(4, "Rohini");
            slst.Add(2, "Trishi");
            slst.Add(3, "Praveen");
            slst.Add(5, "Sateesh");
            Console.WriteLine("Contains Key 2: {0}", slst.ContainsKey(2
));
            Console.WriteLine("Contains Value 'Tutlane': {0}", slst.Con
tainsValue("Tutlane"));
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used
a **ContainsKey()** and **ContainsValue()** methods to check for a particular keys and values exists in sorted list (**slst**) or not.

When you execute the above c# program, we will get the result as shown below.

```
Contains Key 2: True
Contains Value 'Tutlane': False
```

C# SortedList (SortedList<TKey, TValue>)

Overview

The following are the important points that need to remember about the sorted list in c#.

- SortedList is used to store a collection of key/value pairs that are sorted by key based on the associated **IComparer<T>** implementation.
- SortedList will allow us to store duplicate values but keys must be unique to identify the values in the sorted list.
- In SortedList, the keys cannot be null but the values can be null.
- You can access SortedList elements either by using keys, index positions or with for and foreach loops. In foreach loop, we need to use **KeyValuePair<TKey, TValue>** to get a key/value pairs from SortedList.

flyghar.com
noteghar.com

C# Dictionary

In c#, **Dictionary** is a generic type of collection and it is used to store a collection of key/value pairs that are organized based on the key. The dictionary in c# will allow storing only the strongly-typed objects i.e. the key/value pairs of the specified data type.

In c#, while storing the elements in the dictionary object, you need to make sure that the keys are unique because the dictionary object will allow us to store duplicate values but the keys must be unique.

The size of the dictionary object will vary dynamically so you can add or remove elements from the dictionary based on our requirements.

In c#, the dictionary object is same as hashtable object but the only difference is the dictionary object is used to store a key-value pair of same data type elements.

C# Dictionary Declaration

In c#, the dictionary is a generic type of collection and it is provided by **System.Collections.Generic** namespace.

As discussed, the collection is a class so to define a dictionary, you need to declare an instance of dictionary class before we perform any operations such as add, delete, etc. like as shown below.

```
Dictionary<TKey, TValue> dct = new Dictionary<TKey, TValue>();
```

If you observe the above dictionary declaration, we created a generic dictionary (**dct**) with an instance of dictionary class using type parameters (**TKey**, **TValue**) as placeholders with an angle (**<>**) brackets.

Here, the angle (**<>**) brackets will indicate that the dictionary is a generic type and type parameter **TKey** is to represent a type of keys to be accepted by dictionary and **TValue** is used to represent a type of values to be accepted by dictionary.

In c#, the generic dictionary (**Dictionary<T>**) is an implementation of **IDictionary<TKey, TValue>** interface so we can also use **IDictionary<TKey, TValue>** interface to create an object of the generic dictionary (**Dictionary<TKey, TValue>**) like as shown below.

```
IDictionary<TKey, TValue> dct = new Dictionary<TKey, TValue>();
```

C# Dictionary Initialization

Following is the example of initializing a generic dictionary by specifying a required type for key and value.

```
Dictionary<string, int> dct = new Dictionary<string, int>();
```

If you observe the above example, we defined a dictionary (**dct**) with the required key and value types to store. Here, the dictionary object will store a key of **string** type and value of **int** type.

flyghar.com

C# Dictionary Properties

The following are some of the commonly used properties of dictionary objects in c# programming language.

Property	Description
Count	It is used to get the number of key/value pair elements in the dictionary.
Item[TKey]	It is used to get or set the value associated with the specified key.
Keys	It is used to get a collection of keys in the dictionary.
Values	It is used to get a collection of values in the dictionary.

C# Dictionary Methods

The following are some of the commonly used methods of the generic dictionary to perform add, search, insert, delete or sort operations in c# programming language.

noteghar.com

Method	Description
Add	It is used to add elements to the dictionary object with a specified key and value.
Clear	It will remove all the keys and values from the Dictionary<TKey, TValue>.

Method	Description
ContainsKey	It is used to determine whether the specified key exists in Dictionary<TKey, TValue> or not.
ContainsValue	It is used to determine whether the specified value exists in Dictionary<TKey, TValue> or not.
Remove	It is used to remove an element from Dictionary<TKey, TValue> with the specified key.
TryGetValue	It is used to get the value associated with the specified key.

C# Add Elements to Dictionary

As discussed, while adding elements to the dictionary object we need to make sure that there will not be any duplicate keys.

Following is the example of adding a key/value pair elements to dictionary objects in different ways.

```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create a new dictionary with int keys and string values.
            Dictionary<int, string> dct = new Dictionary<int, string>()
;
            // Add elements to the dictionary object.
            // No duplicate keys allowed but values can be duplicate
            dct.Add(1, "Suresh");
            dct.Add(4, "Rohini");
            dct.Add(2, "Trishi");
            dct.Add(3, null);
            // Another way to add elements.
            // If key not exist, then that key adds a new key/value pai
r.
            dct[5] = "Trishi";
            // Add method throws exception if key already in dictionary
            try
            {
                dct.Add(2, "Praveen");
```

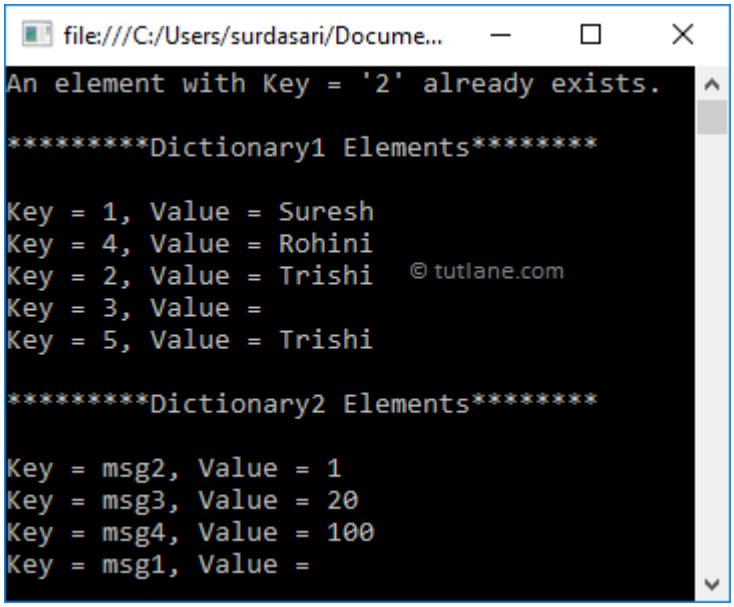
noteghar.com

```
            }
            catch (ArgumentException)
            {
                Console.WriteLine("An element with Key = '2' already ex
ists.");
            }
            Console.WriteLine("*********Dictionary1 Elements********");
            // Accessing elements as KeyValuePair objects.
            foreach (KeyValuePair<int, string> item in dct)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }

            // Creating and initializing dictionary
            Dictionary<string, int?> dct2 = new Dictionary<string, int?
> {
                                                {"msg2", 1},
                                                {"msg3", 20},
                                                {"msg4", 100},
                                                {"msg1", null}
                                            };
            Console.WriteLine("*********Dictionary2 Elements********");
            // Accessing elements as KeyValuePair objects.
            foreach (KeyValuePair<string, int?> item in dct2)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we are able to define a new generic dictionary (**dct**, **dct2**) collections by using **System.Collections.Generic** namespace. Here, we added only the defined data type keys and values to the newly created dictionaries (**dct**, **dct2**) in different ways.

As discussed, the **Add** method will throw an exception in case if we try to add a key (**2**) which is already existing so to handle that exception we used a **try-catch** block.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, we got an exception when we tried to add a key (**2**) which is already existing and added a key (**5**) which is not existing in the dictionary.

C# Access Dictionary Elements

In c#, we have different ways to access dictionary elements i.e. either by using index positions or by iterating through the list using for / foreach loops.

Following is the example of accessing dictionary elements in different ways.

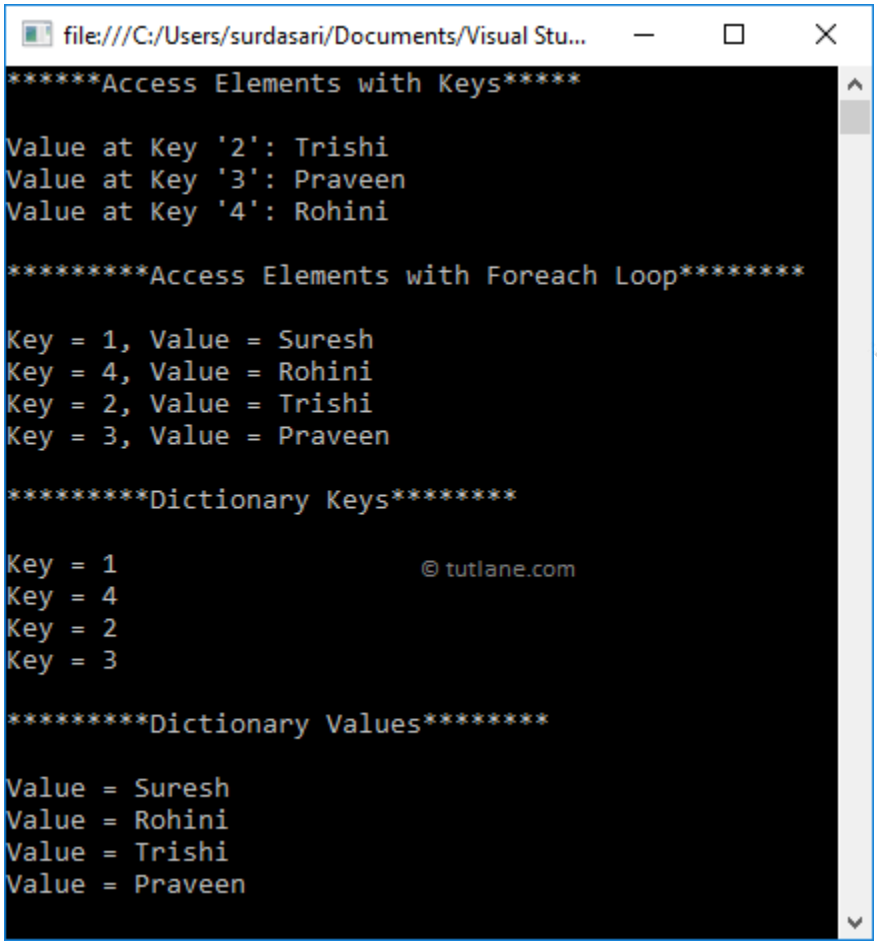
```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create a new dictionary
            Dictionary<int, string> dct = new Dictionary<int, string>()
;
            dct.Add(1, "Suresh");
            dct.Add(4, "Rohini");
            dct.Add(2, "Trishi");
            dct.Add(3, "Praveen");
            // Access value with key (not index)
            string val1 = dct[2];
            string val2 = dct[3];
```

```
            string val3 = dct[4];
            Console.WriteLine("******Access Elements with Keys*****");
            Console.WriteLine("Value at Key '2': " + val1);
            Console.WriteLine("Value at Key '3': " + val2);
            Console.WriteLine("Value at Key '4': " + val3);
            Console.WriteLine("*********Access Elements with Foreach Lo
op********");
            foreach (KeyValuePair<int, string> item in dct)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.WriteLine("*********Dictionary Keys********");
            foreach (var item in dct.Keys)
            {
                Console.WriteLine("Key = {0}", item);
            }
            Console.WriteLine("*********Dictionary Values********");
            foreach (var item in dct.Values)
            {
                Console.WriteLine("Value = {0}", item);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we are accessing dictionary elements in different ways by using keys and foreach loops based on our requirements.

When you execute the above c# program, you will get the result as shown below.

noteghar.com



C# Remove Elements from Dictionary

In c#, by using **Remove()** method we can delete a key/value pair from dictionary object.

Following is the example of deleting elements from the dictionary object in c# programming language.

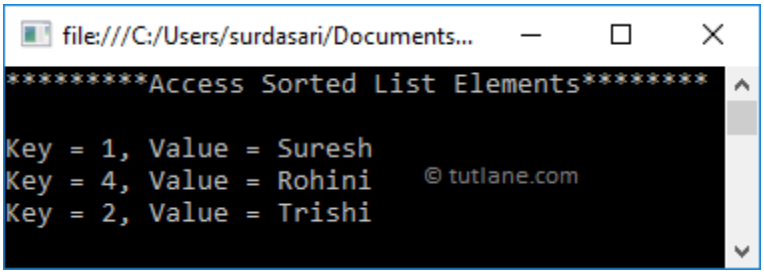
```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create a new dictionary
            Dictionary<int, string> dct = new Dictionary<int, string>()
;
            dct.Add(1, "Suresh");
            dct.Add(4, "Rohini");
            dct.Add(2, "Trishi");
```

noteghar.com
flyghar.com

```
            dct.Add(3, "Praveen");
            dct.Add(5, "Sateesh");
            // Remove element with key (not index)
            dct.Remove(3);
            dct.Remove(5);
            Console.WriteLine("*********Access Dictionary Elements*****
***");
            Console.WriteLine();
            foreach (KeyValuePair<int, string> item in dct)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, i
tem.Value);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **Remove()** method to delete a particular key of element from the dictionary.

When you execute the above c# program, we will get the result as shown below.



C# Dictionary Check If Item Exists

By using **ContainsKey()** and **ContainsValue()** methods, we can check whether the specified key / value element exists in dictionary or not. In case, if it exists these methods will return **true** otherwise **false**.

Following is the example of using **ContainsKey()** and **ContainsValue()** methods to check for an item that exists in a dictionary or not in c#.

```
using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            //Create a new dictionary
            Dictionary<int, string> dct = new Dictionary<int, string>()
;
            dct.Add(1, "Suresh");
            dct.Add(4, "Rohini");
            dct.Add(2, "Trishi");
            dct.Add(3, "Praveen");
            dct.Add(5, "Sateesh");
            Console.WriteLine("Contains Key 2: {0}", dct.ContainsKey(2)
);
            Console.WriteLine("Contains Value 'Tutlane': {0}", dct.Cont
ainsValue("Tutlane"));
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we used a **ContainsKey()** and **ContainsValue()** methods to check for a particular keys and values exists in dictionary (**dct**) or not.

When you execute the above c# program, you will get the result as shown below.

```
Contains Key 2: True
Contains Value 'Tutlane': False
```

C# Dictionary (Dictionary<TKey, TValue>)

Overview

The following are the important points that need to remember about the dictionary in c#.

- Dictionary is used to store a collection of key/value pairs that are organized by key.
- Dictionary will allow us to store duplicate values but keys must be unique to identify the values in the dictionary.
- In dictionary, the keys cannot be null but the values can be null.
- You can access dictionary elements either by using keys or with a foreach loop. In the foreach loop, you need to use **KeyValuePair<TKey, TValue>** to get a key/value pairs from the Dictionary object.

noteghar.com
flyghar.com

Difference between Dictionary and Hashtable in C#

The following are the main differences between hashtable and dictionary in c# programming language.

- Hashtable is a non-generic type of collection so it can store elements of different data types but Dictionary is a generic type of collection so it can store elements of the same data type.
- Hashtable is available with **System.Collections** namespace but the dictionary is available with **System.Collections.Generic** namespace.
- In the foreach loop, you need to use **DictionaryEntry** property to get the key/value pair from hashtable but we need to use **KeyValuePair** property to access key/value pair elements from dictionary.
- When compared with **dictionary** object, the hashtable will provide a lower performance because the hashtable elements are of object type so the boxing and unboxing process will occur when we store or retrieve values from the hashtable.
- In c#, the hashtable will throw an error if we try to find a key that does not exist but the dictionary object will return null in case the defined key not exists.

noteghar.com

C# HashSet (HashSet<T>)

In c#, **HashSet** is a generic type of collection and it is used to represent a set of unique values. The hashset in c# will allow storing only the strongly typed elements i.e. the values of the specified data type.

In c#, the hashset will provide high-performance set operations and the set will return only unique elements and those elements will not be in particular order. Even if we add duplicate elements in hashset, it will return only the unique elements in the result set.

The size of hashset object will vary dynamically so we can add or remove elements from the hashset based on our requirements.

In c#, the hashset is not useful when you want to work with duplicate elements. In that case, you need to consider using the list object in our application.

C# HashSet Declaration

In c#, hashset is a generic type of collection and it is provided by **System.Collections.Generic** namespace.

As discussed, the collection is a class so to define a hashset, you need to declare an instance of hashset class before we perform any operations such as add, delete, etc. like as shown below.

```
HashSet<T> hset = new HashSet<T>();
```

If you observe the above hashset declaration, we created a generic hashset (**hset**) with an instance of hashset class using type parameter (**T**) as placeholders with an angle (**<>**) brackets.

Here, the angle (**<>**) brackets will indicate that the hashset is a generic type and type parameter **T** is to represent a type of values to be accepted by hashset.

C# HashSet Initialization

Following is the example of initializing a generic hashset by specifying a required type to accept values.

```
HashSet<int> hset = new HashSet<int>();
```

If you observe the above example, we defined a hashset (**hset**) with a required value type to store. Here, the hashset object will store a value of **int** type.

C# HashSet Properties

The following are some of the commonly used properties of hashset object in c# programming language.

Property	Description
Count	It is used to get the number of elements in HashSet.

C# HashSet Methods

The following are some of the commonly used methods of generic hashset to perform add, search, insert, delete or sort operations in c# programming language.

Method	Description
Add	It is used to add elements of the specified type to hashset.
Clear	It will remove all the elements from hashset.
Contains	It is used to determine whether the specified element exists in hashset or not.
CopyTo	It is used to copy the elements of a hashset object to an array.
Remove	It is used to remove the specified element from hashset.
TryGetValue	It is used to search for the given value and return if value finds.

noteghar.com

C# HashSet Example

Following is the example of adding and accessing elements from hashset in c# programming language.

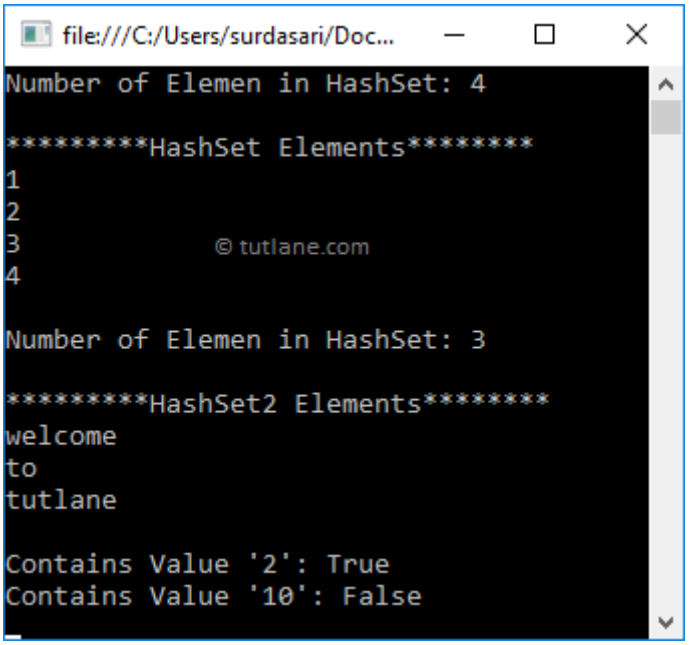
```
using System;
```

```
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create a new hashset
            HashSet<int> hset = new HashSet<int>();
            // Add elements to hashset object.
            hset.Add(1);
            hset.Add(2);
            hset.Add(2);
            hset.Add(3);
            hset.Add(3);
            hset.Add(4);
            Console.WriteLine("Number of Elemen in HashSet: {0}", hset.
Count);
            Console.WriteLine("*********HashSet Elements********");
            // Accessing elements from hashset.
            foreach (int item in hset)
            {
                Console.WriteLine(item);
            }
            // Creating and initializing hashset
            HashSet<string> hset2 = new HashSet<string> { "welcome", "t
o", "tutlane", "tutlane" };
            Console.WriteLine("Number of Elemen in HashSet: {0}", hset2
.Count);
            Console.WriteLine("*********HashSet2 Elements********");
            // Accessing elements.
            foreach (string item in hset2)
            {
                Console.WriteLine(item);
            }
            Console.WriteLine("Contains Value '2': {0}", hset.Contains(
2));
            Console.WriteLine("Contains Value '10': {0}", hset.Contains
(10));
            Console.ReadLine();
        }
    }
}
```

noteghar.com
flyghar.com

If you observe above example, we are able to define a new generic hashset (**hset**, **hset2**) collections by using **System.Collections.Generic** namespace. Here, we added a defined data type values to the newly created hashsets (**hset**, **hset2**) in different ways and used a foreach loop to access elements from hashset.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, we got only the unique elements from hashset and it returns the count of unique elements available in the hashset.

C# HashSet Overview

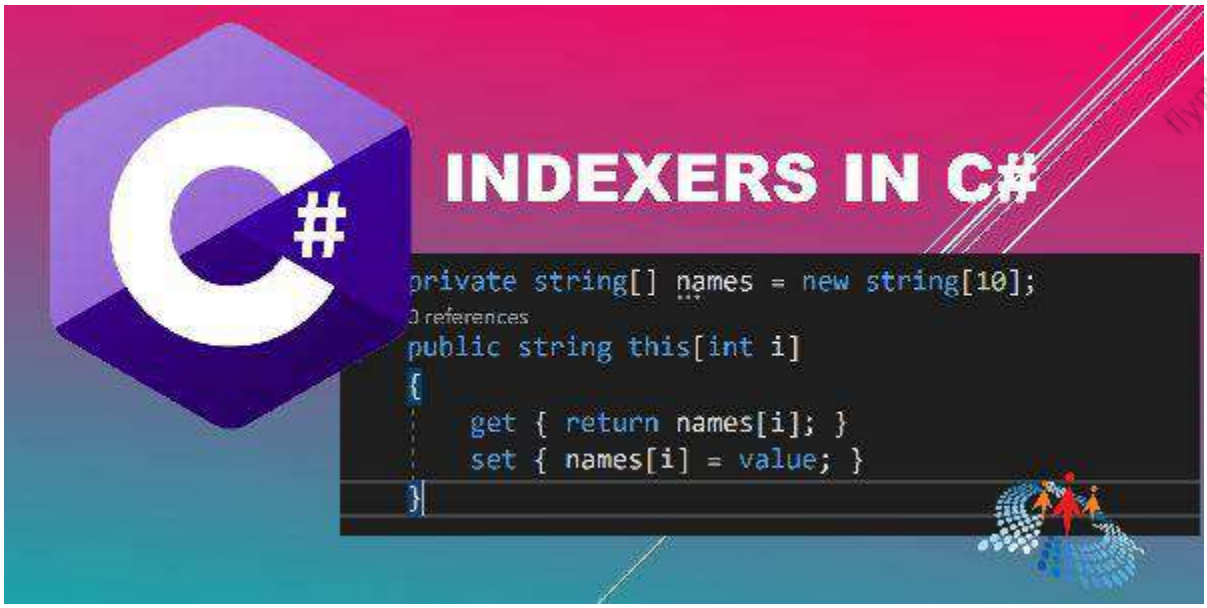
The following are the important points that need to remember about HashSet in c#.

- HashSet is a generic collection and it will available with **System.Collections.Generic** namespace.
- HashSet is useful to represent a set of values.
- HashSet will allow us to store duplicate values but it will return only unique values in result set.
- HashSet collection is not sorted and you can access HashSet values by using a foreach loop.

flyghar.com
noteghar.com

Indexer

C# indexers are usually known as smart arrays. A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array. In C#, indexers are created using this keyword. Indexers in C# are applicable on both classes and structs.



Defining an indexer allows you to create a class like that can allows its items to be accessed an array. Instances of that class can be accessed using the [] array access operator.

Creating an Indexer

```
1. <modifier> <return type> this [argument list]
2. {
3. get
4. {
5. // your get block code
6. }
7. set
8. {
9. // your set block code
10.      }
11.      }
```

In the above code:

<modifier>
can be private, public, protected or internal.

<return type>
can be any valid C# types.

this
this is a special keyword in C# to indicate the object of the current class.

[argument list]
The formal-argument-list specifies the parameters of the indexer.

Important points to remember on indexers:

- Indexers are always created with **this** keyword.
- Parameterized property are called indexer.
- Indexers are implemented through get and set accessors for the [ ] operator.
- ref and out parameter modifiers are not permitted in indexer.
- The formal parameter list of an indexer corresponds to that of a method and at least one parameter should be specified.
- Indexer is an instance member so can't be static but property can be static.
- Indexers are used on group of elements.
- Indexer is identified by its signature where as a property is identified it's name.
- Indexers are accessed using indexes where as properties are accessed by names.
- Indexer can be overloaded.

Indexer are defined in pretty much same way as properties, with get and set functions. The main difference is that the name of the indexer is the keyword this.

Following program demonstrates how to use an indexer.

```
1. using System;
2. namespace Indexer_example1
3. {
4.     class Program
5.     {
6.         class IndexerClass
7.         {
8.             private string[] names = new string[10];
9.             public string this[int i]
10.                 {
11.                     get
12.                     {
13.                         return names[i];
14.                     }
15.                     set
16.                     {
17.                         names[i] = value;
18.                     }
19.                 }
20.             }
21.             static void Main(string[] args)
22.             {
23.                 IndexerClass Team = new IndexerClass();
```

noteghar.com
flyghar.com

```
24.                     Team[0] = "Rocky";
25.                     Team[1] = "Teena";
26.                     Team[2] = "Ana";
27.                     Team[3] = "Victoria";
28.                     Team[4] = "Yani";
29.                     Team[5] = "Mary";
30.                     Team[6] = "Gomes";
31.                     Team[7] = "Arnold";
32.                     Team[8] = "Mike";
33.                     Team[9] = "Peter";
34.                     for (int i = 0; i < 10; i++)
35.                     {
36.                         Console.WriteLine(Team[i]);
37.                     }
38.                     Console.ReadKey();
39.                 }
40.             }
41.         }
```

flyghar.com

Difference between Indexers and Properties

Indexers	Properties
Indexers are created with this keyword.	Properties don't require this keyword.
Indexers are identified by signature.	Properties are identified by their names.
Indexers are accessed using indexes.	Properties are accessed by their names.
Indexer are instance member, so can't be static.	Properties can be static as well as instance members.
A get accessor of an indexer has the same formal parameter list as the indexer.	A get accessor of a property has no parameters.
A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter.	A set accessor of a property contains the implicit value parameter.

Indexers are commonly used for classes, which represents some data structure, an array, list, map and so on.

noteghar.com

Overloaded Indexers

Indexers can be overloaded. Indexers can also be declared with multiple parameters and each parameter may be a different type. It is not necessary that the indexes have to be integers. C# allows indexes to be of other types, for example, a string.

The following example demonstrates overloaded indexers −

Live Demo

tiyeghar.com

```
using System;

namespace IndexerApplication {
   class IndexedNames {
      private string[] namelist = new string[size];
      static public int size = 10;

      public IndexedNames() {
         for (int i = 0; i < size; i++) {
            namelist[i] = "N. A.";
         }
      }
      public string this[int index] {
         get {
            string tmp;

            if( index >= 0 && index <= size-1 ) {
               tmp = namelist[index];
            } else {
               tmp = "";
            }

            return ( tmp );
         }
         set {
            if( index >= 0 && index <= size-1 ) {
               namelist[index] = value;
            }
         }
      }

      public int this[string name] {
         get {
            int index = 0;

            while(index < size) {
               if (namelist[index] == name) {
                return index;
               }
               index++;
            }
            return index;
         }
```

noteghar.com

```
      }

      static void Main(string[] args) {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
         names[5] = "Sunil";
         names[6] = "Rubic";

         //using the first indexer with int parameter
         for (int i = 0; i < IndexedNames.size; i++) {
            Console.WriteLine(names[i]);
         }

         //using the second indexer with the string parameter
         Console.WriteLine(names["Nuha"]);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
2
```

noteghar.com
tlyghar.com

C# Inheritance

In c#, **Inheritance** is one of the primary concept of object-oriented programming (OOP) and it is used to inherit the properties from one class (base) to another (child) class.

The inheritance will enable us to create a new class by inheriting the properties from other classes to reuse, extend and modify the behavior of other class members based on our requirements.

In c# inheritance, the class whose members are inherited is called a **base** (**parent**) class and the class that inherits the members of **base** (**parent**) class is called a **derived** (**child**) class.

C# Inheritance Syntax

Following is the syntax of implementing an inheritance to define a derived class that inherits the properties of base class in c# programming language.

```
<access_modifier> class <base_class_name>
{
    // Base class Implementation
}

<access_modifier> class <derived_class_name> : <base_class_name>
{
    // Derived class implementation
}
```

If you observe the above syntax, we are inheriting the properties of **base** class into **child** class to improve code reusability.

Following is the simple example of implementing inheritance in c# programming language.

flyghar.com
noteghar.com

```
public class X
{
    public void GetDetails()
    {
        // Method implementation
    }
}
public class Y : X
{
    // your class implementation
}
class Program
{
    static void Main(string[] args)
    {
        Y y = new Y();
        y.GetDetails();
    }
}
```

flyghar.com

If you observe the above example, we defined a class "**X**" with the method called "**GetDetails**" and the class "**Y**" is inheriting from class "**X**". After that, we are calling a "**GetDetails**" method by using an instance of derived class "**Y**".

In c#, it's not possible to inherit the base class constructors in the derived class and the accessibility of other members of a base class also depends on the access modifiers which we used to define those members in a base class.

C# Inheritance Example

Following is the example of implementing an **inheritance** by defining two classes in c# programming language.

```
using System;

namespace Tutlane
{
    public class User
    {
        public string Name;
        private string Location;
        public User()
        {
            Console.WriteLine("Base Class Constructor");
        }
        public void GetUserInfo(string loc)
        {
```

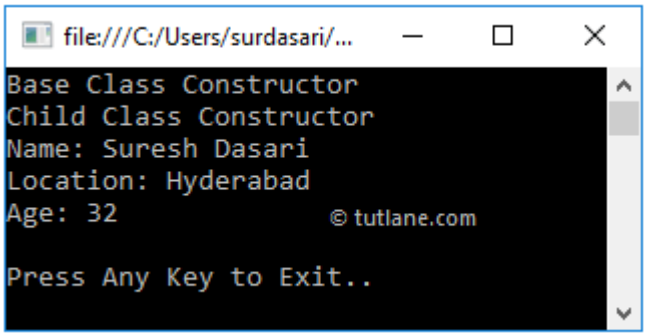
noteghar.com

```
            Location = loc;
            Console.WriteLine("Name: {0}", Name);
            Console.WriteLine("Location: {0}", Location);
        }
    }
    public class Details : User
    {
        public int Age;
        public Details()
        {
            Console.WriteLine("Child Class Constructor");
        }
        public void GetAge()
        {
            Console.WriteLine("Age: {0}", Age);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Details d = new Details();
            d.Name = "Suresh Dasari";
            // Compile Time Error
            //d.Location = "Hyderabad";
            d.Age = 32;
            d.GetUserInfo("Hyderabad");
            d.GetAge();
            Console.WriteLine("\nPress Any Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we defined a base class called "**User**" and inheriting all the properties of **User** class into a derived class called "**Details**" and we are accessing all the members of **User** class with an instance of **Details** class.

If we uncomment the commented code, we will get a compile-time error because the **Location** property in **User** class is defined with a private access modifier and the **private** members can be accessed only within the class.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, we are able to access all the properties of base class into child class based on our requirements.

C# Multi-Level Inheritance

Generally, c# supports only **single inheritance** that means a class can only inherit from one base class. However, in c# the inheritance is transitive and it allows you to define a hierarchical inheritance for a set of types and it is called a multi-level inheritance.

For example, suppose if class **C** is derived from class **B**, and class **B** is derived from class **A**, then class **C** inherits the members declared in both class **B** and class **A**.

```
public class A
{
// Implementation
}
public class B : A
{
// Implementation
}
public class C : B
{
// Implementation
}
```

If you observe above code snippet, class **C** is derived from class **B**, and class **B** is derived from class **A**, then class **C** inherits the members declared in both class **B** and class **A**. This is how we can implement **multi-level** inheritance in our applications.

Following is the example of implementing **multi-level inheritance** in c# programming language.

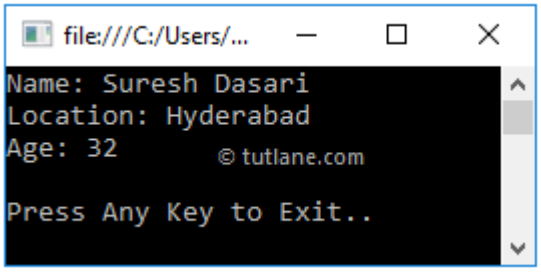
```
using System;

namespace Tutlane
{
    public class A
    {
        public string Name;
        public void GetName()
        {
            Console.WriteLine("Name: {0}", Name);
        }
    }
    public class B : A
    {
        public string Location;
        public void GetLocation()
        {
            Console.WriteLine("Location: {0}", Location);
        }
    }
    public class C : B
    {
        public int Age;
        public void GetAge()
        {
            Console.WriteLine("Age: {0}", Age);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            C c = new C();
            c.Name = "Suresh Dasari";
            c.Location = "Hyderabad";
            c.Age = 32;
            c.GetName();
            c.GetLocation();
            c.GetAge();
            Console.WriteLine("\nPress Any Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

noteghar.com
tlyghar.com

If you observe above example, we implemented a three classes (**A**, **B**, **C**) and the class **C** is derived from class **B**, and class **B** is derived from class **A**.

By implementing a multi-level inheritance, class **C** can inherit the members declared in both class **B** and class **A**.

When you execute the above c# program, you will get the result as shown below.



tlyghar.com

C# Multiple Inheritance

As discussed, c# supports only **single inheritance** that means a class can only inherit from one base class. In case, if we try to inherit a class from multiple base classes, then we will get compile-time errors.

For example, if class **C** is trying to inherit from Class **A** and **B** at the same time, then we will get a compile-time error because multiple inheritance is not allowed in c#.

```
public class A
{
// Implementation
}
public class B
{
// Implementation
}
public class C : A, B
{
// Implementation
}
```

noteghar.com

If you observe above code snippet, class **C** is trying to inherit properties from both classes **A** and **B** at the same time which will lead to compile-time error like "**Class C cannot have multiple classes: A and B**".

As discussed, **multi-level** inheritance is supported in c# but **multiple** inheritance is not supported. In case, if you want to

implement **multiple** inheritance in c#, that we can achieve by using **interfaces**.

Following is the example of implementing **multiple inheritance** using interfaces in c# programming language.

```
1. public interface IA //ineterface  1
2. {
3.     string setImgs(string a);
4. }
5. public interface IB  //Interface 2
6. {
7.     int getAmount(int Amt);
8. }
9. public class Car : IA, IB //implementatin
10.     {
11.         public int getAmount(int Amt)
12.         {
13.             return 100;
14.         }
15.         public string setImgs(string a)
16.         {
17.             return "this is the car";
18.         }
19.     }
```

In the preceding program the Car class inherits the features of the two interfaces hence this type of inheritance is called Multiple Inheritance.

The following are some key points about inheritance:

1. C# does not support multiple inheritances of classes, the same thing can be done using interfaces.
2. Private members are not accessed in a derived class when one class is derived from another.

flyghar.com
noteghar.com

C# Base Keyword

In c#, the **base** keyword is useful to access base class members such as properties, methods, etc. in the derived class.

By using **base** keyword, we can call a base class method that has been overridden by another method in the derived class and we can specify which base class constructor should be called while creating an instance of the derived class.

In c#, the base keyword is permitted to use only within an instance method, constructor or in an instance property accessor and it will throw an error, in case if we try to use **base** keyword within a static method.

flyghar.com

C# Base Keyword Example

Following is the example of using a **base** keyword to access a base class properties and the method that has been overridden by derived class in c# programming language.

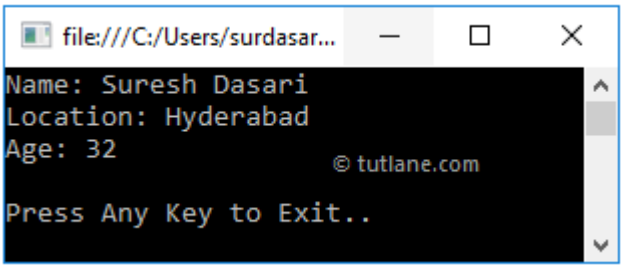
```
using System;

namespace Tutlane
{
    // Base Class
    public class Users
    {
        public string name = "Suresh Dasari";
        public string location = "Hyderabad";
        public int age = 32;
        public virtual void GetInfo()
        {
            Console.WriteLine("Name: {0}", name);
            Console.WriteLine("Location: {0}", location);
        }
    }
    // Derived Class
    public class Details : Users
    {
        public override void GetInfo()
        {
            base.GetInfo();
            Console.WriteLine("Age: {0}", base.age);
        }
    }
    class Program
    {
```

noteghar.com

```
        static void Main(string[] args)
        {
            Details d = new Details();
            d.GetInfo();
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we are able to call a base class (**Users**) method (**GetInfo**) in derived class overridden method using **base** keyword and we are able to access properties of base class (**Users**) in derived class (**Details**) using **base** keyword.

When you execute the above c# program, you will get the result as shown below.



By using **base** keyword, we can directly access a base class fields in derived class but the base class methods can be accessed only in overridden methods of the derived class using **base** keyword.

C# Call Base Class Constructor from Derived Class

As discussed, we can call a base class constructor from the derived class using the **base** keyword.

Following is the example of specifying which base class constructor should be called while creating an instance of the derived class.

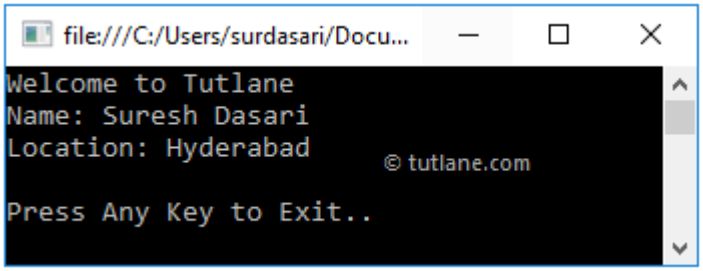
```
using System;

namespace Tutlane
{
    // Base Class
    public class BClass
    {
        public BClass()
        {
```

```
            Console.WriteLine("Welcome to Tutlane");
        }
        public BClass(string a, string b)
        {
            Console.WriteLine("Name: {0}", a);
            Console.WriteLine("Location: {0}", b);
        }
    }
    // Derived Class
    public class DClass : BClass
    {
        // This constructor will call default constructor
        public DClass(): base()
        {
        }

        // This constructor will call parameterized constructor
        public DClass(string x, string y): base(x, y)
        {
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            DClass d = new DClass();
            DClass d1 = new DClass("Suresh Dasari", "Hyderabad");
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

tlyghar.com

If you observe above example, we are calling base class (**BClass**) constructors in derived class (**DClass**) using **base** keyword.

When you execute the above c# program, you will get the result as shown below.



noteghar.com

This is how you can use base keyword in c# to access base class members such as methods, properties, etc. in derived class based on our requirements.

Method Overriding

In c#, **Method Overriding** means override a base class method in the derived class by creating a method with the same name and signatures to perform a different task. The Method Overriding in c# can be achieved by using **override** & **virtual** keywords along with the inheritance principle.

Suppose, if we want to change the behavior of the base class method in a derived class, then we need to use **method overriding**. The base class method which we want to override in the derived class that needs to be defined with **virtual** keyword and we need to use **override** keyword in derived class while defining the method with the same name and parameters then only we can override the base class method in a derived class.

In c#, the Method Overriding is also called as **run time polymorphism** or **late binding**. Following is the code snippet of implementing a **method overriding** in c# programming language.

```
// Base Class
public class Users
{
    public virtual void GetInfo()
    {
        Console.WriteLine("Base Class");
    }
}
// Derived Class
public class Details : Users
{
    public override void GetInfo()
    {
        Console.WriteLine("Derived Class");
    }
}
```

If you observe above code snippet, we created a two classes ("**Users**", "**Details**") and the derived class (Details) is inheriting the properties from base class (**Users**) and we are overriding the base class method **GetInfo** in derived class by creating a method with same name and parameters, this is called a **method overriding** in c#.

Here, we defined a **GetInfo** method with a **virtual** keyword in the base class to allow derived class to override that method using the **override** keyword.

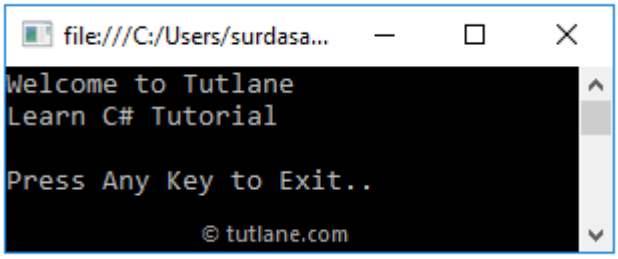
As discussed, only the methods with a **virtual** keyword in the base class are allowed to override in derived class using **override** keyword.

noteghar.com
flyghar.com

Following is the example of implementing a method overriding in c# programming language.

```
using System;

namespace Tutlane
{
    // Base Class
    public class BClass
    {
        public virtual void GetInfo()
        {
            Console.WriteLine("Learn C# Tutorial");
        }
    }
    // Derived Class
    public class DClass : BClass
    {
        public override void GetInfo()
        {
            Console.WriteLine("Welcome to Tutlane");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            DClass d = new DClass();
            d.GetInfo();
            BClass b = new BClass();
            b.GetInfo();
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

When we execute above c# program, we will get the result like as shown below.



This is how we can achieve method overriding in c# to override a base class method in child class by defining a method with the same name and signatures based on our requirements.

Method Overloading

In c#, **Method Overloading** means defining multiple methods with the same name but with different parameters. By using **Method Overloading**, we can perform different tasks with the same method name by passing different parameters.

Suppose, if we want to overload a method in c#, then we need to define another method with the same name but with different signatures. In c#, the Method Overloading is also called as **compile time polymorphism** or **early binding**.

Following is the code snippet of implementing a method overloading in c# programming language.

```
public class Calculate
{
    public void AddNumbers(int a, int b)
    {
        Console.WriteLine("a + b = {0}", a + b);
    }
    public void AddNumbers(int a, int b, int c)
    {
        Console.WriteLine("a + b + c = {0}", a + b + c);
    }
}
```

If you observe above "**Calculate**" class, we defined a two methods with same name (**AddNumbers**) but with different input parameters to achieve **method overloading** in c#.

Following is the example of implementing a method overloading in c# programming language.

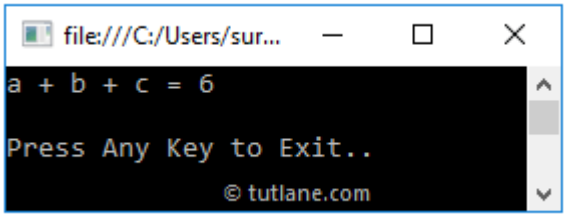
```
using System;

namespace Tutlane
{
    public class Calculate
    {
        public void AddNumbers(int a, int b)
        {
            Console.WriteLine("a + b = {0}", a + b);
        }
        public void AddNumbers(int a, int b, int c)
        {
            Console.WriteLine("a + b + c = {0}", a + b + c);
        }
    }
```

noteghar.com

```
    class Program
    {
        static void Main(string[] args)
        {
            Calculate c = new Calculate();
            c.AddNumbers(1, 2);
            c.AddNumbers(1, 2, 3);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

tiyeghar.com

When you execute above c# program, you will get the result like as shown below.



This is how we can implement method overloading in c# by defining multiple methods with the same name but with different signatures based on our requirements.

noteghar.com

Method Hiding

C# also provides a concept to hide the methods of the base class from derived class, this concept is known as Method Hiding. It is also known as **Method Shadowing**. In method hiding, you can hide the implementation of the methods of a base class from the derived class using the *new* keyword. Or in other words, in method hiding, you can redefine the method of the base class in the derived class by using the *new* keyword.

tiyeghar.com

Example:

```
// C# program to illustrate the
// concept of method hiding
using System;

// Base Class
public class My_Family {

    public void member()
    {
        Console.WriteLine("Total number of family members:
3");
    }
}

// Derived Class
public class My_Member : My_Family {

    // Reimplement the method of the base class
    // Using new keyword
    // It hides the method of the base class
    public new void member()
    {
        Console.WriteLine("Name: Rakesh, Age: 40 \nName:
Somya, "+
                                "Age: 39 \nName: Rohan, Age: 20
");
    }
}

// Driver Class
class GFG {

    // Main method
    static public void Main()
    {

        // Creating the object of the derived class
        My_Member obj = new My_Member();
```

noteghar.com

```
        // Access the method of derived class
        obj.member();
    }
}
```

Output:

```
Name: Rakesh, Age: 40
Name: Somya, Age: 39
Name: Rohan, Age: 20
```

Explanation: In the above example, *My_Family* is the base class and *My_Member* is a derived class. In both the classes we have the same name method, i.e. *member()* method. But in the derived class, the *member()* method is declared with the new keyword. When this method is called, it prints the name and the age of the family members not the total number of family members. Which means when we call the *member()* method with the help of the derived class object, it hides the same name method present in the base class due to the presence of the new keyword.

How to call a hidden method?

In method hiding, you can also call the hidden method of the base class in the derived class using **three** different ways and the ways are:

- ***By using the base keyword*** you can call the hidden method of the base class in your derived class shown in the below example:

  Example:

```
// C# program to illustrate how
// to access hidden method
using System;

// Base Class
public class My_Family {
   public void member()
   {
      Console.WriteLine("Total number of family members: 3");
   }
}
```

```
// Derived Class
public class My_Member : My_Family {


    public new void member()
    {

        // Calling the hidden method of the
        // base class in a derived class
        // Using base keyword
        base.member();
        Console.WriteLine("Name: Rakesh, Age: 40 \nName: Somya,"+
                          " Age: 39 \nName: Rohan, Age: 20");
    }
}

class GFG {

    // Main method
    static public void Main()
    {

        // Creating the object of the derived class
        My_Member obj = new My_Member();

        // Access the method of derived class
        obj.member();
    }
}
```

Output:

```
Total number of family members: 3
Name: Rakesh, Age: 40
Name: Somya, Age: 39
Name: Rohan, Age: 20
```

- ***By casting the derived class type to base class type*** you can invoke the hidden method. As shown in the below example. We know that in inheritance the derived class has all the capabilities of the base class so we can easily cast the object of a derived class into base class type.

Example:

```
// C# program to illustrate how
```

```
// to access hidden method
using System;

// Base Class
public class My_Family {

    public void member()
    {
        Console.WriteLine("Total number of family members: 2");
    }
}

// Derived Class
public class My_Member : My_Family {

    public new void member() {

        Console.WriteLine("Name: Rakesh, Age: 40 "+
                    "\nName: Somya, Age: 39");
    }
}

class GFG {

    // Main method
    static public void Main()
    {

        // Creating the object of the derived class
        My_Member obj = new My_Member();

        // Invoking the hidden method
        // By type casting
        ((My_Family)obj).member();
    }
}
```

flyghar.com

Output:

```
Total number of family members: 2
```

noteghar.com

- Instead of using derived class reference variable we use the parent class reference variable for calling the hidden method. It is similar to the above way of calling a hidden method. Here we also type case the object of the derived class in a different syntax.

  Note: If you try to invoke a hidden method using below syntax,

  ```
  My_Member obj = new My_Family();
  ```

Here, the compiler will give an error because the object of *My_Family* class cannot fulfill the duties of *My_Member* class.

Example:

```
// C# program to illustrate how
// to access hidden method
using System;

// Base Class
public class My_Family {

    public void member()
    {
        Console.WriteLine("Total number of family members: 2");
    }
}

// Derived Class
public class My_Member : My_Family {

    public new void member() {

        Console.WriteLine("Name: Rakesh, Age: 40 "+
                    "\nName: Somya, Age: 39");
    }
}

class GFG {

    // Main method
    static public void Main()
    {

        // Invoking the hidden method
        My_Family obj = new My_Member();
        obj.member();
    }
}
```

flyghar.com
noteghar.com

Output:

```
Total number of family members: 2
```

Method Overriding Vs Method Hiding

METHOD OVERRIDING	METHOD HIDING
In method overriding, you need to define the method of a parent class as a virtual method using virtual keyword and the method of child class as an overridden method using override keyword.	In method hiding, you just simply create a method in a parent class and in child class you need to define that method using new keyword.
It only redefines the implementation of the method.	In method hiding, you can completely redefine the method.
Here overriding is an object type.	Here hiding is a reference type.
If you do not use override keyword, then the compiler will not override the method. Instead of the overriding compiler will hide the method.	If you do not use the new keyword, then the compiler will automatically hide the method of the base class.
In method overriding, when base class reference variable pointing to the object of the derived class, then it will call the overridden method in the derived class.	In the method hiding, when base class reference variable pointing to the object of the derived class, then it will call the hidden method in the base class.

noteghar.com
flyghar.com

Interface

In c#, the **interface** is same as a class but the only difference is class can contain both declarations and implementation of methods, properties and events but **interface** will contain only the declarations of methods, properties, and events that a class or struct can implement.

An **interface** in c# is more like a contract and the class or struct that implements an interface must provide an implementation for all the members that are specified in the interface definition.

Generally, c# will not support multiple inheritance of classes but that can be achieved by using an interface. In addition, a structure in c# cannot be inherited from another structure or class but that can be inherited by using interfaces.

In c#, we can define an interface by using interface keyword. Following is the example of defining an interface using interface keyword.

```
interface IUser
{
    void InsertDetails();
}
```

If you observe the above code snippet, we defined an interface (**IUser**) using interface keyword with the **InsertDetails** method signature. Now, the **IUser** interface can be implemented by any class or struct by providing a definition for the **InsertDetails** method.

To implement an interface in a class or structure the syntax will be like class ClassName : Interface Name. Following is the example of implementing an interface in a class.

```
class User : IUser
{
    void InserDetails()
    {
        // Method Implementation
    }
}
```

If you observe the code snippet, we inherited an interface (**IUser**) in a class (**User**) and implemented a defined interface method in a class.

In c#, an interface cannot be instantiated directly, but it can be instantiated by a class or struct that implements an interface. Following is the example of creating an instance for the interface in c# programming language.

```
IUser u = new User();
```

In c#, a class can inherit only from one class but we can implement multiple interfaces in a class or struct by using interfaces.

By default, the members of an interface are public and we are not allowed to include any other access modifiers. In c#, an interface can contain methods, properties, events, indexers but it can't contain constants, fields, operators, instance constructors, finalizers or types.

Following is the example of creating and implementing an instance using a class in c# programming language.

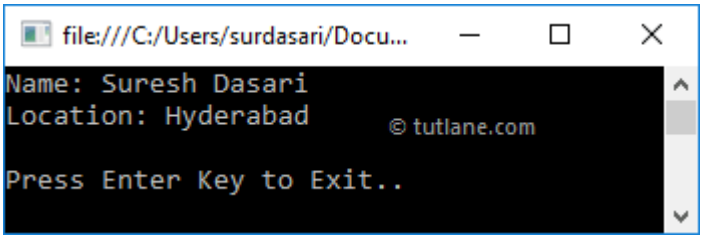
```
using System;

namespace Tutlane
{
   interface IUser
   {
      void GetDetails(string x);
   }
   class User : IUser
   {
      public void GetDetails(string a)
      {
         Console.WriteLine("Name: {0}", a);
      }
   }
   class User1 : IUser
   {
      public void GetDetails(string a)
      {
         Console.WriteLine("Location: {0}", a);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         IUser u = new User();
         u.GetDetails("Suresh Dasari");
```

noteghar.com
tlyghar.com

```
            IUser u1 = new User1();
            u1.GetDetails("Hyderabad");
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created an interface **IUser** and the two classes "**User & User1**" implemented an interface **IUser** by providing an implementation for **GetDetails()** method and we created an instance for interface "**IUser**" using **User & User1** classes.

When we execute the above c# program, we will get the result as shown below.



Explicit Implementation

An interface can be implemented explicitly using `<InterfaceName>.<MemberName>`. Explicit implementation is useful when class is implementing multiple interfaces; thereby, it is more readable and eliminates the confusion. It is also useful if interfaces have the same method name coincidently.

> **Note:**
>
> Do not use *public* modifier with an explicit implementation. It will give a compile-time error.

```
interface IFile
{
    void ReadFile();
    void WriteFile(string text);
}

class FileInfo : IFile
{
```

```
    void IFile.ReadFile()
    {
        Console.WriteLine("Reading File");
    }

    void IFile.WriteFile(string text)
    {
        Console.WriteLine("Writing to file");
    }
}
```

flyghar.com

When you implement an interface explicitly, you can access interface members only through the instance of an interface type.

```
interface IFile
{
    void ReadFile();
    void WriteFile(string text);
}

class FileInfo : IFile
{
    void IFile.ReadFile()
    {
        Console.WriteLine("Reading File");
    }

    void IFile.WriteFile(string text)
    {
        Console.WriteLine("Writing to file");
    }

    public void Search(string text)
    {
        Console.WriteLine("Searching in file");
    }
}

public class Program
{
    public static void Main()
    {
        IFile file1 = new FileInfo();
        FileInfo file2 = new FileInfo();

        file1.ReadFile();
        file1.WriteFile("content");
```

noteghar.com

```
        //file1.Search("text to be searched")//compile-time error

        file2.Search("text to be searched");
        //file2.ReadFile(); //compile-time error
        //file2.WriteFile("content"); //compile-time error
    }
}
```

flyghar.com

In the above example, file1 object can only access members of IFile, and file2 can only access members of FileInfo class. This is the limitation of explicit implementation.

Implementing Multiple Interfaces

As discussed, c# will not support multiple inheritance of classes but that can be achieved by using the interface.

Following is the example of implementing a multiple inheritance using interfaces in c# programming language.

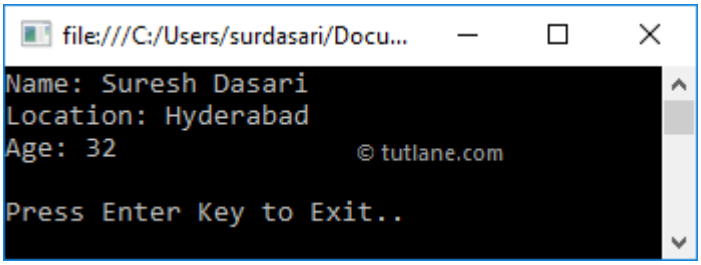
```
using System;

namespace Tutlane
{
   interface IName
   {
      void GetName(string x);
   }
   interface ILocation
   {
      void GetLocation(string x);
   }
   interface IAge
   {
      void GetAge(int x);
   }
   class User : IName, ILocation, IAge
   {
      public void GetName(string a)
      {
         Console.WriteLine("Name: {0}", a);
      }
      public void GetLocation(string a)
      {
         Console.WriteLine("Location: {0}", a);
      }
```

noteghar.com

```
        public void GetAge(int a)
        {
            Console.WriteLine("Age: {0}", a);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User();
            u.GetName("Suresh Dasari");
            u.GetLocation("Hyderabad");
            u.GetAge(32);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe example, we created a multiple interfaces and implementing those interfaces using User class to achieve multiple inheritance.

When you execute the above c# program, you will get the result as shown below.



This is how you can achieve multiple inheritance using the interface in the c# programming language.

Interface Overview

The following are the important properties of the interface in the c# programming language.

- In c#, the interface is like an abstract class and it can contain only declarations of members such as methods, properties, indexers and events.
- By default, the members of an interface are public and we are not allowed to include any other access modifiers.
- In c#, an interface cannot be instantiated directly, but it can be instantiated by a class or struct that implements an interface.
- The class or struct that implements an interface must provide an implementation for all the members that are specified in the interface definition.
- The class or struct can implement multiple interfaces.

flyghar.com

Difference between Abstract Class and Interface

Abstract Class	Interface
In c#, the abstract classes cannot be instantiated.	In c#, an interface cannot be instantiated directly, but it can be instantiated by a class or struct that implements an interface.
An abstract class can contain both declarations and implementations of methods, properties, etc.	An interface can contain only declarations of methods, properties, indexers and events.
The members of the abstract class can contain different access modifiers.	By default, all the members of an interface are public and we are not allowed to include any other access modifiers.
A class can inherit only one abstract class.	A class can inherit multiple interfaces.
A class that is derived from the abstract class must implement all inherited abstract methods and accessors.	The class or struct that implements an interface must provide an implementation for all the members that are specified in the interface definition.

noteghar.com

Partial Class

In c#, a **partial class** is useful to split the functionality of a particular class into multiple class files and all these files will be combined into one single class file when the application is compiled.

While working on large scale projects, multiple developers want to work on the same class file at the same time. To solve this problem, c# provides an ability to spread the functionality of a particular class into multiple class files using partial keyword.

In c#, we can use partial keyword to split the definition of a particular class, structure, **interface** or a method over two or more source files.

Following is the example of splitting the definition of **User** class into two class files, **User1.cs** and **User2.cs**.

User1.cs

```
public partial class User
{
   private string name;
   private string location;
   public User(string a, string b)
   {
      this.name = a;
      this.location = b;
   }
}
```

If you observe above code, we created a partial class called **User** in **User1.cs** class file using partial keyword with required variables and constructor.

User2.cs

```
public partial class User
{
   public void GetUserDetails()
   {
      Console.WriteLine("Name: " + name);
      Console.WriteLine("Location: " + location);
   }
}
```

If you observe above code, we created a partial class called **User** in **User2.cs** class file using partial keyword with **GetUserDetails()** method.

When you execute the above code, the compiler will combine these two partial classes into one **User** class like as shown below.

tiyeghar.com

User

```
public class User
{
   private string name;
   private string location;
   public User(string a, string b)
   {
      this.name = a;
      this.location = b;
   }
   public void GetUserDetails()
   {
      Console.WriteLine("Name: " + name);
      Console.WriteLine("Location: " + location);
   }
}
```

This is how the compiler will combine all the partial classes into single class while executing the application in c# programming langauge.

Rules to Implement Partial Class

In c#, we need to follow certain rules to implement a partial class in our applications.

noteghar.com

- To split the functionality of class, structure, **interface** or a method over multiple files we need to use partial keyword and all files must be available at compile time to form the final type.
- The partial modifier can only appear immediately before the keywords class, struct or **interface**.
- All parts of partial type definitions must be in the same namespace or assembly.
- All parts of partial type definitions must have the same accessibility, such as **public**, **private**, etc.

- If any partial part is declared as **abstract**, **sealed** or **base**, then the whole type is considered as **abstract** or **sealed** or **base** based on the defined type.
- In c#, different parts can have different **base** types but the final type will inherit all the base types.
- Nested partial types are allowed in partial type definitions.

tlyghar.com

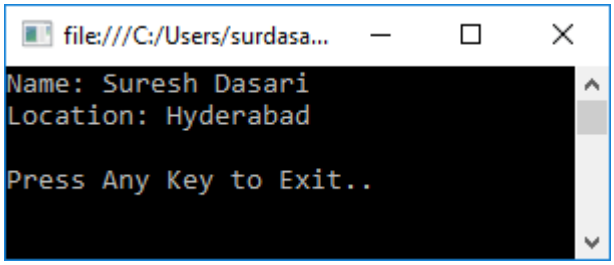
Following is the example of defining partial classes using partial keyword in c# programming language.

```
using System;

namespace Tutlane
{
   public partial class User
   {
      private string name;
      private string location;
      public User(string a, string b)
      {
         this.name = a;
         this.location = b;
      }
   }
   public partial class User
   {
      public void GetUserDetails()
      {
         Console.WriteLine("Name: " + name);
         Console.WriteLine("Location: " + location);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         User u = new User("Suresh Dasari", "Hyderabad");
         u.GetUserDetails();
         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

noteghar.com

If you observe above example we created a partial class **User** using partial keyword and we are able access all partial classes as single class to perform required operations.

When you execute the above c# program, we will get the result as shown below.



This is how you can split the functionality of class, structure, **interface** or a method over two or more source files using partial modifier based on our requirements.

noteghar.com
noteghar.com

Partial Method

In c#, the **partial method** is a special type of method which will contain a declaration part in one partial class and the definition part in another partial class or in the same partial class.

Generally, the partial methods will exist either in partial class or struct and it will contain two main parts one is **declaration** and **definition**. Here, the **declaration** part represents a signature of the partial method and the **definition** part will represent an implementation of the partial method.

In the partial method, the definition part is optional so it may or may not have an implementation in partial classes. In case, the implementation part of the partial method is not available in any partial class, then the compiler will remove the definition of the method and all the calls related to that particular method in the final class.

Following is the example of implementing a partial method using partial keyword in two class files, **User1.cs** and **User2.cs**.

User1.cs

```
public partial class User
{
  // Partial Method Declaration
    partial void GetUserDetails(string a, string b);
    public void TestMethod()
    {
        Console.WriteLine("Test");
    }
}
```

If you observe above code, we declared a partial method called **GetUserDetails** in **User1.cs** class file using partial keyword along with normal method called **TestMethod()**.

User2.cs

```
public partial class User
{
    public User(string a, string b)
    {
        GetUserDetails(a, b);
    }
    // Partial Method Implementation
    partial void GetUserDetails(string x, string y)
    {
```

```
        Console.WriteLine("Name: " + x);
        Console.WriteLine("Location: " + y);
    }
}
```

If you observe above code, we implemented a partial method called **GetUserDetails** in **User1.cs** class file using partial keyword with required variables and constructor.

When you execute the above code, the compiler will combine these two partial classes into one **User** class like as shown below.

tiyeghar.com

User

```
public class User
{
    public User(string a, string b)
    {
        GetUserDetails(a, b);
    }
    public void TestMethod()
    {
        Console.WriteLine("Test");
    }
    private void GetUserDetails(string x, string y)
    {
        Console.WriteLine("Name: " + x);
        Console.WriteLine("Location: " + y);
    }
}
```

This is how the compiler will combine all the partial classes into single class while executing the application in c# programming langauge.

Rules to Implement Partial Method

In c#, we need to follow certain rules to implement the partial method in our applications.

noteghar.com

- In c#, the partial method can be implemented within partial class or structure only and the signatures in both parts of the partial type must be the same.
- The partial method declaration must begin with partial keyword and method must return **void**.
- By default, partial methods are implicitly **private** so no **access modifiers** are allowed and they cannot be **virtual**.

- Partial methods can have ref but not out parameters.
- Partial methods can be static and **generic**.

Partial Method Example

Following is the example of defining a partial method in partial class using partial modifier in c# programming language.

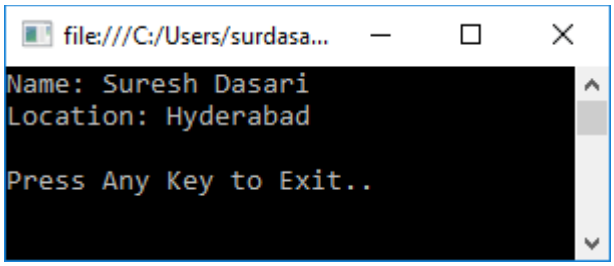
tiyghar.com

```
using System;

namespace Tutlane
{
    public partial class User
    {
        // Partial Method Declaration
        partial void GetUserDetails(string a, string b);
    }

    public partial class User
    {
        public User(string a, string b)
        {
            GetUserDetails(a, b);
        }
        // Partial Method Implementation
        partial void GetUserDetails(string x, string y)
        {
            Console.WriteLine("Name: " + x);
            Console.WriteLine("Location: " + y);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            User u = new User("Suresh Dasari", "Hyderabad");
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

noteghar.com

If you observe above example we created a partial method **GetUserDetails** in two partial classes using partial keyword to perform required operations.

When you execute the above c# program, we will get the result as shown below.



This is how we can use partial methods in partial class or structure when we have a method declaration and implementation in multiple files with partial modifier based on our requirements.

noteghar.com
noteghar.com

C# Properties (GET, SET)

In c#, **Property** is an extension of the class variable. It provides a mechanism to read, write, or change the class variable's value without affecting the external way of accessing it in our applications.

In c#, properties can contain one or two code blocks called **accessors,** and those are called a get accessor and set accessor. By using get and set accessors, we can change the internal implementation of class variables and expose it without affecting the external way of accessing it based on our requirements.

Generally, in object-oriented programming languages like c# you need to define fields as **private** and then use properties to access their values in a **public** way with get and set accessors.

Following is the syntax of defining a **property** with get and set accessor in c# programming language.

```
<access_modifier> <return_type> <property_name>
{
   get
   {
     // return property value
   }
   set
   {
    // set a new value
   }
}
```

If you observe the above syntax, we used an access modifier and **return type** to define a property along with get and set accessors to make required modifications to the class variables based on our requirements.

Here, the get accessor code block will be executed whenever the property is read, and the code block of set accessor will be executed whenever the property is assigned to a new value.

In c#, the properties are categorized into three types, those are.

Type	Description
Read-Write	A property that contains a both get and set accessors, then we will call it a read-write property.
Read-Only	A property that contains only get accessor, then we will call it a read-only property.
Write-Only	A property that contains only set accessor, then we will call it a write-only property.

In c#, Properties won't accept any parameters, and we should not pass a property as a ref or out parameter in our application.

Following is the simple example of defining a private variable and a property in the c# programming language.

```
class User
{
   private string name;
   public string Name
   {
      get { return name; }
      set { name = value; }
   }
}
```

noteghar.com
flyghar.com

If you observe the above example, we defined a property called "**Name**" and we used a get accessor to return a property value and set accessors to set a new value. Here, the **value** keyword in set accessor is used to define a value that is being assigned by set accessor.

In c#, the get accessor needs to be used only to return the field value or to compute it and return it, but we should not use it for changing the state of an object.

As discussed, we can extend the behavior of class variables using properties get and set accessors. Following is the example of extending the behavior of **private** variable in property using get and set accessors in c# programming language.

```
class User
{
   private string name = "Suresh Dasari";
   public string Name
   {
     get
     {
        return name.ToUpper();
     }
     set
     {
        if (value == "Suresh")
          name = value;
     }
   }
}
```

If you observe the above example, we are extending the behavior of **private** variable **name** using a property called **Name** with get and set accessors by performing some validations like to make sure **Name** value equals to only "**Suresh**" using set accessor and converting property text to uppercase with get accessor.

Here the field "**name**" is marked as **private,** so if you want to make any changes to this field, we can do it only by calling the property (**Name**).

In c# properties, the get accessor will be invoked while reading the value of a property, and when we assign a new value to the property, then the set accessor will be invoked by using an argument that provides the new value.

Following is the example of invoking get and set accessors of properties in c# programming language.

```
User u = new User();
u.Name = "Rohini"; // set accessor will invoke
Console.WriteLine(u.Name); // get accessor will invoke
```

In the above example, when we assign a new value to the property, then the set accessor will be invoked and the get accessor will be invoked when we try to read the value from the property.

C# Properties (Get, Set) Example

Following is the example of
defining **properties** with get and set accessors to implement required validations without affecting the external way of using it in the c# programming language.

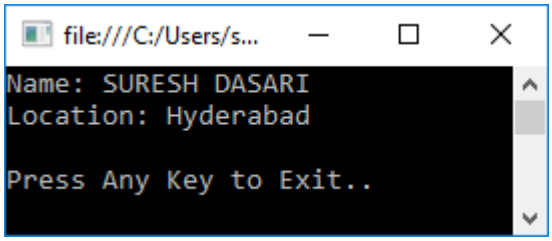
```
using System;

namespace Tutlane
{
    class User
    {
      private string location;
      private string name = "Suresh Dasari";
      public string Location
      {
```

```
        get { return location; }
        set { location = value; }
    }
    public string Name
    {
      get
      {
         return name.ToUpper();
      }
      set
      {
         if (value == "Suresh")
           name = value;
      }
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
       User u = new User();
       // set accessor will invoke
       u.Name = "Rohini";
       // set accessor will invoke
       u.Location = "Hyderabad";
       // get accessor will invoke
       Console.WriteLine("Name: " + u.Name);
       // get accessor will invoke
       Console.WriteLine("Location: " + u.Location);
       Console.WriteLine("\nPress Enter Key to Exit..");
       Console.ReadLine();
    }
  }
}
```

If you observe the above example, we are extending the behavior of private variables (**name**, **location**) using properties (**Name**, **Location**) with get and set accessors by performing some validations like to

make sure **Name** value equals to only "**Suresh**" using set accessor and converting property text to uppercase with get accessor.

When you execute the above c# program, we will get the result as shown below.



If you observe the above example, our variable text converted to upper case, and even after we set variable text as "**Rohini**", it displayed text as "**Suresh Dasari**" because of the set accessor validation fails in the property.

C# Create Read-Only Properties

As discussed, if a property contains the only get accessor, then we will call it a read-only property. Following is the example of creating read-only properties in the c# programming language.

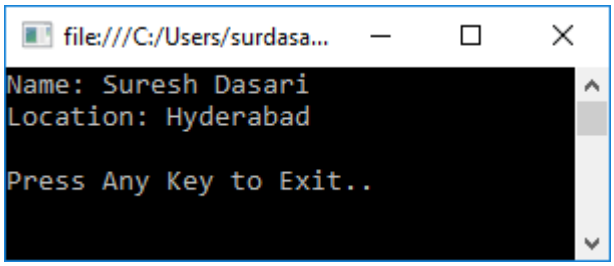
```
using System;

namespace Tutlane
{
   class User
   {
      private string name;
      private string location;
      public User(string a, string b)
      {
         name = a;
         location = b;
      }
      public string Name
      {
```

```
        get
        {
          return name;
        }
      }
      public string Location
      {
        get
        {
          return location;
        }
      }
    }
    class Program
    {
      static void Main(string[] args)
      {
         User u = new User("Suresh Dasari", "Hyderabad");
         // compile error
         // u.Name = "Rohini";
         // get accessor will invoke
         Console.WriteLine("Name: " + u.Name);
         // get accessor will invoke
         Console.WriteLine("Location: " + u.Location);
         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
      }
    }
}
```

If you observe the above example, we created properties using only get accessor to make the properties are read-only based on our requirements.

If we uncomment the commented code, we will get a compile error because our **Name** property doesn't contain any set accessor to set a new value. It's just read-only property.

When you execute the above c# program, you will get a result like as shown below.



This is how we can create read-only properties in c# applications based on our requirements.

C# Create Write Only Properties

As discussed, if a property contains the only set accessor, then we will call it a **write-only** property. Following is the example of creating write-only properties in the c# programming language.

```
using System;

namespace Tutlane
{
    class User
    {
      private string name;
      public string Name
      {
        set
        {
          name = value;
        }
      }
      private string location;
      public string Location
      {
         set
         {
            location = value;
```

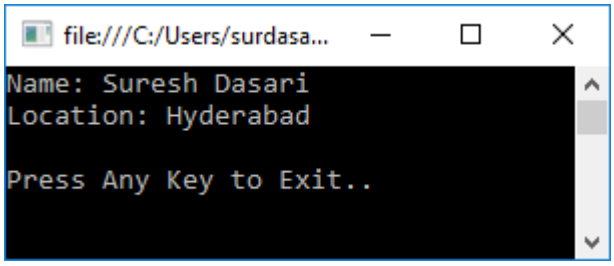
```
        }
      }
      public void GetUserDetails()
      {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Location: " + location);
      }
    }
    class Program
    {
       static void Main(string[] args)
       {
           User u = new User();
           u.Name = "Suresh Dasari";
           u.Location = "Hyderabad";
           // compile error
           //Console.WriteLine(u.Name);
           u.GetUserDetails();
           Console.WriteLine("\nPress Enter Key to Exit..");
           Console.ReadLine();
       }
    }
}
```

If you observe the above example, we created properties using only set accessor to make the properties are **write-only** based on our requirements.

If we uncomment the commented code, then we will get a compile error because our **Name** property doesn't contain any get accessor to return a value. It's a just write-only property.

When you execute the above c# program, you will get a result like as shown below.

noteghar.com
tiyghar.com



This is how we can create **write-only** properties in c# applications based on our requirements.

C# Auto Implemented Properties

In c#, a property is called an auto-implemented property when it contains accessors (**get**, **set**) without having any logic implementation.

Generally, the auto-implemented properties are useful whenever there is no logic implementation required in property accessors.

Following is the example of creating auto-implemented properties in the c# programming language.

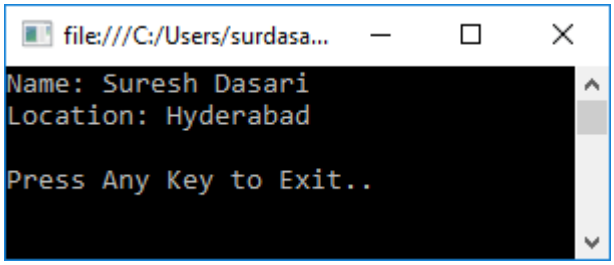
```
using System;

namespace Tutlane
{
    class User
    {
      public string Name { get; set; }
      public string Location { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User();
            u.Name = "Suresh Dasari";
            u.Location = "Hyderabad";
            Console.WriteLine("Name: " + u.Name);
```

noteghar.com
flyghar.com

```
            Console.WriteLine("Location: " + u.Location);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we created properties with get and set accessors without having any logic implementation.

When you execute the above c# program, we will get a result like as shown below.



This is how we can create **auto-implemented** properties in c# applications based on our requirements.

C# Properties Overview

The following are the important points which we need to remember about properties in the c# programming language.

- In c#, properties will enable class variables to expose in a public way using get and set accessors by hiding implementation details.
- In properties, a get accessor is used to return a property value and a set accessor is used to assign a new value.
- The **value** keyword in set accessor is used to define a value that is going to be assigned by the set accessor.
- In c#, the properties are categorized as read-write, read-only, or write-only.

flyghar.com
noteghar.com

Static Properties

C# also supports static properties, which belongs to the class rather than to the objects of the class. All the rules applicable to a static member are applicable to static properties also.

The following program shows a class with a static property.

flyghar.com

```
1. //C# : static Property
2. //Author: rajeshvs@msn.com
3. using System;
4. class MyClass
5. {
6.     private static int x;
7.     public static int X
8.     {
9.         get
10.             {
11.                 return x;
12.             }
13.             set
14.             {
15.                 x = value;
16.             }
17.         }
18.     }
19.     class MyClient
20.     {
21.         public static void Main()
22.         {
23.             MyClass.X = 10;
24.             int xVal = MyClass.X;
25.             Console.WriteLine(xVal);//Displays 10
26.         }
27.     }
```

noteghar.com

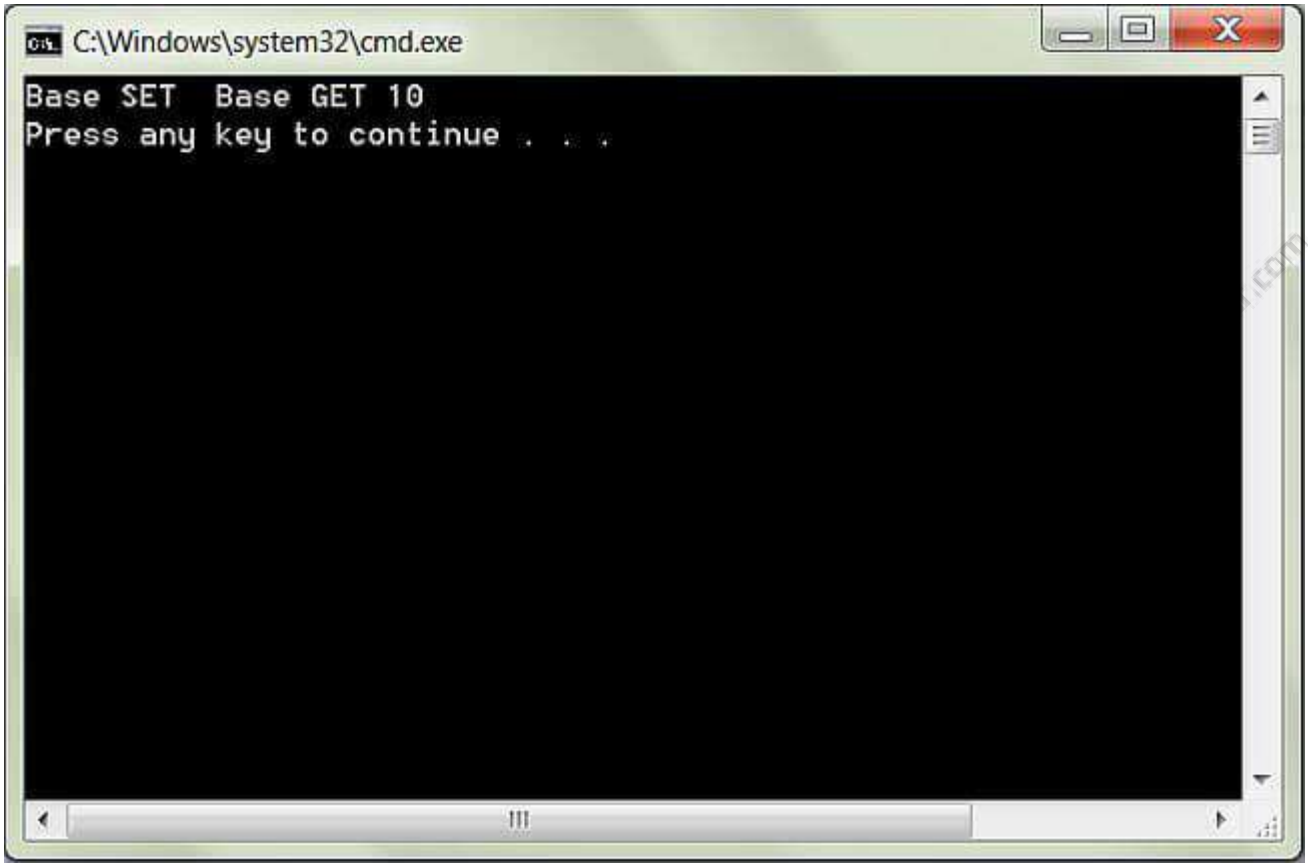
Remember that set/get accessor of static property can access only other static members of the class. Also ,static properties are invoking by using the class name.

Properties & Inheritance

The properties of a Base class can be inherited to a Derived class.

```
1. //C# : Property : Inheritance
2. //Author: rajeshvs@msn.com
3. using System;
4. class Base
5. {
6.     public int X
7.     {
8.         get
9.         {
10.                 Console.Write("Base GET");
11.                 return 10;
12.             }
13.             set
14.             {
15.                 Console.Write("Base SET");
16.             }
17.         }
18.     }
19.     class Derived : Base
20.     {
21.     }
22.     class MyClient
23.     {
24.         public static void Main()
25.         {
26.             Derived d1 = new Derived();
27.             d1.X = 10;
28.             Console.WriteLine(d1.X);
29.         }
30.     }
```

The output from above listing is shown below.



The above program is very straightforward. The inheritance of properties is just like inheritance any other member.

Properties & Polymorphism

A Base class property can be polymorphically overridden in a Derived class. But remember that the modifiers like virtual, override etc are using at property level, not at accessor level.

```
1. //C# : Property : Polymorphism
2. //Author: rajeshvs@msn.com
3. using System;
4. class Base
5. {
6.     public virtual int X
7.     {
8.         get
9.         {
```

noteghar.com

```
10.                 Console.Write("Base GET");
11.                 return 10;
12.             }
13.             set
14.             {
15.                 Console.Write("Base SET");
16.             }
17.         }
18.     }
19.     class Derived : Base
20.     {
21.         public override int X
22.         {
23.             get
24.             {
25.                 Console.Write("Derived GET");
26.                 return 10;
27.             }
28.             set
29.             {
30.                 Console.Write("Derived SET");
31.             }
32.         }
33.     }
34.     class MyClient
35.     {
36.         public static void Main()
37.         {
38.             Base b1 = new Derived();
39.             b1.X = 10;
40.             Console.WriteLine(b1.X);
41.         }
42.     }
```

noteghar.com
tiyghar.com

The output from above listing is shown below.

Abstract Properties

A property inside a class can be declared as abstract by using the keyword abstract. Remember that an abstract property in a class carries no code at all. The get/set accessors are simply represented with a semicolon. In the derived class we must implement both set and get assessors.

If the abstract class contains only set accessor, we can implement only set in the derived class.

The following program shows an abstract property in action.

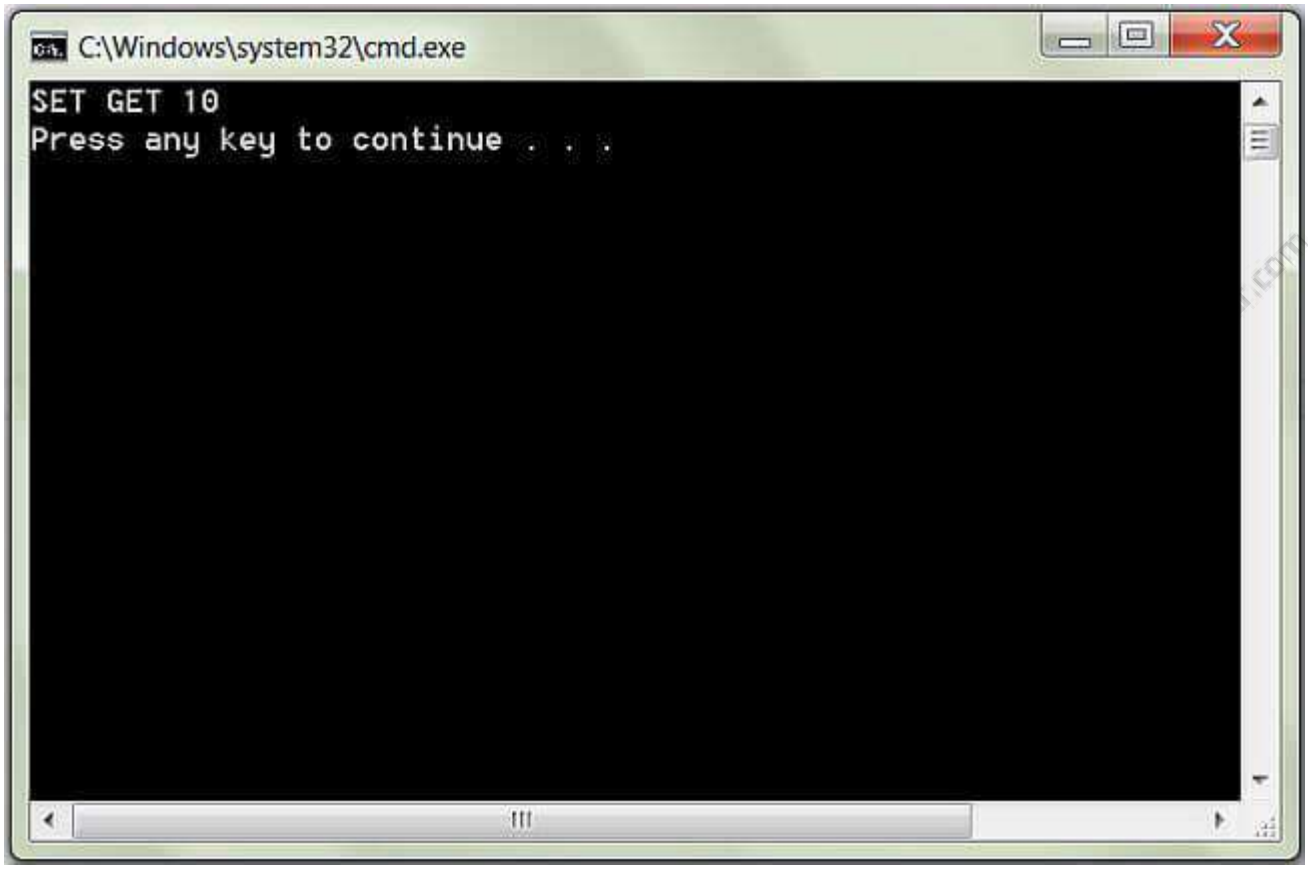
noteghar.com

```
1. //C# : Property : Abstract
2. //Author: rajeshvs@msn.com
3. using System;
4. abstract class Abstract
5. {
6.     public abstract int X
```

```
7.    {
8.        get;
9.        set;
10.           }
11.       }
12.       class Concrete : Abstract
13.       {
14.           public override int X
15.           {
16.               get
17.               {
18.                   Console.Write(" GET");
19.                   return 10;
20.               }
21.               set
22.               {
23.                   Console.Write(" SET");
24.               }
25.           }
26.       }
27.       class MyClient
28.       {
29.           public static void Main()
30.           {
31.               Concrete c1 = new Concrete();
32.               c1.X = 10;
33.               Console.WriteLine(c1.X);
34.           }
35.       }
```

The output from above listing is shown below.

noteghar.com
tiyghar.com



The properties are important features added in language level inside C#. They are very useful in GUI programming. Remember that the compiler actually generates the appropriate getter and setter methods when it parses the C# property syntax.

noteghar.com

Structures (Structs)

In c#, **structures** are same as classes but the only difference is classes are the reference types and structures are the value types. As a value type, the structures directly contain their value so their object or instance is stored on the stack and structures are faster than classes.

In c#, the structures can contain fields, properties, member functions, operators, constructors, events, indexers, constants and even other structure types.

flyghar.com

Create Structures in C#

In c#, structures can be created by using struct keyword. Following is the declaration of structure in the c# programming language.

```
public struct users
{
// Properties, Methods, Events, etc.
}
```

If you observe the above syntax, we defined a structure "**users**" using struct keyword with **public** access modifier. Here, the **public** access specifier will allow the users to create objects for this structure and inside of the body structure, we can create required fields, properties, methods and events to use it in our applications.

Following is the example of defining a structure in c# programming language.

```
public struct User
{
    public string name;
    public string location;
    public int age;
}
```

noteghar.com

If you observe the above example, we defined a structure called "**User**" with required fields and we can add required methods and properties based on our requirements.

Structure Initialization

In c#, structures can be instantiated with or without new keyword. Following is the example of assigning values to the variables of structure.

```
User u = new User();
u.name = "Suresh Dasari";
u.location = "Hyderabad";
u.age = 32;
```

To make use of fields, methods and events of structure, then it's mandatory to instantiate a structure with new keyword in c# programming language.

flyghar.com

You can also create an instance of a struct by just declaring it, without using the **new** operator. The object is created, although you can't use it until you've explicitly initialized all of its data members.

```
// Method 3 - Just declare it
DogCollar collar3;

// Error - Can't use collar3 yet (use of unassigned field)
Console.WriteLine(collar3.Length);
```

We need to first initialize all data members:

```
1  // Correct - initialize first
2  DogCollar collar3;
3  collar3.Length = 10.0;
4  collar3.Width = 5.0;
5  Console.WriteLine(collar3.Length);
```

Structure with Constructor

In c#, the structures won't allow us to declare a default constructor or a constructor without parameters and it won't allow us to initialize fields with values unless they are declared as const or static.

Following is the example of defining a structure with parameterized constructor and initializing the fields in constructor in c# programming language.

noteghar.com

```
public struct User
{
    public string name, location;
    // Parameterized Constructor
    public User(string a, string b)
    {
        name = a;
        location = b;
    }
}
```

If you observe above example, we defined a structure called "**User**" with required fields and parameterized constructor.

Structure with Default Constructor

As discussed, the structures will allow only parameterized constructors and fields cannot be initialized unless they are declared it as const or static.

Following is the example of defining a structure with a default constructor and initializing fields with values.

```
struct User
  {
   // Compile error
      public string name = "Suresh Dasari";
      public string location;
      public int age;
      // Compile error
      public User()
      {
          location = "Hyderabad";
          age = 32;
      }
  }
```

When we execute above code we will get a compiler errors because we declared a structure with default constructor (parameter less) and initialized a fields without defining it as const or static.

In c#, if we create a structure with the parameterized constructor, then we must need to explicitly initialize all the fields within the constructor before the control is returned to the caller otherwise we will get a compile-time error.

Following is the example of defining a structure with the parameterized constructor and required fields in c# programming language.

```
public struct User
{
    public string name, location;
    // Compile Error
    public User(string a)
    {
        name = a;
    }
}
```

When you execute the above code, we will get a compile-time error because we defined a **name** and **location** variables but we are assigning a value to the only **name** variable.

As discussed, if we create a structure with the parameterized constructor, then we must need to explicitly initialize all the fields before leaving the constructor.

Now we will see how to create a structure with fields and parameterized constructors in c# programming language with example.

Following is the example of creating a structure with different types of fields and parameterized constructors in c# programming language with various data members and member functions.

```
using System;

namespace Tutlane
{
    struct User
    {
        public const string name = "Suresh Dasari";
        public string location;
        public int age;
        public User(string a, int b)
        {
            location = a;
            age = b;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Declare object with new keyword
            User u = new User("Hyderabad", 31);
            // Declare object without new keyword
            User u1;
```

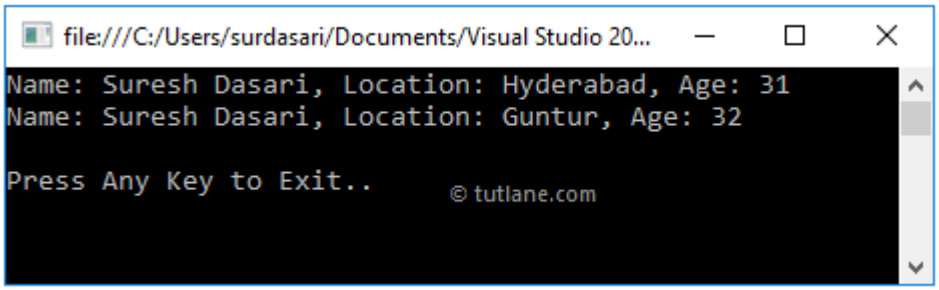
noteghar.com
flyghar.com

```
            Console.WriteLine("Name: {0}, Location: {1}, Age:
{2}", User.name, u.location, u.age);
            // Initialize Fields
            u1.location = "Guntur";
            u1.age = 32;
            Console.WriteLine("Name: {0}, Location: {1}, Age:
{2}", User.name, u1.location, u1.age);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

tiyeghar.com

If you observe above example, we defined a structure (**User**) by including a required fields, parameterized constructor and created an instance of structure (**User**) with and without new keyword to initialize or get field values based on our requirements.

When you execute the above c# program, you will get the result as shown below.



In c#, by using structures we can implement an **interface** but structures cannot inherit from another structure or class.

Structure Characteristics

The following are the important characteristics of structures in c# programming language.

noteghar.com

- In c#, structures are value types and those are defined by using struct keyword.
- During the structure declaration, the fields cannot be initialized unless they are defined as const or static.
- Structures in c# can include fields, properties, member functions, operators, constructors, events, indexers, constants and even other structure types.

- Structures cannot include default constructor (constructor without parameters) or destructor but it will allow us to declare constructors with parameters.
- A structure cannot inherit from another structure or class.
- In c#, the structure can implement **interfaces**.
- A structure can be instantiated with or without using a new keyword.

tiyeghar.com

Difference between Structure and Class

The following are the difference between structures and classes in c# programming language.

- In c#, classes are the reference types and structures are the value types.
- Classes can contain default constructor or destructor but structures will contain only constructors that have parameters.
- We can implement **inheritance** using Classes but structures won't support **inheritance**.
- Unlike classes, structs can be instantiated with or without using a new operator.

noteghar.com

Enum (Enumerator)

In c#, **enum** is a keyword which is used to declare an enumeration. In c#, the enumeration is a type that consists of a set of named constants as a list.

By using an enumeration, we can group constants that are logically related to each other. For example, the days of the week can be grouped together by using enumeration in c#.

flyghar.com

Enum Syntax

Following is the syntax of defining an enumeration using enum keyword in c# programming language.

```
enum enum_name
{
// enumeration list
}
```

If you observe the above syntax, we used enum keyword to define an enumeration based on our requirements.

Following is the example of defining an enumeration using enum keyword in c# programming language.

```
enum Week
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}
```

If you observe the above example, we define an enumeration "**Week**" with a list of named constants called enumeration list.

noteghar.com

In c# by default, the first-named constant in enumerator has a value of **0**, and the value of each successive item in enumerator will be increased by **1**. For example, in the above enumeration, Sunday value is **0**, Monday is **1**, Tuesday is **2**, and so forth.

In case, if we want to change the default values of an enumerator, then assigning a new value to the first item in enumerator will automatically assign incremental values to the successive items in an enumerator.

Following is the example of overriding the default values of an enumerator by assigning a new value to the first item in an enumerator.

```
enum Week
  {
      Sunday = 10,
      Monday,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday
  }
```

flyghar.com

In the above enumeration, the sequence of elements is forced to start from **10** instead of **0** like Sunday value is **10**, Monday is **11**, Tuesday is **12**, and so forth.

Enum with Integer Types

In c#, an enumeration can contain only integral data type items such as **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long** or **ulong**. It's not possible to use an enum with **string** or any other data types to define enum elements except numeric types.

The default type of enumeration element is **int** (integer). In case, if you want to change the integral type of enum to **byte**, then you need to mention a **byte** type with a colon (**:**) after the identifier like as shown below.

```
enum Week : byte
  {
      Sunday,
      Monday,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday
  }
```

noteghar.com

If you observe the above example, we are trying to change the default integral type of elements in an enumeration to **byte** type.

To get the values of enum elements in c#, an explicit cast is necessary to convert from **enum** type to an integral type.

For example, the following are the statements to get an enum item value by using a cast to convert from **enum** to **int**.

```
int a = (int)Week.Sunday; // It will return 0
int b = (int)Week.Monday; // It will return 1
```

tiyeghar.com

If you observe above statements, we are doing cast conversion (**enum** to **int**) to get an enum item values.

Enum Example

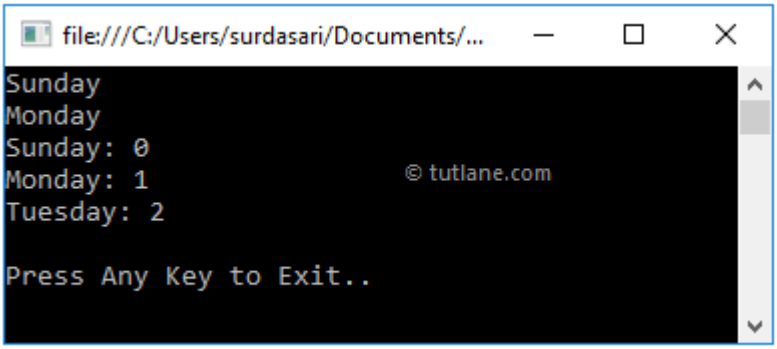
Following is the example of declaring an enumeration using enum keyword in c# programming language.

```
using System;

namespace Tutlane
{
    class Program
    {
        enum Week
        {
            Sunday,
            Monday,
            Tuesday,
            Wednesday,
            Thursday,
            Friday,
            Saturday
        }
        static void Main(string[] args)
        {
            int a = (int)Week.Sunday;
            int b = (int)Week.Monday;
            int c = (int)Week.Tuesday;
            Console.WriteLine(Week.Sunday);
            Console.WriteLine(Week.Monday);
            Console.WriteLine("Sunday: {0}", a);
            Console.WriteLine("Monday: {0}", b);
            Console.WriteLine("Tuesday: {0}", c);
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

noteghar.com

If you observe above example, we defined an enumeration **Week** and getting the values of enumeration items by explicitly converting from **enum** to **int** and assigned to an integer variables.

When you execute the above c# program, you will get the result as shown below.



If you observe the above result, the first enumerator (**Sunday**) has a value of **0** and the value of each successive enumerator is increased by **1**.

Enum Methods

In c#, we have a class called **Enum** that contains the following helper methods to work with an enumeration (enum).

Method	Description
Format	It is useful to convert the value of enum type to a specified string format.
GetName	It is useful to get the name of the specified enum item.
GetNames	It is useful to get all item names of the specified enum as an array.
GetValues	It is useful to get all item values of the specified enum as an array.
Parse	It is useful to convert the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object.
GetUnderlyingType	It is useful to return the underlying type of the specified enumeration.

Iterate through Enum

In c#, we can iterate or loop through enum items using for or foreach loop to get the enumeration item names or values by using enum helper methods.

Following is the example of a loop through enum items to get all item names and values using enum helper methods based on our requirements.

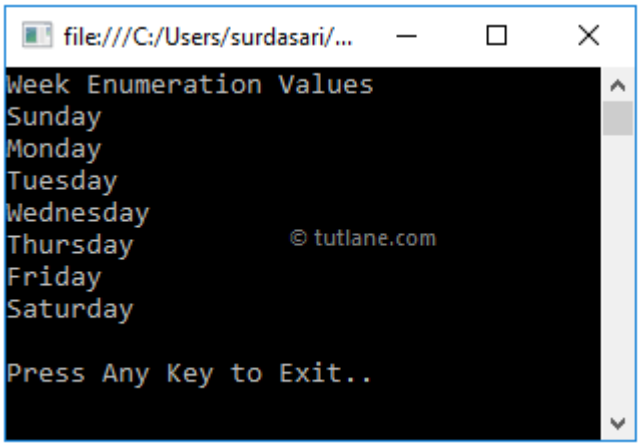
```
using System;
```

```
namespace Tutlane
{
    class Program
    {
        enum Week
        {
            Sunday,
            Monday,
            Tuesday,
            Wednesday,
            Thursday,
            Friday,
            Saturday
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Week Enumeration Values");
            foreach (string w in Enum.GetNames(typeof(Week)))
            {
                Console.WriteLine(w);
            }
            Console.WriteLine("\nPress Any Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

tiyeghar.com

If you observe above example, we are looping through an enumeration using foreach loop and getting all item names using **GetNames()** helper method.

When you execute the above c# program, we will get the result as shown below.



noteghar.com

If you observe the above result, we are able to get all enumeration item names by looping through an enumeration using a foreach loop based on our requirements.

In c#, it's better to define an enum within a namespace so that all the classes in the namespace can access equally and we can also nested an enum within class or struct based on our requirements.

tiyeghar.com

noteghar.com

Sealed Keyword

In c#, **sealed** is a keyword that is used to stop inheriting the particular class from other classes and we can also prevent overriding the particular properties or methods based on our requirements.

Generally, when we create a particular class we can inherit all the properties and methods in any class. In case, if you want to restrict access of defined class and its members, then by using a **sealed** keyword we can prevent other classes from inheriting the defined class.

Sealed Class

In c#, a sealed class can be defined by using a **sealed** keyword. As discussed, when we define a class with the **sealed** keyword, then we don't have a chance to inherit that particular class.

The following are the various ways of defining **sealed classes** in c# programming language.

```
sealed class Test
{
public string Name;
public string Location;
}
public sealed class Test1
{
public int Age;
}
sealed public class Test2
{
// Implementation
}
```

If you observe the above code snippet, we defined various classes with a **sealed** keyword to make sure that the defined classes are not inheritable to any class. In c#, we can use a **sealed** keyword before or after the access modifier to define sealed classes.

Following is the example of using a **sealed** keyword to define sealed classes in c# programming language.

```
using System;

namespace Tutlane
{
    // Base Class
```

```
    sealed class Users
    {
        public string name = "Suresh Dasari";
        public string location = "Hyderabad";
        public void GetInfo()
        {
            Console.WriteLine("Name: {0}", name);
            Console.WriteLine("Location: {0}", location);
        }
    }
    // Derived Class
    public class Details : Users
    {
        public int age = 32;
        public void GetAge()
        {
            Console.WriteLine("Age: {0}", age);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Details d = new Details();
            d.GetAge();
            d.GetInfo();
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

tlyghar.com

If you observe above example, we defined a class "**Users**" with **sealed** keyword and we are trying to inherit the properties from **Users** class using **Details** class.

When you execute the above c# program, we will get the result as shown below.

© tutlane.com

	Description
1	'Tutlane.Details': cannot derive from sealed type 'Tutlane.Users'
2	Inconsistent accessibility: base class 'Tutlane.Users' is less accessible than class 'Tutlane.Details'

noteghar.com

If you observe the above result, we are getting compile-time errors because we tried to inherit sealed class properties in another class.

Sealed Method or Property

In c#, we can also use the **sealed** keyword on a method or property that overrides a virtual method or property in a base class to allow other classes to derive from base class but to prevent them from overriding specific virtual methods or properties.

Following is the example of using a **sealed** keyword on a method that overrides a virtual method in a base class.

tiyeghar.com

```
using System;

namespace Tutlane
{
    public class A
    {
        public virtual void GetInfo()
        {
            Console.WriteLine("Base Class A Method");
        }
        public virtual void Test()
        {
            Console.WriteLine("Base Class A Test Method");
        }
    }
    public class B : A
    {
        public sealed override void GetInfo()
        {
            Console.WriteLine("Derived Class B Method");
        }
        public override void Test()
        {
            Console.WriteLine("Derived Class B Test Method");
        }
    }
    public class C : B
    {
        // Compile time error
        public override void GetInfo()
        {
            Console.WriteLine("Age: {0}", base.age);
        }
        public override void Test()
        {
            Console.WriteLine("Derived Class C Test Method");
        }
    }
    class Program
    {
```

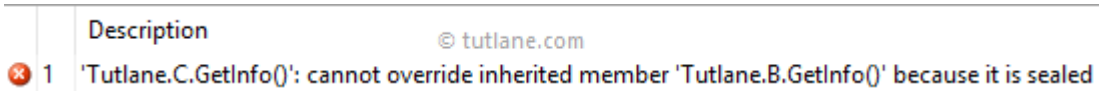
noteghar.com

```
        static void Main(string[] args)
        {
            C c = new C();
            c.GetInfo();
            c.Test();
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

tiyeghar.com

If you observe above example, we used a **sealed** keyword on method (**GetInfo**) that overrides a base class virtual method. In above example, class **C** is inherits from class **B** but class **C** cannot override a virtual method **GetInfo** that is declared in class **A** because that method is sealed in class **C**.

When you execute the above c# program, you will get a result as shown below.



This is how you can use the **sealed** keyword on method or property that overrides a base class virtual method or property.

Instead of using **sealed** keyword, we can also prevent derived classes to override a base class methods or properties by not declaring them as virtual.

Sealed Keyword Features

The following are the important points which we need to remember about the **sealed** keyword in c# programming language.

noteghar.com

- In c#, to apply a **sealed** keyword on method or property, it must always be used with override.
- In c#, we should not use **abstract** modifier with **sealed** class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.
- In c#, the local variables cannot be sealed.
- In c#, structs are implicitly sealed, they cannot be inherited.

Abstract

In c#, **abstract** is a keyword and it is useful to define classes and class members that are needed to be implemented or overridden in a derived class.

In c#, you can use abstract modifiers with classes, methods, properties, events, and indexers based on our requirements. The members which we defined as abstract or included in an abstract class must be implemented by classes that derive from an abstract class.

Now we will see how to use abstract modifier in classes and methods with examples.

Abstract Class

In c#, **abstract class** is a class that is declared with an abstract modifier. If we define a class with abstract modifier, then that class is intended only to be used as a base class for other classes.

The **abstract class** cannot be instantiated and it can contain both abstract and non-abstract members. The class that is derived from the abstract class must implement all the inherited abstract methods and accessors.

In c#, you can define an abstract class by using abstract keyword. Following is the example of defining an abstract class using abstract keyword.

```
abstract class Info
{
abstract public void GetDetails();
}
```

If you observe above code snippet, we defined an abstract class (**Info**) using abstract keyword with **GetDetails** method signature.

If we define a method with abstract modifier, then that method implementation must be done in a derived class.

Following is the example of implementing a class by deriving from the abstract class.

```
class User : Info
{
    public override void GetDetails()
    {
        // Method Implementation
    }
}
```

If you observe the code snippet, we inherited an abstract class (**Info**) in the **User** class and implemented a defined abstract method in the **User** class using override keyword. In c#, abstract methods are internally treated as virtual methods so those methods need to be overridden by the derived class.

In c#, we should not use a sealed keyword with abstract class because the sealed keyword will make a class as not inheritable but abstract modifier requires a class to be inherited.

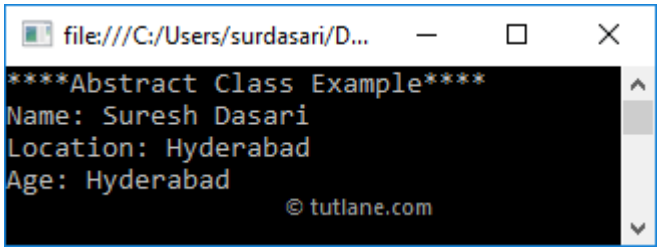
Following is the example of defining an abstract class using abstract modifier in c# programming language.

```
using System;

namespace Tutlane
{
    abstract class Info
    {
        abstract public void GetDetails(string x, string y, int z);
    }
    class User : Info
    {
        public override void GetDetails(string a, string b, int c)
        {
            Console.WriteLine("Name: {0}", a);
            Console.WriteLine("Location: {0}", b);
            Console.WriteLine("Age: {0}", b);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User();
            Console.WriteLine("****Abstract Class Example****");
            u.GetDetails("Suresh Dasari", "Hyderabad", 32);
            Console.ReadLine();
        }
    }
}
```

noteghar.com
noteghar.com

If you observe above example, we defined an abstract class called "**Info**" with required methods and the derived class "**User**" has implemented all the inherited abstract methods and accessors.

When you execute the above c# program, you will get the result as shown below.



This is how we can use abstract classes in our applications based on our requirements.

Abstract Class Features

The following are important features of abstract class in c# programming language.

- In c#, abstract classes cannot be instantiated.
- The abstract classes can contain both abstract and non-abstract methods and accessors.
- In c#, we should not use a sealed keyword with abstract class because the sealed keyword will make a class as not inheritable but abstract modifier requires a class to be inherited.
- A class that is derived from an abstract class must include all the implementations of inherited abstract methods and accessors.

Abstract Method

In c#, the **abstract method** is a method that is declared with an abstract modifier. If we define a method with abstract modifier, then that method doesn't contain any implementation and method declaration simply ends with a semicolon.

Following is the example of defining an abstract method in c# programming language.

```
public abstract void GetDetails();
```

The abstract methods in c# are permitted to declare only in abstract classes and the class that is derived from an abstract class must provide an implementation for defined abstract methods.

In c#, abstract methods are internally treated as virtual methods so those methods need to be overridden in the derived class and we should not static or virtual modifiers during abstract method declaration.

flyghar.com

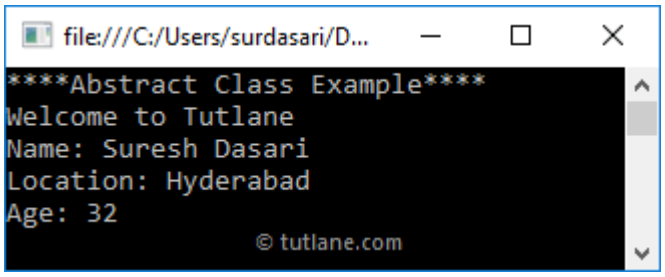
Following is the example of declaring an abstraction method in an abstract class in c# programming language.

```
using System;

namespace Tutlane
{
    abstract class Info
    {
        public void Welcome()
        {
            Console.WriteLine("Welcome to Tutlane");
        }
        public int age = 32;
        public abstract void GetDetails(string x, string y);
    }
    class User : Info
    {
        public override void GetDetails(string a, string b)
        {
            Welcome();
            Console.WriteLine("Name: {0}", a);
            Console.WriteLine("Location: {0}", b);
            Console.WriteLine("Age: {0}", age);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User();
            Console.WriteLine("****Abstract Class Example****");
            u.GetDetails("Suresh Dasari", "Hyderabad");
            Console.ReadLine();
        }
    }
}
```

noteghar.com

If you observe the above example, we defined an abstract class called "**Info**" with required abstract and non-abstract methods and the derived class "**User**" has implemented all the inherited abstract methods and accessors.

When you execute the above c# program, we will get the result as shown below.



This is how we can use abstract methods in c# abstract classes based on our requirements.

Abstract Method Features

The following are the important features of the abstract method in c# programming language.

- In c#, abstract methods are permitted to declare only within abstract classes.
- The abstract method declaration will not contain any implementation, only the derived classes will provide an actual implementation for abstract methods.
- In c#, abstract methods are internally treated as virtual methods so those methods need to be overridden in the derived class.
- We should not use static or virtual modifiers during the abstract method declaration.

In c#, abstract properties will act the same as abstract methods but the only difference is declaration and invocation syntax.

ADO.NET is a set of classes (a framework) to interact with data sources such as databases and XML files. ADO is the acronym for ActiveX Data Objects. It allows us to connect to underlying data or databases. It has classes and methods to retrieve and manipulate data.

The following are a few of the .NET applications that use ADO.NET to connect to a database, execute commands and retrieve data from the database.

- ASP.NET Web Applications
- Console Applications
- Windows Applications.

Various Connection Architectures

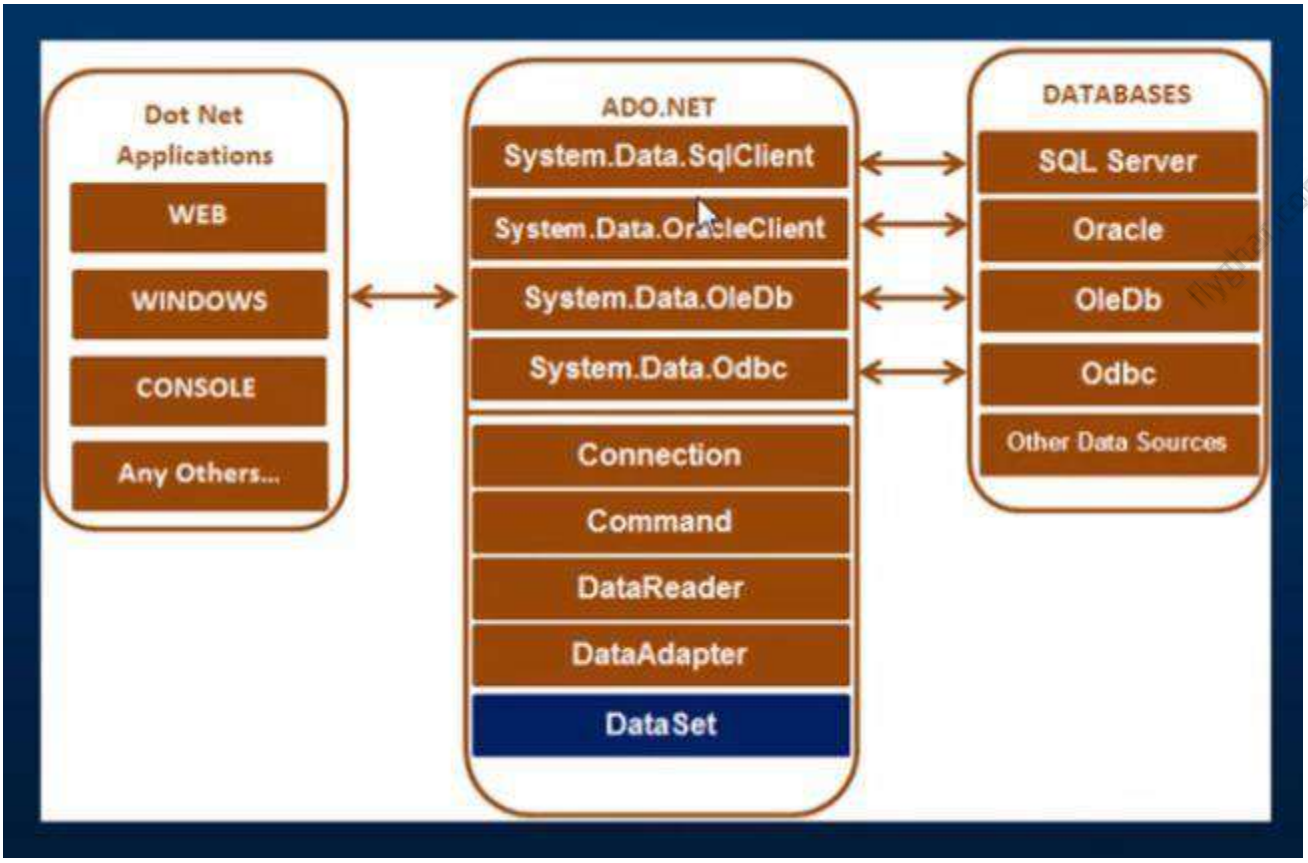
There are the following two types of connection architectures:

1. **Connected architecture:** the application remains connected with the database throughout the processing.

2. **Disconnected architecture:** the application automatically connects/disconnects during the processing. The application uses temporary data on the application side called a DataSet.

Understanding ADO.NET and it's class library

noteghar.com



In this diagram, we can see that there are various types of applications (Web Application, Console Application, Windows Application and so on) that use ADO.NET to connect to databases (SQL Server, Oracle, OleDb, ODBC, XML files and so on).

Important Classes in ADO.NET

We can also observe various classes in the preceding diagram. They are:

1. Connection Class
2. Command Class
3. DataReader Class
4. DataAdaptor Class
5. DataSet.Class

1. Connection Class

In ADO.NET, we use these connection classes to connect to the database. These connection classes also manage transactions and connection pooling. SqlConnection in ADO.NET represents a connection to a SQL Server database. In this article, we will learn in depth about SqlConnection class including what is SqlConnection, create a connection, and use SqlConnection in C# and how to use ADO.NET classes to work with SQL Server, MS Access, MySQL and Oracle databases.

What is a connection in ADO.NET?

ADO.NET connection is an object that provides database connectivity and the entry point to a database. When the connection of an object is instantiated, the constructor takes a connection string that contains the information about the database server, server type, database name, connection type, and database user credentials. Once the connection string is passed and the connection object is created, you can establish a connection with the database. A connection string is usually stored in the web.config file or app.config or appsetting.json file of an application.

What namespace or provider is used for connection class?

ADO.NET provides connection to multiple providers. Each provider has a functionality to connect with different database. Here is a list of data providers in ADO.NET and their purpose.

- Data Provider for SQL Server (System.Data.SqlClient).
- Data Provider for MS ACCESS (System.Data.OleDb).
- Data Provider for MYSQL (System.Data.Odbc).
- Data Provider for ORACLE (System.Data.OracleClient).

How to use connection class with this provider is given below-

- Connection object for SQL Server (SqlConnection).
- Connection object for MSACCESS (OleDbConnection).

- Connection object for MYSQL (OdbcConnaction).
- Connection object for ORACLE (OracleConnection).

ADO.NET SqlConnection Class

flyghar.com

It is used to establish an open connection to the SQL Server database. It is a sealed class so that cannot be inherited. SqlConnection class uses SqlDataAdapter and SqlCommand classes together to increase performance when connecting to a Microsoft SQL Server database.

Connection does not close explicitly even it goes out of scope. Therefore, you must explicitly close the connection by calling Close() method.

SqlConnection Signature

```
public sealed class SqlConnection : System.Data.Common.DbConnection, ICloneable, IDisposable
```

SqlConnection Constructors

Constructors	Description
SqlConnection()	It is used to initializes a new instance of the SqlConnection class.
SqlConnection(String)0	It is used to initialize a new instance of the SqlConnection class and takes connection string as an argument.
SqlConnection(String, SqlCredential)	It is used to initialize a new instance of the SqlConnection class that takes two parameters. First is connection string and second is sql credentials.

noteghar.com

SqlConnection Methods

Method	Description
BeginTransaction()	It is used to start a database transaction.
ChangeDatabase(String)	It is used to change the current database for an open SqlConnection.
ChangePassword(String, String)	It changes the SQL Server password for the user indicated in the connection string.
Close()	It is used to close the connection to the database.
CreateCommand()	It enlists in the specified transaction as a distributed transaction.
GetSchema()	It returns schema information for the data source of this SqlConnection.
Open()	It is used to open a database connection.
ResetStatistics()	It resets all values if statistics gathering is enabled.

In the connection string:

- **Data Source**: This identifies the Server name, which could be the local machine, machine domain name or IP address
- **Initial Catalog**: This identifies the database name.
- **Integrated Security**: When you have started database authentication login with Windows authentication, Integrated Security specifies Integrated Security=”True” in connection string, else when you have started the database authentication login with Server authentication Integrated Security specifies Integrated Security=”false” in the connection string
- **User Id**: Name of the user configured in SQL Server.
- **Password**: Password matching SQL Server User ID.

noteghar.com

SqlConnection Example

Now, let's create an example that establishes a connection to the SQL Server. We have created a **Student** database and will use it to connect. Look at the following C# code.

```
1. using (SqlConnection connection = new SqlConnection(connection
   String))
2. {
3.   connection.Open();
4. }
```

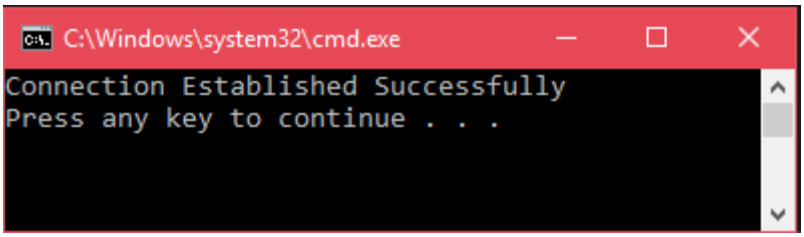
Using block is used to close the connection automatically. We don't need to call close () method explicitly, **using** block do this for ours implicitly when the code exits the block.

// Program.cs

```
1. using System;
2. using System.Data.SqlClient;
3. namespace AdoNetConsoleApplication
4. {
5.     class Program
6.     {
7.         static void Main(string[] args)
8.         {
9.             new Program().Connecting();
10.            }
11.            public void Connecting()
12.            {
13.                using (
14.                        // Creating Connection
15.                        SqlConnection con = new SqlConnection("dat
   a source=.; database=student; integrated security=SSPI")
16.                    )
17.                {
18.                    con.Open();
19.                    Console.WriteLine("Connection Established Succe
   ssfully");
20.                }
21.            }
```

```
22.         }
23.     }
```

Output:



What, if we don't use **using** block.

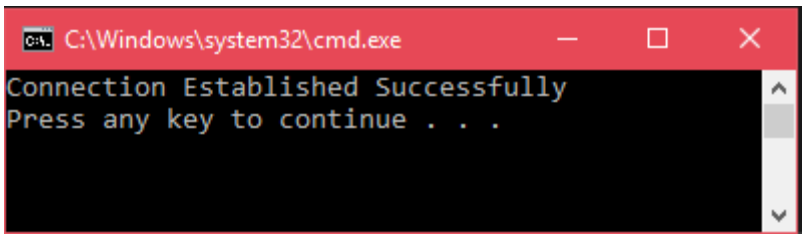
If we don't use using block to create connection, we have to close connection explicitly. In the following example, we are using try-block instead of using block.

// Program.cs

```
1. using System;
2. using System.Data.SqlClient;
3. namespace AdoNetConsoleApplication
4. {
5.     class Program
6.     {
7.         static void Main(string[] args)
8.         {
9.             new Program().Connecting();
10.         }
11.         public void Connecting()
12.         {
13.             SqlConnection con = null;
14.             try
15.             {
16.                 // Creating Connection
17.                 con = new SqlConnection("data source=.; datab
ase=student; integrated security=SSPI");
```

```
18.                 con.Open();
19.                 Console.WriteLine("Connection Established Succe
   ssfully");
20.             }
21.             catch (Exception e)
22.             {
23.                 Console.WriteLine("OOPs, something went wrong
   .\n"+e);
24.             }
25.             finally
26.             {   // Closing the connection
27.                 con.Close();
28.             }
29.         }
30.     }
31. }
```

Output:



2. Command Class

The Command class provides methods for storing and executing SQL statements and Stored Procedures.

What is ADO.NET SqlCommand Class in C#?

The ADO.NET SqlCommand class in C# is used to store and execute the SQL statement against the SQL Server database. As you can

see in the below image, the **SqlCommand** class is a sealed class and is inherited from the DbCommand class and implement the ICloneable interface. As a sealed class, it cannot be inherited.

```
public sealed class SqlCommand : DbCommand, ICloneable
{
}
```

flyghar.com

Constructors of ADO.NET SqlCommand Class in C#

The SqlCommand class in C# provides the following four constructors.

```
public SqlCommand();
public SqlCommand(string cmdText);
public SqlCommand(string cmdText, SqlConnection connection);
public SqlCommand(string cmdText, SqlConnection connection, SqlTransaction transaction);
```

Let us discuss each of these constructors in detail.

SqlCommand():

This constructor is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class..

SqlCommand(string cmdText):

It is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class with the text of the query. Here, the cmdText is the text of the query that we want to execute.

SqlCommand(string cmdText, SqlConnection connection):

It is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class with the text of the query and a System.Data.SqlClient.SqlConnection. Here, the cmdText is the text of the query that we want to execute and the parameter connection is the connection to an instance of SQL Server.

notegha

SqlCommand(string cmdText, SqlConnection connection, SqlTransaction transaction):

It is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class with the text of the query, a SqlConnection instance, and the SqlTransaction instance.

Here, the parameter cmdText is the text of the query. The parameter connection a SqlConnection that represents the connection to an instance of SQL Server and the parameter transaction is the SqlTransaction in which the SqlCommand executes.

Methods of SqlCommand Class in C#

The SqlCommand class in C# provides the following methods.

1. **BeginExecuteNonQuery():** This method initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this System.Data.SqlClient.SqlCommand.
2. **Cancel():** This method tries to cancel the execution of a System.Data.SqlClient.SqlCommand.
3. **Clone():** This method creates a new System.Data.SqlClient.SqlCommand object is a copy of the current instance.
4. **CreateParameter():** This method creates a new instance of a System.Data.SqlClient.SqlParameter object.
5. **ExecuteReader():** This method Sends the System.Data.SqlClient.SqlCommand.CommandText to the System.Data.SqlClient.SqlCommand.Connection and builds a System.Data.SqlClient.SqlDataReader.
6. **ExecuteScalar():** This method Executes the query, and returns the first column of the first row in the result set returned by the query. Additional columns or rows are ignored.
7. **ExecuteNonQuery():** This method executes a Transact-SQL statement against the connection and returns the number of rows affected.
8. **Prepare():** This method creates a prepared version of the command on an instance of SQL Server.
9. **ResetCommandTimeout():** This method resets the CommandTimeout property to its default value.

Example to understand the ADO.NET SqlCommand Object in C#:

We are going to use the following student table to understand the SqlCommand object.

flyghar.com
noteghar.com

Id	Name	Email	Mobile
101	Anurag	Anurag@dotnettutorial.net	1234567890
102	Priyanka	Priyanka@dotnettutorial.net	2233445566
103	Preety	Preety@dotnettutorial.net	6655443322
104	Sambit	Sambit@dotnettutorial.net	9876543210

Please use the below SQL script to create a database called StudentDB, a table called Student with the required test data.

```
CREATE DATABASE StudentDB;
GO
USE StudentDB;
GO
CREATE TABLE Student(
Id INT PRIMARY KEY,
Name VARCHAR(100),
Email VARCHAR(50),
Mobile VARCHAR(50)
)
GO
INSERT INTO Student VALUES (101, 'Anurag',
'Anurag@dotnettutorial.net', '1234567890')
INSERT INTO Student VALUES (102, 'Priyanka',
'Priyanka@dotnettutorial.net', '2233445566')
INSERT INTO Student VALUES (103, 'Preety',
'Preety@dotnettutorial.net', '6655443322')
INSERT INTO Student VALUES (104, 'Sambit',
'Sambit@dotnettutorial.net', '9876543210')
GO
```

Note: ExecuteReader, **ExecuteNonQuery**, and **ExecuteScalar** are the methods that are commonly used. Let us see three examples to understand these methods.

ExecuteReader method of SqlCommand Object in C#:

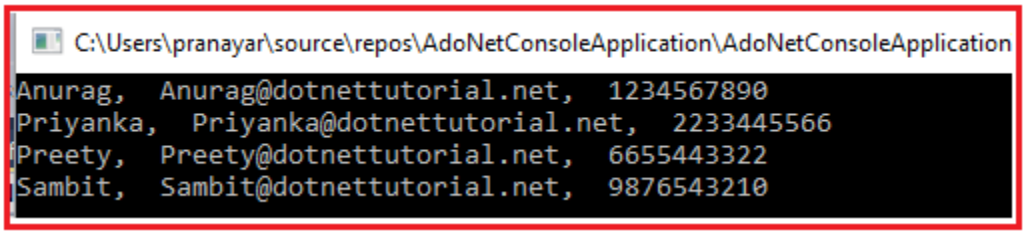
As we already discussed this method is used to send the CommandText to the Connection and builds a SqlDataReader. When your T-SQL statement returns more than a single value (for example rows of data), then you need to use the ExecuteReader method.

Let us understand this with an example. The following example uses the **ExecuteReader** method of the SqlCommand object to executes the T-SQL statement which returns multiple rows of data.

```
using System;
using System.Data.SqlClient;
namespace AdoNetConsoleApplication
{
class Program
{
static void Main(string[] args)
{
try
{
string ConString = "data source=.; database=StudentDB;
integrated security=SSPI";
using (SqlConnection connection = new
SqlConnection(ConString))
{
// Creating SqlCommand objcet
```

```
SqlCommand cm = new SqlCommand("select * from student", connection);
// Opening Connection
connection.Open();
// Executing the SQL query
SqlDataReader sdr = cm.ExecuteReader();
while (sdr.Read())
{
Console.WriteLine(sdr["Name"] + ", " + sdr["Email"] + ", " + sdr["Mobile"]);
}
}
}
catch (Exception e)
{
Console.WriteLine("OOPs, something went wrong.\n" + e);
}
Console.ReadKey();
}
}
}
```

Once you execute the program, you will get the following output as expected.



Understanding the ADO.NET SqlCommand Object in C#:

In our example, we are creating an instance of the **SqlCommand** by using the constructor which takes two parameters as shown in the below image. The first parameter is the **command text** that we want to execute, and the second parameter is the connection object which provides the database details on which the command is going to execute.

You can also create the **SqlCommand** object using the parameterless constructor, and later you can specify the command text and connection using the **CommandText** and the **Connection** properties of the **SqlCommand** object as shown in the below example.

```
using System;
using System.Data.SqlClient;
namespace AdoNetConsoleApplication
{
class Program
{
static void Main(string[] args)
{
try
```

```
{
string ConString = "data source=.; database=StudentDB;
integrated security=SSPI";
using (SqlConnection connection = new
SqlConnection(ConString))
{
// Creating SqlCommand objcet
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "select * from student";
cmd.Connection = connection;
// Opening Connection
connection.Open();
// Executing the SQL query
SqlDataReader sdr = cmd.ExecuteReader();
while (sdr.Read())
{
Console.WriteLine(sdr["Name"] + ", " + sdr["Email"] + ", " +
sdr["Mobile"]);
}
}
}
catch (Exception e)
{
Console.WriteLine("OOPs, something went wrong.\n" + e);
}
```

```
Console.ReadKey();
}
}
}
```

ExecuteScalar Method of SqlCommand Object in C#:

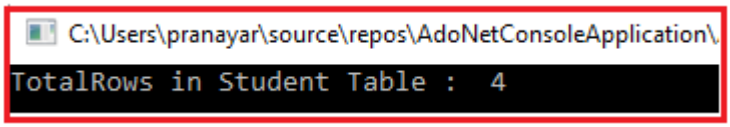
When your T-SQL query or stored procedure returns a single(i.e. scalar) value then you need to use the **ExecuteScalar** method of the SqlCommand object in C#.

Let us understand this with an example. Now, we need to fetch the total number of records present in the Student table. As we know it is going to return a single value, so this is an ideal situation to use the **ExecuteScalar** method. The following example will retrieve the total number of records present in the **Student** table.

```
using System;
using System.Data.SqlClient;
namespace AdoNetConsoleApplication
{
class Program
{
static void Main(string[] args)
{
try
{
string ConString = "data source=.; database=StudentDB;
integrated security=SSPI";
using (SqlConnection connection = new
SqlConnection(ConString))
{
```

```
// Creating SqlCommand objcet
SqlCommand cmd = new SqlCommand("select count(id) from student", connection);
// Opening Connection
connection.Open();
// Executing the SQL query
// Since the return type of ExecuteScalar() is object, we are type casting to int datatype
int TotalRows = (int)cmd.ExecuteScalar();
Console.WriteLine("TotalRows in Student Table : " + TotalRows);
}
}
catch (Exception e)
{
Console.WriteLine("OOPs, something went wrong.\n" + e);
}
Console.ReadKey();
}
}
}
```

The return type of the ExecuteScalar method is an object, so here we need to typecast it into integer type. Now, if you execute the above program, then you will get the following output.

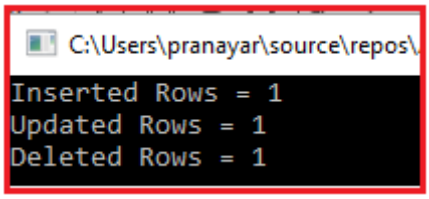


ExecuteNonQuery Method of ADO.NET SqlCommand Object in C#:
When you want to perform Insert, Update or Delete operations and want to return the number of rows affected by your query then you need to use the **ExecuteNonQuery** method of the SqlCommand object in C#.
Let us understand this with an example. The following example performs an Insert, Update and Delete operations using the **ExecuteNonQuery()** method.

```
using System;
using System.Data.SqlClient;
namespace AdoNetConsoleApplication
{
class Program
{
static void Main(string[] args)
{
try
{
string ConString = "data source=.; database=StudentDB;
integrated security=SSPI";
using (SqlConnection connection = new
SqlConnection(ConString))
{
SqlCommand cmd = new SqlCommand("insert into Student
values (105, 'Ramesh', 'Ramesh@dotnettutorial.net',
'1122334455')", connection);
connection.Open();
int rowsAffected = cmd.ExecuteNonQuery();
```

```
Console.WriteLine("Inserted Rows = " + rowsAffected);
//Set to CommandText to the update query. We are reusing the command object,
//instead of creating a new command object
cmd.CommandText = "update Student set Name = 'Ramesh Changed' where Id = 105";
rowsAffected = cmd.ExecuteNonQuery();
Console.WriteLine("Updated Rows = " + rowsAffected);
//Set to CommandText to the delete query. We are reusing the command object,
//instead of creating a new command object
cmd.CommandText = "Delete from Student where Id = 105";
rowsAffected = cmd.ExecuteNonQuery();
Console.WriteLine("Deleted Rows = " + rowsAffected);
}
}
catch (Exception e)
{
Console.WriteLine("OOPs, something went wrong.\n" + e);
}
Console.ReadKey();
}
}
}
```

Output:



flyghar.com
noteghar.com

What is ADO.NET SqlDataReader Class in C#?

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

The **SqlDataReader is connection-oriented**. It means it requires an open or active connection to the data source while reading the data. The data is available as long as the connection with the database exists

SqlDataReader is read-only. It means it is also not possible to change the data using SqlDataReader. You also need to open and close the connection explicitly.

SqlDataReader Class Signature:

If you look at the following image, you will see that this class is inherited from **DbDataReader** class and implements the **IDisposable** interface.

```
public class SqlDataReader : DbDataReader, IDataReader, IDisposable, IDataRecord
{
}
```

ADO.NET SqlDataReader Class Properties in C#:

The SqlDataReader class provides the following properties.

1. **Connection**: It gets the System.Data.SqlClient.SqlConnection associated with the System.Data.SqlClient.SqlDataReader.
2. **Depth**: It gets a value that indicates the depth of nesting for the current row.
3. **FieldCount**: It gets the number of columns in the current row.
4. **HasRows**: It gets a value that indicates whether the System.Data.SqlClient.SqlDataReader contains one or more rows.
5. **IsClosed**: It retrieves a Boolean value that indicates whether the specified System.Data.SqlClient.SqlDataReader instance has been closed.
6. **RecordsAffected**: It gets the number of rows changed, inserted, or deleted by the execution of the Transact-SQL statement.
7. **VisibleFieldCount**: It gets the number of fields in the System.Data.SqlClient.SqlDataReader that is not hidden.
8. **Item[String]**: It gets the value of the specified column in its native format given the column name.
9. **Item[Int32]**: It gets the value of the specified column in its native format given the column ordinal.

ADO.NET SqlDataReader Class Methods in C#:

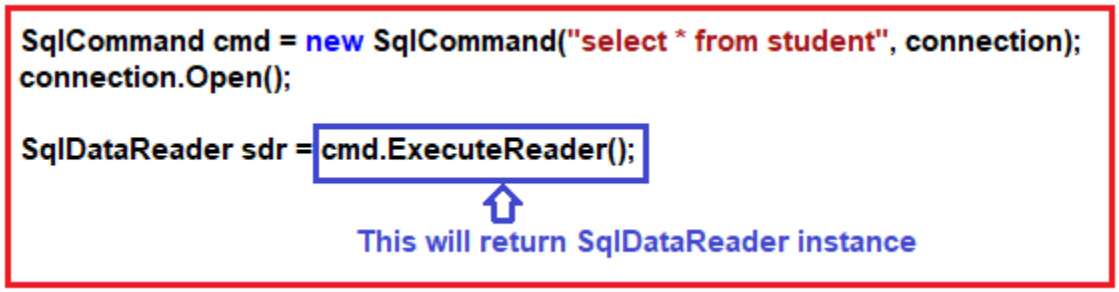
The SqlDataReader class provides the following methods.

1. **Close():** It closes the SqlDataReader object.
2. **GetBoolean(int i):** It gets the value of the specified column as a Boolean. Here, parameter i is the zero-based column ordinal.
3. **GetByte(int i):** It gets the value of the specified column as a byte. Here, parameter i is the zero-based column ordinal.
4. **GetChar(int i):** It gets the value of the specified column as a single character. Here, parameter i is the zero-based column ordinal.

5. **GetDateTime(int i):** It gets the value of the specified column as a System.DateTime object. Here, parameter i is the zero-based column ordinal.
6. **GetDecimal(int i):** It gets the value of the specified column as a System.Decimal object. Here, parameter i is the zero-based column ordinal.
7. **GetDouble(int i):** It gets the value of the specified column as a double-precision floating-point number. Here, parameter i is the zero-based column ordinal.
8. **GetFloat(int i):** It gets the value of the specified column as a single-precision floating-point number. Here, parameter i is the zero-based column ordinal.
9. **GetName(int i):** It gets the name of the specified column. Here, parameter i is the zero-based column ordinal.
10. **GetSchemaTable():** It returns a System.Data.DataTable that describes the column metadata of the System.Data.SqlClient.SqlDataReader
11. **GetValue(int i):** It gets the value of the specified column in its native format. Here, parameter i is the zero-based column ordinal.
12. **GetValues(object[] values):** It Populates an array of objects with the column values of the current row. Here, the parameter values is an array of System.Object into which to copy the attribute columns.
13. **NextResult():** It advances the data reader to the next result when reading the results of batch Transact-SQL statements.
14. **Read():** It Advances the System.Data.SqlClient.SqlDataReader to the next record and returns true if there are more rows; otherwise false.

How to create an instance of the ADO.NET SqlDataReader class in C#?

You can not create the instance of SqlDataReader using the new keyword. Then the question is how we get or create the instance of SqlDataReader class. In order to create the instance of SqlDataReader class, what you need to do is, call the **ExecuteReader** method of the **SqlCommand** object which will return an instance of SqlDataReader class as shown in the below image.



Example to understand the C# SqlDataReader Object in ADO.NET:

We are going to use the following student table to understand the SqlDataReader object in C#.

Id	Name	Email	Mobile
101	Anurag	Anurag@dotnettutorial.net	1234567890
102	Priyanka	Priyanka@dotnettutorial.net	2233445566
103	Preety	Preety@dotnettutorial.net	6655443322
104	Sambit	Sambit@dotnettutorial.net	9876543210

Please use the below SQL script to create a database called StudentDB, a table called Student with the required test data.

```
CREATE DATABASE StudentDB;
GO
USE StudentDB;
GO
CREATE TABLE Student(
Id INT PRIMARY KEY,
Name VARCHAR(100),
Email VARCHAR(50),
Mobile VARCHAR(50)
)
GO
INSERT INTO Student VALUES (101, 'Anurag', 'Anurag@dotnettutorial.net',
'1234567890')
INSERT INTO Student VALUES (102, 'Priyanka', 'Priyanka@dotnettutorial.net',
'2233445566')
INSERT INTO Student VALUES (103, 'Preety', 'Preety@dotnettutorial.net', '6655443322')
INSERT INTO Student VALUES (104, 'Sambit', 'Sambit@dotnettutorial.net',
'9876543210')
GO
```

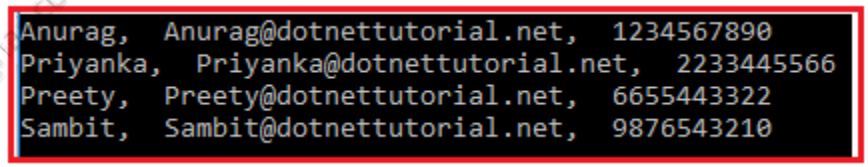
Example: Using SqlDataReader in C#

We need to fetch all the data from the student table and need to display it in the console using SqlDataReader. The following code exactly does the same thing. In the below example, we use the **Read()** method of the SqlDataReader object to loop through the items of the **SqlDataReader** object. The Read method returns true as long as there are rows to read from the SqlDataReader object. If there are no more rows to read, then this method will simply return false.

```
using System;
using System.Data.SqlClient;
namespace AdoNetConsoleApplication
{
class Program
{
static void Main(string[] args)
```

```
{
try
{
string ConString = "data source=.; database=StudentDB; integrated security=SSPI";
using (SqlConnection connection = new SqlConnection(ConString))
{
// Creating the command object
SqlCommand cmd = new SqlCommand("select * from student", connection);
// Opening Connection
connection.Open();
// Executing the SQL query
SqlDataReader sdr = cmd.ExecuteReader();
//Looping through each record
while (sdr.Read())
{
Console.WriteLine(sdr["Name"] + ", " + sdr["Email"] + ", " + sdr["Mobile"]);
}
}
}
catch (Exception e)
{
Console.WriteLine("OOPs, something went wrong.\n" + e);
}
Console.ReadKey();
}
}
}
```

Output:



Note: The DataReader object increases the performance of the application as well as reduces the system overheads and the reason for this is, one row at a time is stored in memory.

flyghar.com

noteghar.com

ADO.NET DataSet

It is a collection of data tables that contain the data. It is used to fetch data without interacting with a Data Source that's why, it also known as **disconnected** data access method. It is an in-memory data store that can hold more than one table at the same time. We can use DataRelation object to relate these tables. The DataSet can also be used to read and write data as XML document.

flyghar.com

ADO.NET provides a DataSet class that can be used to create DataSet object. It contains constructors and methods to perform data related operations.

ADO.NET DataAdapter

The DataAdapter works as a bridge between a DataSet and a data source to retrieve data. DataAdapter is a class that represents a set of SQL commands and a database connection. It can be used to fill the DataSet and update the data source.

DataAdapter Class Signature

```
public class DataAdapter : System.ComponentModel.Component, System.Data.IDataAdapter
```

DataAdapter Constructors

Constructors	Description
DataAdapter()	It is used to initialize a new instance of a DataAdapter class.
DataAdapter(DataAdapter)	It is used to initializes a new instance of a DataAdapter class from an existing object of the same type.

noteghar.com

Methods

Method	Description
CloneInternals()	It is used to create a copy of this instance of DataAdapter.
Dispose(Boolean)	It is used to release the unmanaged resources used by the DataAdapter.
Fill(DataSet)	It is used to add rows in the DataSet to match those in the data source.
FillSchema(DataSet, SchemaType, String, IDataReader)	It is used to add a DataTable to the specified DataSet.
GetFillParameters()	It is used to get the parameters set by the user when executing an SQL SELECT statement.
ResetFillLoadOption()	It is used to reset FillLoadOption to its default state.
ShouldSerializeAcceptChangesDuringFill()	It determines whether the AcceptChangesDuringFill property should be persisted or not.
ShouldSerializeFillLoadOption()	It determines whether the FillLoadOption property should be persisted or not.
ShouldSerializeTableMappings()	It determines whether one or more DataTableMapping objects exist or not.
Update(DataSet)	It is used to call the respective INSERT, UPDATE, or DELETE statements.

Example

// DataSetDemo.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="DataSetDemo.aspx
.cs"
Inherits="DataSetExample.DataSetDemo" %>
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
```

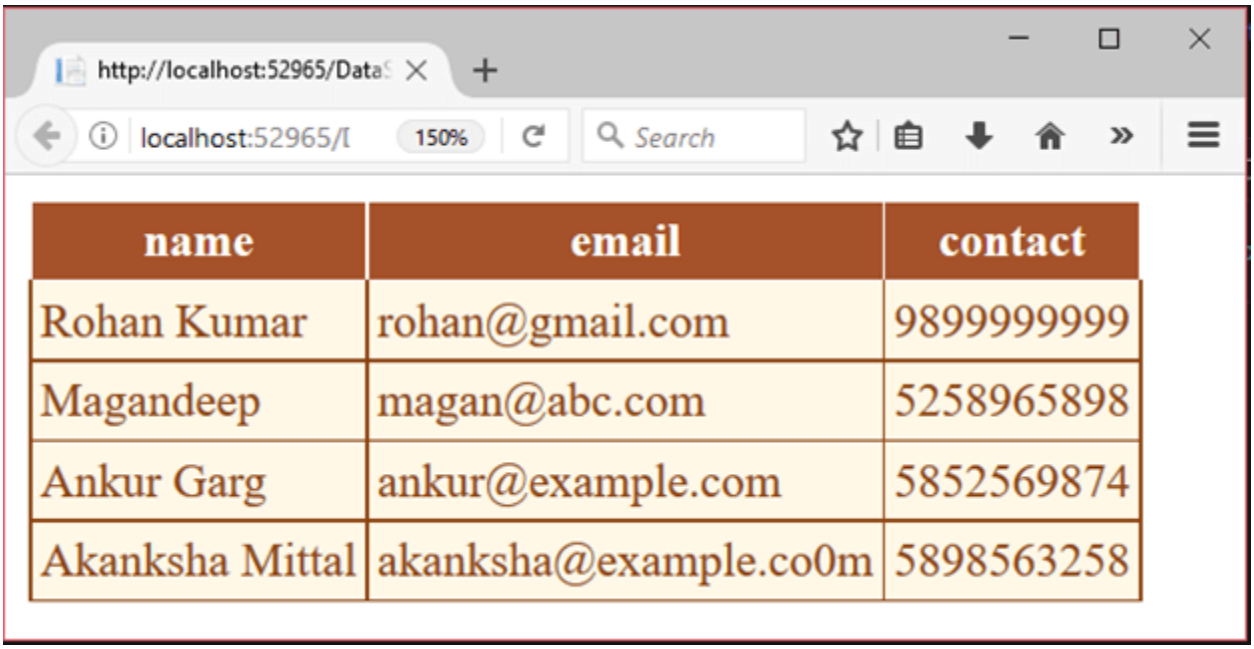
```
10.    <form id="form1" runat="server">
11.        <div>
12.
13.        </div>
14.        <asp:GridView ID="GridView1" runat="server" CellPadding="3" BackColor="#D
    EBA84"
15.        BorderColor="#DEBA84" BorderStyle="None" BorderWidth="1px" CellSpacing="2
    ">
16.            <FooterStyle BackColor="#F7DFB5" ForeColor="#8C4510" />
17.            <HeaderStyle BackColor="#A55129" Font-
    Bold="True" ForeColor="White" />
18.            <PagerStyle ForeColor="#8C4510" HorizontalAlign="Center" />
19.            <RowStyle BackColor="#FFF7E7" ForeColor="#8C4510" />
20.            <SelectedRowStyle BackColor="#738A9C" Font-
    Bold="True" ForeColor="White" />
21.            <SortedAscendingCellStyle BackColor="#FFF1D4" />
22.            <SortedAscendingHeaderStyle BackColor="#B95C30" />
23.            <SortedDescendingCellStyle BackColor="#F1E5CE" />
24.            <SortedDescendingHeaderStyle BackColor="#93451F" />
25.        </asp:GridView>
26.    </form>
27.</body>
28.</html>
```

CodeBehind

```
1.  using System;
2.  using System.Data.SqlClient;
3.  using System.Data;
4.  namespace DataSetExample
5.  {
6.      public partial class DataSetDemo : System.Web.UI.Page
7.      {
8.          protected void Page_Load(object sender, EventArgs e)
9.          {
```

```
10.         using (SqlConnection con = new SqlConnection("data source=.; database=stu
    dent; integrated security=SSPI"))
11.         {
12.             SqlDataAdapter sde = new SqlDataAdapter("Select * from student", con);
13.             DataSet ds = new DataSet();
14.             sde.Fill(ds);
15.             GridView1.DataSource = ds;
16.             GridView1.DataBind();
17.         }
18.     }
19.   }
20.}
```

Output:



name	email	contact
Rohan Kumar	rohan@gmail.com	9899999999
Magandeep	magan@abc.com	5258965898
Ankur Garg	ankur@example.com	5852569874
Akanksha Mittal	akanksha@example.co0m	5898563258

noteghar.com
noteghar.com

DataSet	DataReader
Dataset object can contain mutilple rowsets from the same data source as well as from the relationships between them.	DataReader provides forward only and read only access to data.
Dataset is a disconnected architecture.	Datareader is connected architecture.
Dataset can persist data.	Datareader can not persist data.
A Dataset is well suited for data that needs to be retrieved from multiple tables.	It has live connection while reading data.
Dataset is slower tha Datareader due to overhead.	Speed performance is better in DataReader.

noteghar.com
flyghar.com

C# Delegates

In c#, the **delegate** is a type that defines a method signature and it is useful to hold the reference of one or more methods which are having the same signatures.

By using delegates, you can invoke the methods and send methods as an argument to other methods.

In c#, the delegate is a reference type and it's type-safe and secure. The delegates are similar to function pointers in c++.

C# Delegate Declaration Syntax

In c#, the declaration of delegate will be same as method signature but the only difference is we will use a `delegate` keyword to define delegates.

Following is the syntax of defining a delegate using `delegate` keyword in c# programming language.

```
<access_modifier> delegate <return_type> <delegate_name>(<parameters>)
```

Following is the example of declaring a delegate using `delegate` keyword in c#.

```
public delegate void UserDetails(string name);
```

If you observe the above example, the declaration of a delegate is same as method declaration with required parameter types and return value.

The above defined method "**UserDetails**" can be pointed to any other method which is having the same parameters and return type.

In c#, the delegates must be instantiated with a method or an expression that has the same return type and parameters. We can invoke a method through the delegate instance.

C# Delegate Example

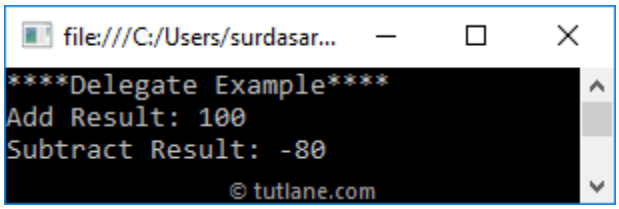
Following is the example of declaring and using a **delegate** in c# programming language.

```
using System;

namespace Tutlane
{
    // Declare Delegate
    public delegate void SampleDelegate(int a, int b);
    class MathOperations
```

```
    {
        public void Add(int a, int b)
        {
            Console.WriteLine("Add Result: {0}", a + b);
        }
        public void Subtract(int x, int y)
        {
            Console.WriteLine("Subtract Result: {0}", x - y);
        }
    }
    class Program
    {

        static void Main(string[] args)
        {
            Console.WriteLine("****Delegate Example****");
            MathOperations m = new MathOperations();
            // Instantiate delegate with add method
            SampleDelegate dlgt = m.Add;
            dlgt(10, 90);
            // Instantiate delegate with subtract method
            dlgt = m.Subtract;
            dlgt(10, 90);
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created a delegate called "**SampleDelegate**" and the delegate has been instantiated by using defined methods.

When you execute the above c# program, you will get the result as shown below.



This is how we can use delegates in our applications to call all the methods which are having the same signatures with a single object.

In c#, you can invoke the delegates same as method or by using the **Invoke** method like as shown below.

```
SampleDelegate dlgt = m.Add;
dlgt(10, 90);
```

```
// or
dlgt.Invoke(10, 90);
```

C# Delegate Types

In c#, we have a two different type of delegates available, those are

- Single cast Delegates
- Multicast Delegates

C# Single Cast Delegates

In c#, a delegate which points to a single method is called a single cast delegate and it is used to hold the reference of a single method like as explained in the above example.

C# Multicast Delegates

In c#, a delegate that points to multiple methods is called a multicast delegate and it is used to hold the reference of multiple methods with a single delegate.

By using `"+"` operator, we can add the multiple method references to the delegate object. Same way, by using `"-"` operator we can remove the method references from the delegate object.

In c#, Multicast delegates will work with only the methods that are having `void` as return type. In case, if we want to create a multicast delegate with the return type, then we will get a return type of the last method in the invoking list.

C# Multicast Delegate Example

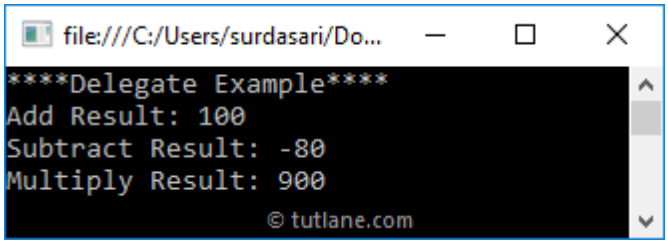
Following is the example of implementing a multicast delegate to hold the reference of multiple methods with `"+"` operator in c# programming language.

```
using System;

namespace Tutlane
{
    // Declare Delegate
    public delegate void SampleDelegate(int a, int b);
    class MathOperations
    {
        public void Add(int a, int b)
        {
```

```
            Console.WriteLine("Add Result: {0}", a + b);
        }
        public void Subtract(int x, int y)
        {
            Console.WriteLine("Subtract Result: {0}", x - y);
        }
        public void Multiply(int x, int y)
        {
            Console.WriteLine("Multiply Result: {0}", x * y);
        }
    }
    class Program
    {

        static void Main(string[] args)
        {
            Console.WriteLine("****Delegate Example****");
            MathOperations m = new MathOperations();
            // Instantiate delegate with add method
            SampleDelegate dlgt = m.Add;
            dlgt += m.Subtract;
            dlgt += m.Multiply;
            dlgt(10, 90);
            Console.ReadLine();
        }
    }
}
```

If you observe above example, we created a delegate called "**SampleDelegate**" and holding the reference of multiple methods using `"+"` operator. Now, our delegate become a multicast delegate and invoking a dlgt instance will invoke all the methods sequentially.

When you execute the above c# program, you will get the result like as shown below.



This is how we can use multicast delegates to hold the reference of multiple methods based on our requirements.

flyghar.com
noteghar.com

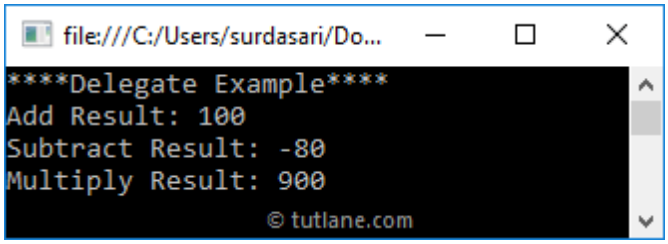
C# Pass Method as Parameter using Delegate

In c#, by using delegate we can pass methods as the parameter. Following is the example of sending methods as a parameter using a delegate.

```
using System;

namespace Tutlane
{
    // Declare Delegate
    public delegate void SampleDelegate(int a, int b);
    class MathOperations
    {
        public void Add(int a, int b)
        {
            Console.WriteLine("Add Result: {0}", a + b);
        }
        public void Subtract(int x, int y)
        {
            Console.WriteLine("Subtract Result: {0}", x - y);
        }
        public void Multiply(int x, int y)
        {
            Console.WriteLine("Multiply Result: {0}", x * y);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("****Delegate Example****");
            MathOperations m = new MathOperations();
            SampleMethod(m.Add, 10, 90);
            SampleMethod(m.Subtract, 10, 90);
            SampleMethod(m.Multiply, 10, 90);
            Console.ReadLine();
        }
        static void SampleMethod(SampleDelegate dlgt, int a, int b)
        {
            dlgt(a, b);
        }
    }
}
```

If you observe the above example, we created a "**SampleMethod**" with a **delegate** as a parameter type. By declaring like this, you can pass the method as a parameter to the newly created method (**SampleMethod**).

When you execute the above c# program, you will get the result as shown below.



flyghar.com

C# Delegates Overview

The following are the important properties of delegate in c# programming language.

- We need to use a `delegate` keyword to define delegates.
- In c#, delegates are used to hold the reference of one or more methods which is having the same signature as delegate.
- In c#, delegates are like function pointers in C++ but these are type-safe and secure.
- By using delegates, you can pass methods as a parameter to the other methods.
- In c#, we can invoke delegates as normal methods or by using **Invoke** property.
- By using `"+"` operator, we can add multiple methods to delegates.
- By using delegates, we can call multiple methods with a single event.

noteghar.com

C# Func Delegate

In c#, **Func** is a built-in generic delegate. It is useful to hold the reference of one or more methods with the same method signature without declaring any custom delegate object.

In previous chapters, we learned about delegates and will be used as shown following to hold the reference of methods with the same signature.

flyghar.com

```
using System;

namespace TutlaneExamples
{
   // Declare Delegate
   public delegate int SampleDelegate(int a, int b);
   class Program
   {
      static void Main(string[] args)
      {
         int result = 0;
         SampleDelegate dlgt = Add;
         result = dlgt(10, 90);
         Console.WriteLine("Add Result: {0}", result);
         dlgt = Subtract;
         result = dlgt(10, 90);
         Console.WriteLine("Subtract Result: {0}", result);
         Console.ReadLine();
      }
      public static int Add(int a, int b)
      {
         return a + b;
      }
      public static int Subtract(int x, int y)
      {
         return x - y;
      }
   }
}
```

noteghar.com

If you observe the above example, we created a delegate object called "**SampleDelegate**" to hold the reference of **Add** & **Subtract** methods.

When we execute the above example, we will get the result as shown below.

```
Add Result: 100
Subtract Result: -80
```

To avoid the declaration of custom delegate object (**SampleDelegate**), as we defined in the above example, the generic built-in delegates such as **Func**, **Action**, **Predicate** have been introduced in C# 3.0.

C# 3.0 onwards, the **Func** delegate will be available automatically with the **System** namespace, and it will accept zero or more (**16**) input parameters and one output parameter.

C# Func Syntax

Following is the syntax of declaring a **Func** delegate with one input parameter and one output parameter in c#.

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

Here, the last parameter in the angle bracket `< >` will be considered as an output parameter (return type), and the remaining all are considered as input parameters.

Parameter	Description
TResult	It represents the type of return value of the method that the delegate encapsulates.
T	It represents the type of parameter of the method that the delegate encapsulates.

If we want to create a **Func** delegate with two input parameters and return type (output parameter), that would be as shown below.

```
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
```

Like this, a **Func** delegate can include **0** to **16** input parameters of different types and one output parameter for the result.

C# Func Example

Following is the example of defining the **Func** delegate to hold the reference of one or more methods which is having the same method signature.

```
using System;

namespace TutlaneExamples
{
```

```
   class Program
   {
      static void Main(string[] args)
      {
         int result = 0;
         Func<int, int, int> dlgt = Add;
         result = dlgt(10, 90);
         Console.WriteLine("Add Result: {0}", result);
         dlgt = Subtract;
         result = dlgt(10, 90);
         Console.WriteLine("Subtract Result: {0}", result);
         Console.ReadLine();
      }
      public static int Add(int a, int b)
      {
         return a + b;
      }
      public static int Subtract(int x, int y)
      {
         return x - y;
      }
   }
}
```

If you observe the above example, we created a **Func** delegate object (**dlgt**) with two input parameters (**int**) and one output parameter (**int**) to return an **int** value.

Here, the declaration of **Func<int, int, int> dlgt** is same as the **SampleDelegate** object in the previous example.

When we execute the above example, we will get the result as shown below.

```
Add Result: 100
Subtract Result: -80
```

Every time while creating the **Func** delegate, we must need to remember that we can include 0 to 16 input parameters of different types, and that is optional, but we must need to include one output parameter for the return type.

Following is the example of creating a **Func** delegate with zero (**0**) input parameters and one output parameter.

```
Func<int> dlgt
```

C# Func with Anonymous Method

In c#, we can assign the anonymous method directly to the **Func** delegate by using the delegate keyword like as shown below.

```
static void Main(string[] args)
{
    Func<int, int, int> dlgt = delegate(int x, int y)
    {
       return x + y;
    };
    int i = dlgt(10, 90);
    Console.WriteLine("Result: {0}", i); // Result: 100
}
```

If you observe the above code, we assigned an anonymous method directly to the **Func** delegate object (**dlgt**) using the delegate keyword.

C# Func with Lambda Expressions

In c#, we can also use a **Func** delegate with lambda expressions. The lambda expressions are the shorthand way of declaring the anonymous method.

```
static void Main(string[] args)
{
   Func<int, int, int> dlgt = (x, y) => x + y;
   int i = dlgt(10, 90);
   Console.WriteLine("Result: {0}", i); // Result: 100
}
```

C# Func Overview

Following are the important points which we need to remember about **Func** delegate in c#.

- In c#, **Func** is a built-in generic delegate, and it is useful to hold the reference of one or more methods which is having the same method signature.
- Func delegate is introduced in C# 3.0, and it is available with the **System** namespace.
- While creating a **Func** delegate, we can include **0** to **16** input parameters of different types and one output parameter for the result.
- While creating a **Func** delegate, specifying input parameters is optional, but we must need to include one output parameter for the return type.
- We can use **Func** delegate in anonymous methods and Lambda expressions.

noteghar.com
flyghar.com

C# Action Delegate

In c#, **Action** is a built-in generic delegate same as Func delegate to hold the reference of one or more methods, but the only difference is the Action delegate will not return any value.

In previous chapters, we learned about delegates and will be used as shown following to hold the reference of methods with the same signature.

flyghar.com

```
using System;

namespace TutlaneExamples
{
   // Declare Delegate
   public delegate void SampleDelegate(int a, int b);
   class Program
   {
      static void Main(string[] args)
      {
         SampleDelegate dlgt = Add;
         dlgt(10, 90);
         dlgt = Subtract;
         dlgt(10, 90);
         Console.ReadLine();
      }
      public static void Add(int a, int b)
      {
         Console.WriteLine("Add Result: {0}", a + b);
      }
      public static void Subtract(int x, int y)
      {
         Console.WriteLine("Subtract Result: {0}", x - y);
      }
   }
}
```

If you observe the above example, we created a delegate object called "**SampleDelegate**" to hold the reference of Add & Subtract methods.

noteghar.com

When we execute the above example, we will get the result as shown below.

```
Add Result: 100
Subtract Result: -80
```

To avoid the declaration of custom delegate object (**SampleDelegate**), as we defined in the above example, the generic built-in delegates such as Func, **Action**, **Predicate** have been introduced in C# 3.0.

From C# 3.0 onwards, the **Action** delegate will available automatically with the **System** namespace, and it will accept zero or more (**16**) input parameters and does not return a value.

flyghar.com

C# Action Delegate Syntax

Following is the syntax of declaring an **Action** delegate with one input parameter in c#.

```
public delegate void Action<in T>(T arg);
```

Here, the parameters in the angle bracket `< >` will be considered as input parameters and `void` is the return type.

If we want to create an **Action** delegate with two input parameters, that would be as shown below.

```
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
```

Like this, an **Action** delegate can include **0** to **16** input parameters of different types based on our requirements.

C# Action Delegate Example

Following is the example of defining the **Action** delegate to hold the reference of one or more methods which is having the same method signature.

```
using System;

namespace TutlaneExamples
{
   class Program
   {
      static void Main(string[] args)
      {
         Action<int, int> dlgt = Add;
         dlgt(10, 90);
         dlgt = Subtract;
         dlgt(10, 90);
         Console.ReadLine();
      }
      public static void Add(int a, int b)
      {
         Console.WriteLine("Add Result: {0}", a + b);
```

noteghar.com

```
        }
        public static void Subtract(int x, int y)
        {
            Console.WriteLine("Subtract Result: {0}", x - y);
        }
    }
}
```

If you observe the above example, we created an **Action** delegate object (**dlgt**) with two input parameters (**int**) and assigned the methods directly to the delegate object.

Here, the declaration of `Action<int, int> dlgt` is same as the `SampleDelegate` object in the previous example.

When we execute the above example, we will get the result as shown below.

```
Add Result: 100
Subtract Result: -80
```

While creating the **Action** delegate, we must remember that we can include 0 to 16 input parameters of different types and optional.

In c#, we can also initialize an **Action** delegate using `new` keyword with required input parameters like as shown below.

```
Action<int, int> dlgt = new Action<int,int>(Add);
```

C# Action Delegate with Anonymous Method

In c#, we can assign the anonymous method directly to the **Action** delegate by using the delegate keyword like as shown below.

```
static void Main(string[] args)
{
    Action<int, int> dlgt = delegate(int x, int y)
    {
        Console.WriteLine("Result: {0}", x + y);
    };
    dlgt(10, 90); // Result: 100
}
```

If you observe the above code, we assigned an anonymous method directly to the **Action** delegate object (**dlgt**) using the delegate keyword.

C# Acton Delegate with Lambda Expressions

In c#, we can also use Action delegate with lambda expressions. The lambda expressions are the shorthand way of declaring the anonymous method.

```
static void Main(string[] args)
{
    Action<int, int> dlgt = (x, y) =>
    {
       Console.WriteLine("Result: {0}", x + y);
    };
    dlgt(10, 90); // Result: 100
}
```

flyghar.com

C# Action Delegate Overview

Following are the important points which we need to remember about Action delegate in c#.

- In c#, **Action** is a built-in generic delegate like Func delegate, but the only difference is that the Action delegate will not return any value.
- While creating an **Action** delegate, we can include **0** to **16** input parameters of different types.
- We can use **Action** delegate in anonymous methods and lambda expressions.

noteghar.com

C# Predicate Delegate

In c#, **Predicate** is a built-in generic delegate, and it is useful to validate whether the input parameter meets the specified condition or not, and it's same as Func and Action delegates to hold the reference of one or more methods.

The Predicate delegate can hold only the methods that take only one input parameter and returns a Boolean type value, either **true** or **false**.

From C# 3.0 onwards, the **Predicate** delegate will available automatically with the **System** namespace, and it will accept only one input parameter and returns a Boolean value, either **true** or **false**.

C# Predicate Delegate Syntax

Following is the syntax of declaring the **Predicate** delegate with one input parameter in c#.

```
public delegate bool Predicate<in T>(T arg);
```

Here, the parameter in the angle bracket `< >` will be considered as the input parameter, and the **bool** is the return type.

C# Predicate Delegate Example

Following is the example of defining the **Predicate** delegate to hold the reference of one or more methods which is having the same method signature.

```
using System;

namespace TutlaneExamples
{
   class Program
   {
      static void Main(string[] args)
      {
        Predicate<int> dlgt = IsGreaterthanZero;
        bool result = dlgt(10);
        Console.WriteLine("Result: {0}", result);
        Console.ReadLine();
      }
      public static bool IsGreaterthanZero(int a)
      {
        bool i = false;
        if (a > 0)
```

noteghar.com

```
        {
          i = true;
        }
        return i;
      }
   }
}
```

If you observe the above example, we created a **Predicate** delegate object (**dlgt**) with one input parameter (**int**) and assigned the method (**IsGreaterthanZero**) directly to the delegate object.

Here, the **IsGreaterthanZero** method is accepting only one input parameter and validating whether the input parameter is greater than zero or not. If it is greater than zero, it will return the value **true**; otherwise, it will return **false**.

When we execute the above example, we will get the result as shown below.

```
Result: True
```

Every time while creating the **predicate** delegate, we must need to remember that we can include only one input parameter and the return type must be Boolean.

C# Predicate Delegate with Anonymous Method

In c#, we can assign the anonymous method directly to the Predicate delegate by using the delegate keyword like as shown below.

```
static void Main(string[] args)
{
   Predicate<int> dlgt = delegate(int x)
   {
      bool i = false;
      if (x > 0)
      {
         i = true;
      }
      return i;
   };
   bool result = dlgt(10); // Result: True
}
```

If you observe the above code, we assigned an anonymous method directly to the **Predicate** delegate object (**dlgt**) using the delegate keyword.

C# Predicate Delegate with Lambda Expressions

In c#, we can also use Predicate delegate with lambda expressions. The lambda expressions are the shorthand way for declaring the anonymous methods.

```
static void Main(string[] args)
{
   Predicate<int> dlgt = (x) =>
   {
      bool i = false;
      if (x > 0)
      {
        i = true;
      }
      return i;
   };
   bool result = dlgt(10); // Result: True
}
```

flyghar.com

C# Predicate Delegate Overview

Following are the important points which we need to remember about Predicate delegate in c#.

- In c#, Predicate is a built-in generic delegate, and it is useful to validate whether the input parameter meets the specified condition or not, and it's same as Func and Action delegates to hold the reference of one or more methods.
- The Predicate delegate can hold only the methods that take only one input parameter and returns a Boolean type value, either **true** or **false**.
- While creating the **predicate** delegate, we must remember that we can include only one input parameter and the return type must be Boolean.
- We can use Predicate delegate in anonymous methods and lambda expressions.

noteghar.com

C# Events

In c#, the **event** is a message which is sent by an object to indicate that particular *action* is going to happen. The *action* could be caused either by a button click, mouse movements or by some other programming logic. The object that raises an event is called an *event sender*.

In simple words, we can say that **events** are used to signal user actions such as button click, mouse over, menu selection, etc. in the user interface of windows and web applications.

Generally, an event is nothing but something special that is going to happen. For example, Microsoft or Google will conduct Build / Keynote events to announce their new features or products and they will **notify** about their events to the users either through an email or advertisements. Here Google or Microsoft is a **publisher** who conducts (or **raises**) an **event** and users become a **subscriber** to attend (or **handle**) the event.

In c#, **events** will follow the same pattern and it will contain a *publisher*, *subscriber*, *notification,* and *handler*. The events will enable a class or object to notify other classes or objects when something special happens. The class that sends (or raises) an event is called the **publisher** and the class that receives (or **handle**) an event is called **subscriber**.

In c#, the **publisher** will determine when an event is raised and the **subscribers** will determine what action can be taken in response to the event. An event in c# will have multiple **subscribers** and the events that have no subscribers will never be raised. The **subscriber** can handle multiple events from multiple **publishers**.

Now we will see how to create and use events in c# to notify other classes or objects when something of interest happens.

C# Events Declaration

In c#, events are the encapsulated delegates so first, we need to define a delegate before we declare an event inside of a class by using `event` keyword.

Following is the example of declaring an event using `event` keyword in c# programming language.

```
// Declare the delegate
public delegate void SampleDelegate();
//Declare an event
public event SampleDelegate SampleEvent;
```

If you observe above event declaration, first we declared a delegate (**SampleDelegate**) and we made that delegate as an event (**SampleEvent**) by using `event` keyword.

In c#, to raise an event we need to invoke the event delegate and subscribe to an event using `+=` operator. In case, if you want to unsubscribe from an event, then use `-=` operator.

To respond to an event, we need to define an event **handler** method in the event receiver and this method signature must match with the signature of event delegate. In the event handler, you can perform actions that are required whenever the event is raised, such as getting the user input after a click on the button.

flyghar.com

C# Events Example

Following is the example of declaring and raising an event using underlying delegate type in c# programming language.

```
using System;

namespace Tutlane
{
    class Maths
    {
        // Declare the delegate
        public delegate void SampleDelegate();
        //Declare an event
        public event SampleDelegate SampleEvent;
        public void Add(int a, int b)
        {
            // Calling event delegate to check subscription
            if (SampleEvent != null)
            {
                // Raise the event by using () operator
                SampleEvent();
                Console.WriteLine("Add Result: {0}", a + b);
            }
            else
            {
                Console.WriteLine("Not Subscribed to Event");
            }
        }
        public void Subtract(int x, int y)
        {
            // Calling event delegate to check subscription
            if (SampleEvent != null)
            {
                // Raise the event by using () operator
                SampleEvent();
                Console.WriteLine("Subtract Result: {0}", x - y);
            }
            else
            {
                Console.WriteLine("Not Subscribed to Event");
```

noteghar.com

```
            }
        }
    }
    class Operations
    {
        Maths m;
        public int a { get; set; }
        public int b { get; set; }
        public Operations(int x, int y)
        {
            m = new Maths();
            // Subscribe to SampleEvent event
            m.SampleEvent += SampleEventHandler;
            a = x;
            b = y;
        }
        // SampleEvent Handler
        public void SampleEventHandler()
        {
            Console.WriteLine("SampleEvent Handler: Calling Method");
        }
        public void AddOperation()
        {
            m.Add(a, b);
        }
        public void SubOperation()
        {
            m.Subtract(a, b);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("****Events Example****");
            Operations op = new Operations(10, 20);
            op.AddOperation();
            op.SubOperation();
            Console.ReadLine();
        }
    }
}
```

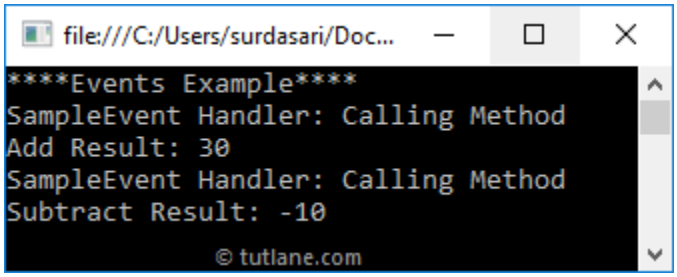
flyghar.com
noteghar.com

If you observe the above example, we created a **Maths** class (**publisher**) and **Operations** class (**subscriber**) with required methods and **SampleEvent** event. In **Maths** class methods, we are checking the condition like whether the calling class subscribed to **SampleEvent** event or not to execute the required functionality.

In the **Operations** class, we subscribed to **SampleEvent** event using `+=` operator and mentioned the name of handler (**SampleEventHandler**) to perform the required operations when an event is raised.

If you observe the handler method (**SampleEventHandler**) in the **Operations** class, it's having the same signature of our delegate (**SampleDelegate**) in **Maths** class.

When you execute the above c# program, you will get the result as shown below.



C# Events Overview

The following are the important points which we need to remember about events in c# programming language.

- In c#, events are used to enable a class or object to notify other classes or objects about the action that is going to happen.
- To declare an event, we need to use `event` keyword with delegate type.
- Before raising an event, we need to check whether an event is subscribed or not.
- By using `+=` operator, we can subscribe to an event and by using `-=` operator we can unsubscribe from an event.
- To raise an event, we need to invoke the event delegate.
- To respond to an event, we can define an event handler method in the event receiver and the handler method must have the same signature of delegate in the event.
- In c#, to raise events there must be subscribers otherwise they won't be raised.
- In c#, an event can have multiple subscribers and a subscriber can handle multiple events from multiple publishers.
- In case, an event has multiple subscribers, then event handlers are invoked synchronously when an event raised.
- In c#, the **publisher** determines when an event is raised, the **subscriber** determines what action is taken in response to the event.
- In .NET Framework, events are based on **EventHandler** delegate and an **EventArgs** base class.

flyghar.com
noteghar.com

flyghar.com

noteghar.com

AsNoTracking

Introduction

Tracking behavior controls whether to keep track or not of information about an entity instance when it is changed. If an entity is tracked, any changes detected in the entity will be persisted to the database during SaveChanges().

In the following example, the change to the customer address will be detected and persisted to the database during `SaveChanges()`.

```
using (var context = new MyContext())
{
    var customer = context.Customers
        .Where(c => c.CustomerId == 2)
        .FirstOrDefault();

     customer.Address = "43 rue St. Laurent";
     context.SaveChanges();
}
```

No-tracking

No tracking query executes quickly because there is no need to setup change tracking information. It is useful when the results are used in a read-only scenario.

You can convert a query to a no-tracking query by using `AsNoTracking()` method.

AsNoTracking

The `AsNoTracking()` method returns a new query where the change tracker will not track any of the entities that are returned. If the entity instances are modified, this will not be detected by the change tracker, and SaveChanges() will not persist those changes to the database.

```
using (var context = new MyContext())
{
    var customers = context.Customers
        .AsNoTracking().ToList();
}
```

You can also change the default tracking behavior at the context instance level.

```
using (var context = new MyContext())
{
```

```
    context.ChangeTracker.QueryTrackingBehavior =
QueryTrackingBehavior.NoTracking;
    var customers = context.Customers.ToList();
}
```

flyghar.com

Basic Query

Entity Framework Core uses Language Integrate Query (LINQ) to query data from the database.

- LINQ allows you to use C# (or your .NET language of choice) to write strongly typed queries based on your derived context and entity classes.
- Querying in Entity Framework Core remains the same as in EF 6 to load entities from the database.
- It provides more optimized SQL queries and the ability to include C#/VB.NET functions into LINQ-to-Entities queries.

Let's say we have a simple model which contains three entities.

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public virtual List<Invoice> Invoices { get; set; }
}
```

Load All data

The following example load all the data from `Customers` table.

```
using (var context = new MyContext())
{
    var customers = context.Customers.ToList();
}
```

noteghar.com

Load a Single Entity

The following example load a single record from `Customers` table based on `CustomerId`.

```
using (var context = new MyContext())
{
    var customers = context.Customers
```

```
        .Single(c => c.CustomerId == 1);
}
```

Filter Data

The following example loads all customers with `FirstName` **Mark**.

```
using (var context = new MyContext())
{
    var customers = context.Customers
        .Where(c => c.FirstName == "Mark")
        .ToList();
}
```

flyghar.com

Global Filter

Introduction

Entity Framework Core 2.0 introduces global query filters that can be applied to entities when a model is created. It makes it easier to build multi-tenant applications and support soft deleting of entities.

- It allows us to specify a filter in the model level that is automatically applied to all queries that are executed in the context of the specified type.
- It means that entity framework automatically adds the filter in the where clause before executing the LINQ queries.
- Usually, Global query filters are applied in `OnModelCreating` method of context.

Here is a simple model which contains only one entity i.e., Customer

noteghar.com

```
public class Customer
{
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public bool IsDeleted { get; set; }
}
```

Global query filters are defined when database context builds the model. So we need to configure the global query filter in `OnModelCreating` method of context class, using `HasQueryFilter` method, and we can apply the global filter on the entity type.

```
public class MyContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Data Source=(localdb)\ProjectsV13;Initial
Catalog=CustomerDB;");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Customer>().HasQueryFilter(c => !c.IsDeleted);
    }
}
```

The expression passed in HasQueryFilter method is automatically applied to any LINQ queries for Customer Type. We have four records in the Customers table in the database.

One record is soft deleted already, and the `IsDeleted` column contains a **True** value for that record. Now if we retrieve all the all the customers from the database using the LINQ query.

```
using (var context = new MyContext())
{
    var customers = context.Customers.ToList();
}
```

We will get only three records, but in the database, we have four records, Because the global filter has filtered records and returns only those records whose `IsDeleted` column contains **False** value.

In some cases, we do not need these filters to be applied, we can also disable filter for individual LINQ queries using the `IgnoreQueryFilters()` method.

```
using (var context = new MyContext())
{
    var customers = context.Customers
        .IgnoreQueryFilters().ToList();
}
```

Now the filter won't be applied, and you will get all the records.

Include & ThenInclude

Include

The `Include` method specifies the related objects to include in the query results. It can be used to retrieve some information from the database and also want to include related entities. Now let's say we have a simple model which contains three entities.

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public virtual List<Invoice> Invoices { get; set; }
}

public class Invoice
{
    public int InvoiceId { get; set; }
    public DateTime Date { get; set; }
    public int CustomerId { get; set; }
    [ForeignKey("CustomerId")]
    public Customer Customer { get; set; }
    public List<InvoiceItem> Items { get; set; }
}

public class InvoiceItem
{
    public int InvoiceItemId { get; set; }
    public int InvoiceId { get; set; }
    public string Code { get; set; }
    [ForeignKey("InvoiceId")]
    public virtual Invoice Invoice { get; set; }
}
```

Now to retrieve all customers and their related invoices we have to use the `Include` Method.

```
using (var context = new MyContext())
{
    var customers = context.Customers
        .Include(c => c.Invoices)
        .ToList();
}
```

ThenInclude

The `Include` method works quite well for Lists on objects, but what if there is a need for multiple levels of depth. For example, Customer contains a list of invoices and each invoice then contains a list of items.

EF Core has a new extension method ThenInclude(). You can drill down thru relationships to include multiple levels of related data using the ThenInclude method.

```
using (var context = new MyContext())
{
    var customers = context.Customers
        .Include(i => i.Invoices)
            .ThenInclude(it => it.Items)
        .ToList();
}
```

Projection

Introduction

Projection is a way of translating a full entity into a C# class with a subset of those properties. It is used to create a query that selects from a set of entities in your model but returns results that are of a different type.

- Projection queries improve the efficiency of your application because only specific fields are retrieved from your database.
- When you have the data, you can project or filter it as you want the data before output.
- You can use `Select()` LINQ statement for projection.

Anonymous Type

Anonymous types provide an easy way to create a new type without initializing them. Projection queries load the data into this Anonymous Type.

```
var customer = context.Customers
    .Select(cust => new
    {
        Id = cust.CustomerId,
        FullName = cust.FirstName + cust.LastName,
        InvoiceList = cust.Invoices
    }).ToList();
```

In this example, the customer data is projected to anonymous type which contains `Id`, `FullName`, and `InvoiceList`.

Concrete Type

You can also write Projection query to return a Concrete Type, and to do so we will have to create a custom class first, and it must have all properties, which we want to return from the table.

```
public class CustomerData
{
    public int Id { get; set; }
    public string FullName { get; set; }
    public List<Invoice> InvoiceList { get; set; }
}
```

Now change the select clause in the query to map the result to the CustomerData.

```
List<CustomerData> customers = context.Customers
    .Select(cust => new CustomerData
    {
        Id = cust.CustomerId,
        FullName = cust.FirstName + cust.LastName,
        InvoiceList = cust.Invoices
    }).ToList();
```

Loading (Eager & Lazy)

Entity Framework Core allows you to use the navigation properties in your model to load related entities. There are three common patterns used to load related data.

- Eager loading
- Explicit loading
- Lazy loading

We have a simple model which contains two entities.

```
public class Customer
{
    public int CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public virtual List<Invoice> Invoices { get; set; }
}

public class Invoice
{
    public int Id { get; set; }
    public DateTime Date { get; set; }
    public int CustomerId { get; set; }
    [ForeignKey("CustomerId")]
```

```
    public virtual Customer Customer { get; set; }
}
```

Eager loading

In eager loading, the related data is loaded from the database as part of the initial query using Include & ThenInclude methods.

- The Include method specifies the related objects to include in the query results.
- It can be used to retrieve some information from the database and also want to include related entities.

Now to retrieve all customers and their related invoices we have to use the `Include` Method.

```
using (var context = new MyContext())
{
    var customers = context.Customers
        .Include(c => c.Invoices)
        .ToList();
}
```

Explicit loading

In explicit loading, the related data is explicitly loaded from the database at a later time. You can explicitly load a navigation property via the DbContext.Entry() method.

```
using (var context = new MyContext())
{
    var customer = context.Customers
        .Single(c => c.CustomerId == 1);

    context.Entry(customer)
        .Collection(c => c.Invoices)
        .Load();
}
```

You can get a LINQ query that represents the contents of a navigation property and then filters which related entities are loaded into memory.

```
using (var context = new MyContext())
{
    var customer = context.Customers
        .Single(c => c.CustomerId == 1);

    context.Entry(customer)
        .Collection(c => c.Invoices)
        .Query()
        .Where(i => i.Date >= new DateTime(2018, 1, 1))
        .ToList();
}
```

noteghar.com

Lazy loading

In lazy loading, the related data is transparently loaded from the database when the navigation property is accessed. To use lazy-loading, the simple way is to install the Microsoft.EntityFrameworkCore.Proxies and enable it by calling `UseLazyLoadingProxies()` in `OnConfiguring` method on your data context.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(@"Data Source=(localdb)\ProjectsV13;Initial
Catalog=CustomerDB;");
}
```

EF Core will enable lazy-loading for any navigation property that is virtual and in a class that can be inherited. So in our model the `Invoice.Customer` and `Customer.Invoices` navigation properties will be lazy-loaded.

```
using (var context = new MyContext())
{
    var customers = context.Customers.ToList();
}
```

noteghar.com

C# Exception

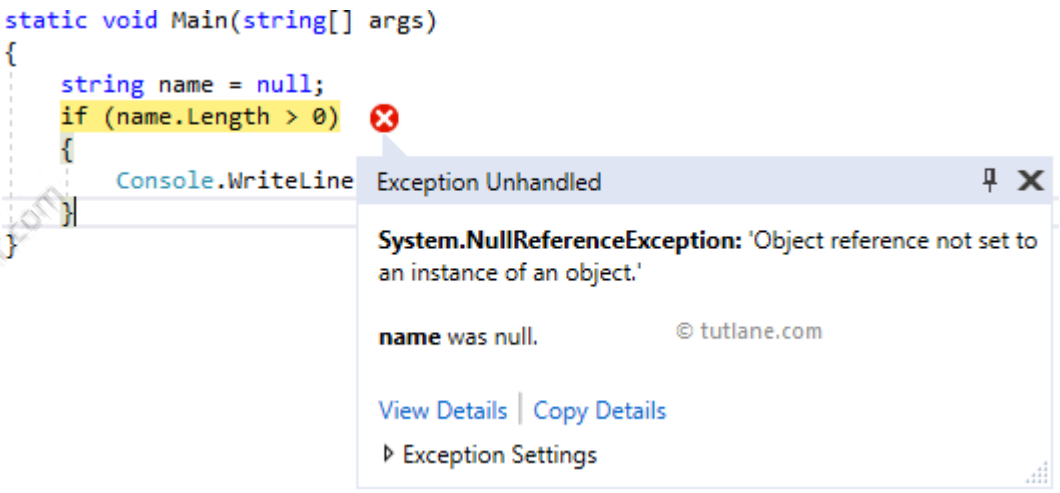
In c#, **Exception** is an unexpected event or an error that may occur during the execution of the program and it will provide necessary information about the error which occurred in our application.

Generally, the exceptions are thrown by .NET framework CLR (Common Language Runtime) or by the code in our program. Following is the example which will throw an exception when we execute it in c#.

```
using System;

namespace TutlaneExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = null;
            // Null Reference Exception Error
            if (name.Length > 0)
            {
                Console.WriteLine("Name: "+name);
            }
        }
    }
}
```

If you observe the example, we assigned a **null** value to the **name** variable and trying to validate the length of the string variable.

When we execute the above c# program, we will get an exception like as shown below because the variable (**name**) value is **null**.

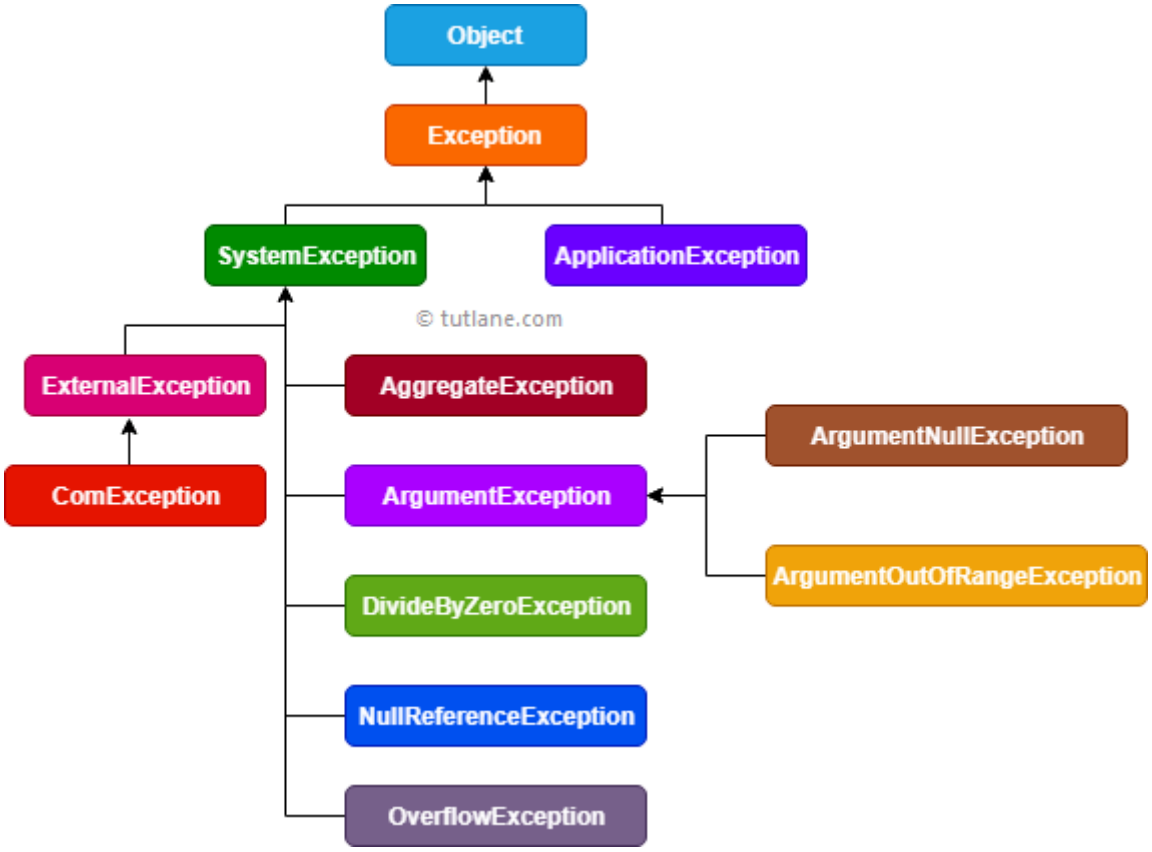


C# has provided necessary built-in exception classes to handle all the exceptions in our applications and all exception classes are derived from the base class called `Exception`. The `Exception` class will identify the type of exception and provide the properties that have details about the exception.

In c#, the `Exception` class further classified into two other classes called **SystemException** and **ApplicationException** to handle the exceptions. Generally, the **SystemException** is a base class for all CLR generated errors whereas the **ApplicationException** is a base class for all application-related exceptions.

flyghar.com

The following diagram will illustrate more details about the exception classes in c#.



If you observe the above diagram, the **SystemException** is a base class for all CLR exceptions and these will occur during the execution of the program. The **ApplicationException** is a base class for application-related exceptions and we can derive this class to create our own exception classes.

noteghar.com

C# Exception Classes

The following table lists some of the important exception classes available in c#.

Exception	Description
AccessViolationException	This exception will raise when there is an attempt to read or write protected memory.
ArgumentException	This exception will be thrown when one of the arguments provided to a method is not valid.
ArgumentNullException	This exception will occur when a null argument is passed to a method that does not accept it as a valid argument.
ArgumentOutOfRangeException	This exception will occur when the value of an argument is outside the allowable range of values.
ArithmeticException	This exception will occur for errors in arithmetic, casting, or conversion operation.
DivideByZeroException	This exception will occur when we try to divide an integral or decimal value by zero.
FormatException	This will occur when the format of an argument is invalid.
IndexOutOfRangeException	This will occur when we try to access an element of an array or collection with an index that is outside of its bounds.
InvalidCastException	This exception occurs for invalid casting or explicit conversion.
NullReferenceException	This exception will occur when we try to reference an object which is of NULL type.
OutOfMemoryException	This will occur when the program is not having enough memory to execute the code.
OverflowException	This will occur when arithmetic, casting, or conversion operation results in an overflow.

Exception	Description
StackOverflowException	This exception will occur when the execution stack overflows.
TimeoutException	This will occur when the time allotted for a process or operation has expired.

flyghar.com

As we discussed, every exception class will derive from the base `Exception` class and provide the following important properties to get the details of an exception.

Property	Description
Data	This property will hold arbitrary data in key-value pairs.
HelpLink	It will hold a help file URL that provides detailed information about the cause of the exception.
InnerException	This property will provide information about the series of exceptions that might have occurred.
Message	It will provide details about the cause of an exception.
Source	It will set or get the name of an application or object that causes the error.
StackTrace	It provides a stack trace to determine where an error occurred.

Generally, in c# we can handle all the exceptions by using **try-catch** blocks. In the next chapter, we will learn how to handle exceptions in c# with examples.

noteghar.com

C# Exception Handling

As we learned in the previous c# exception topic, exceptions are generated by CLR (common language runtime) or application code. To handle runtime or unexpected errors in applications, c# has provided a built-in exception handling support by using **try**, **catch** and **finally** blocks.

In c#, when an exception is thrown, then the CLR (common language runtime) will look for the `catch` block that handles the exception. In case, if the currently executing method does not contain such a `catch` block, then the CLR will display an unhandled exception message to the user and stops the execution of the program.

Following is the syntax of handling errors in c# using **try**, **catch** and **finally** blocks.

```
try
{
      // code that may cause exception
}
catch (Exception ex)
{
     // exception handling
}
finally
{
     // clean up resources
}
```

As per the above syntax, the `try` block will contain the guarded code that may cause an exception so that if any errors occurred in our code, then immediately the code execution will be moved to `catch` block to handle those exceptions. In case, if no exception occurred in the `try` block, then the `catch` block will be skipped and the execution will be moved to `finally` block.

After completion of `try` or `try & catch` blocks, the `finally` block will always execute even if an exception occurred or not and it is useful to clean up or dispose of unmanaged objects based on the requirements.

In c#, the `try` block must be followed by `catch` or `finally` or both blocks otherwise we will get a compile-time error. In a `try-catch-finally` statement, only one `try & finally` blocks are allowed but we can use multiple `catch` blocks to handle different exception types.

C# Exception Handling Example

Following is the example of handling exceptions in c# using `try`, `catch` and `finally` blocks.

```
using System;

namespace TutlaneExamples
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            string name = null;
            try
            {
                if (name.Length > 0) // Exception will occur
                {
                    Console.WriteLine("Name: " + name);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception: {0}", ex.Message);
            }
            finally
            {
                Console.WriteLine("Finally Block.");
            }
            Console.ReadLine();
        }
    }
}
```

If you observe the above code, we used a `try`, `catch` and `finally` blocks to handle runtime or unexpected errors during the execution of the program. Here, we wrote a code that may throw an exception inside of `try` block and in **catch** block, we are handling the exception. As discussed, the `finally` block will execute after completion of `try` or `catch` block execution.

When you execute the above code, you will get the result as shown below.

```
Exception: Object reference not set to an instance of an object.
Finally Block.
```

Try Catch in C#

In c#, the **try-catch** statement is useful to handle unexpected or runtime exceptions which will occur during the execution of the program. The **try-catch** statement will contain a `try` block followed by one or more `catch` blocks to handle different exceptions.

In c#, whenever an exception occurred in the `try` block, then the CLR (common language runtime) will look for the `catch` block that handles an exception. In case, if the currently executing method does not contain such a `catch` block, then the CLR will display an unhandled exception message to the user and stops the execution of the program.

C# try-catch Syntax

Following is the syntax of handling errors in c# using **try**, **catch** blocks.

```
try
{
      // code that may cause exception
}
catch (Exception ex)
{
     // exception handling
}
```

As per the above syntax, the `try` block will contain the guarded code that may cause an exception so that if any errors occurred in our code, then immediately the code execution will be moved to `catch` block to handle those exceptions.

In a **try-catch** statement, the `catch` block will execute only when an exception occurred in the code that is guarded by `try` block.

C# try-catch Example

Following is the example of using a **try-catch** statement in c# to handle the exceptions.

```
using System;

namespace TutlaneExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = null;
            try
            {
                if (name.Length > 0) // Exception will occur
                {
                    Console.WriteLine("Name: " + name);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception: {0}", ex.Message);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe the above code, we used `try` and `catch` blocks to handle runtime or unexpected errors during execution of the program. Here, we wrote a code that may throw an exception inside of `try` block and in `catch` block we are handling the exception.

When we execute the above code, we will get the result as shown below.

```
Exception: Object reference not set to an instance of an object.
```

This is how we can use try-catch blocks to handle unexpected or runtime exceptions based on our requirements.

In c#, the `try` block must be followed by `catch` or `finally` or both blocks otherwise we will get a compile-time error.

tiyeghar.com

C# try with Multiple catch Blocks

In the above **try-catch** statement example, we used only single `catch` block with the `Exception` base class argument to handle all the exceptions.

In case, if we want to handle a different type of exceptions in different ways, then we can specify multiple catch blocks with different exception types in the same **try-catch** statement and this process called an exception filtering.

If we define multiple `catch` blocks with a `try` statement, then the first `catch` statement that can handle an exception will be executed, any following `catch` statements that are even more compatible to handle an exception will be ignored so the order of catch blocks must be always from most specific to least specific.

Following is the example of defining multiple catch blocks in a **try-catch** statement to handle different exceptions in different ways.

```
using System;

namespace TutlaneExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Enter x value:");
                int x = int.Parse(Console.ReadLine());
                Console.WriteLine("Enter y value:");
                int y = int.Parse(Console.ReadLine());
                Console.WriteLine(x / y);
            }
            catch (FormatException ex)
            {
                Console.WriteLine("Input string was not in a correct fo
rmat.");
            }
            catch (InvalidOperationException ex)
            {
```

noteghar.com

```
                Console.WriteLine("Not a valid numbers to perform opera
tion");
            }
            catch (DivideByZeroException ex)
            {
                Console.WriteLine("Cannot Divide By Zero");
            }
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we defined multiple **catch** blocks to handle different exception types so that we can easily get know what exact error it is and we can fix it easily.

When we execute the above example, we will get the result like as shown below.

```
Enter x value:
10
Enter y value:
0
Cannot Divide By Zero
```

This is how we can use multiple **catch** blocks with a single **try** block to handle different types of exceptions based on our requirements.

C# Invalid Catch Blocks

In c#, defining a parameterless **catch** block (catch{}) and **catch** block with an exception argument (catch (Exception ex) { }) are not allowed in the same **try-catch** statement because both will do the same thing.

Following is the example of defining an invalid `catch` blocks in the same **try-catch** statement.

```
try
{
// code that may cause an exception
}
catch{
// Exception Handling
}
catch (Exception ex) {
// Exception Handling
}
```

In case, if we use a parameterless **catch** block (catch{}) or **catch** block with an exception argument (catch (Exception ex) {}) with other **catch** blocks, then those parameters must be last otherwise we will get a compile time error.

Following is the invalid way of defining a multiple `catch` blocks with a parameterless `catch` block in the same **try-catch** statement.

```
try
{
// code that may cause an exception
}
catch {
// Exception Handling
}
catch (FormatException ex)
{
// Exception Handling
}
catch (InvalidOperationException ex)
{
// Exception Handling
}
```

Following are the few point which we needs to consider while defining a multiple **catch** blocks.

- Multiple **catch** blocks with the same exception type are not allowed.
- Defining a parameterless **catch** block (catch{}) and **catch** block with an exception argument (catch (Exception ex) { }) are not allowed in the same **try-catch** statement because both will do the same thing.
- While defining multiple **catch** blocks, the order of catch blocks must be always from most specific to least specific.
- A parameterless **catch** block (catch{}) or **catch** block with an exception argument (catch (Exception ex) {}) must be the last block if we use it with other **catch** blocks otherwise we will get a compile-time error.

C# Nested try-catch Statements

In c#, we can define a **try-catch** statement within another **try-catch** statement based on our requirements. In nested **try-catch** statements, if there isn't any inner `catch` block with an appropriate exception type, then the exception will flow to the outer `catch` block.

Following is the example of defining nested **try-catch** statements in c#.

```
using System;

namespace TutlaneExamples
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            try
            {
                try
                {
                    Console.WriteLine("Enter x value:");
                    int x = int.Parse(Console.ReadLine());
                    Console.WriteLine("Enter y value:");
                    int y = int.Parse(Console.ReadLine());
                    Console.WriteLine(x / y);
                }
                catch (FormatException ex)
                {
                    Console.WriteLine("Input string was not in a correc
t format.");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception: " + ex.Message);
            }
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we defined a **try-catch** statement within another **try-catch** statement. Here, if we get any exception other than **FormatException** in an inner **try-catch** statement, then the exception will flow to outer **catch** block until it finds an appropriate exception filter.

When we execute the above program, we will get the result like as shown below.

```
Enter x value:
10
Enter y value:
0
Exception: Attempted to divide by zero.
```

If you observe the above result, the inner **try-catch** statement has failed to handle the exception so the catch block in the outer **try-catch** statement has been handled the exception.

noteghar.com

C# try-catch Overview

The following are the important points that you need to remember about the try-catch statement.

- In try-catch statement, the `try` block will hold the code that may raise an exception and in the `catch` block, exception handling can be done.
- The try-catch statement will contain a `try` block followed by one or more `catch` blocks to handle different exceptions.
- In c#, whenever an exception occurred in the `try` block, then the CLR (common language runtime) will look for the appropriate `catch` block that handles an exception.
- In c#, we can use nested try-catch statements.
- In nested try-catch statements, if there isn't any inner `catch` block with an appropriate exception type, then the exception will be caught by an outer `catch` block.

flyghar.com

Try Catch Finally in C#

In c#, the **try-catch-finally** statement is useful to handle unexpected or runtime exceptions which will occur during the execution of the program.

In **try-catch-finally** statement, the `try` block is used to hold the code that may cause an exception, `catch` block to handle exceptions and `finally` block is used to clean up or release any resources that are allocated in a `try` block. For example, close database connections or any streams or files that are opened in `try` block.

In c#, the **finally** block will always come after `try` or `catch` blocks and the `finally` block will always execute even if an exception occurred or not and it is useful to clean up or dispose of unmanaged objects based on the requirements.

C# try catch finally Syntax

Following is the syntax of handling errors in c# using a **try-catch-finally** statement.

```
try
{
      // code that may cause exception
}
catch (Exception ex)
{
     // exception handling
}
finally
{
     // clean up resources
}
```

noteghar.com

As per the above syntax, the **try** block will contain the guarded code that may cause an exception so that if any errors occurred in our code, then immediately the code execution will be moved to **catch** block to handle those exceptions.

After completion of **try** or **try & catch** blocks, the **finally** block will always be executed even if an exception occurred or not and it is useful to clean up or release unmanaged resources.

In **try-catch-finally** statement, only one **try & finally** blocks are allowed but, we can use multiple catch blocks to handle different exception types.

flyghar.com

In c#, the **try** block must always be followed by **catch** or **finally** or both blocks otherwise we will get a compile-time error.

C# try-catch-finally Example

Following is the example of handling exceptions in c# using **try-catch-finally** statement.

```
using System;
using System.IO;

namespace TutlaneExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            string fpath = @"D:\Test.txt";
            StreamReader sr = new StreamReader(fpath);
            try
            {
                string txt;
                while ((txt = sr.ReadLine()) != null)
                {
                    Console.WriteLine(txt);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception: {0}", ex.Message);
            }
            finally
            {
                if (sr != null)
                {
                    sr.Close();
                }
            }
            Console.ReadLine();
        }
    }
```

noteghar.com

```
}
```

If you observe the above code, we used **try**, **catch** and **finally** blocks to handle runtime or unexpected errors during execution of the program. Here, we wrote a code that may throw an exception inside of `try` block and in `catch` block we are handling the exception. As discussed, the `finally` block will execute after completion of `try` or `catch` block execution and release the required resources.

This is how we can use **try-catch-finally** blocks to handle unexpected or runtime exceptions based on our requirements.

tiyeghar.com

C# Throw Keyword

In c#, the **throw** is a keyword and it is useful to throw an exception manually during the execution of the program and we can handle those thrown exceptions using try-catch blocks based on our requirements.

The `throw` keyword will raise only the exceptions that are derived from the `Exception` base class.

C# throw Keyword Syntax

Following is the syntax of raising an exception using `throw` keyword in c#.

```
throw e;
```

Here, **e** is an exception that is derived from the `Exception` class and `throw` keyword to throw an exception.

C# throw Exception Example

Following is the example of using `throw` keyword to throw **NullReferenceException** and handling thrown exceptions in a try-catch block in c#.

noteghar.com

```
using System;

namespace TutlaneExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
```

```
                GetDetails();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }
        private static void GetDetails()
        {
            string name = null;
            if (string.IsNullOrEmpty(name))
            {
                throw new NullReferenceException("Name is Empty");
            }
            else
            {
                Console.WriteLine("Name: " + name);
            }
        }
    }
}
```

If you observe the above example, the **GetDetails()** method will throw **NullReferenceException** using **throw** keyword whenever the **name** variable value is empty or null.

Here, we used a `new` keyword in `throw` statement to create an object of valid exception type and we used a try-catch block in method caller to handle thrown exception.

When we execute the above example, we will get the result as shown below.

```
Name is Empty
```

This is how we can throw an exception in c# using `throw` keyword based on our requirements.

C# Re-throw an Exception

By using `throw` keyword in the `catch` block, we can re-throw an exception that is handled in the `catch` block. The re-throwing an exception is useful when we want to pass an exception to the caller to handle it in a way they want.

Following is the example of re-throwing an exception to the caller using `throw` keyword with try-catch blocks in c#.

```
using System;

namespace TutlaneExamples
{
    class Program
    {
```

flyghar.com
noteghar.com

```
        static void Main(string[] args)
        {
            try
            {
                GetDetails();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }
        private static void GetDetails()
        {
            string name = null;
            try
            {
                if (name.Length > 0)
                {
                    Console.WriteLine("Name: " + name);
                }
            }
            catch (Exception ex)
            {
                throw;
            }
        }
    }
}
```

If you observe the above example, in **GetDetails()** method we used a `throw` keyword in the `catch` block to re-throw an exception to the caller. In **Main()** method, we used a try-catch statement to handle the re-thrown exception.

To re-throw an exception, you need to use `throw` keyword without any exception parameter. In case, if we try to re-throw an exception using exception parameter like `throw e`, then the stack trace of original exception will not be preserved so if you want to re-throw an exception, then don't use any exception parameter with `throw` keyword.

When you execute the above program, you will get the result like as shown below.

```
Object reference not set to an instance of an object.
```

This is how we can re-throw an exception to the caller using `throw` keyword in `catch` block based on our requirements.

C# Custom Exception

As we learned in previous sections, c# has provided a lot of built-in exception classes to handle runtime or unexpected errors in the application. In case, if none of the predefined exception classes meet our needs, then we can create our own exception classes by deriving from `Exception` base class.

While creating our own user-defined exception class, we need to make sure that the end of class name will contain "**Exception**" word and it must be derived from `Exception` base class. After creating our own custom exception class, we need to implement the three common constructors like as shown in the following example.

flyghar.com

C# Create Custom Exception Class

Following is the example of creating a new exception class named "**TutlaneCustomException**" by deriving from `Exception` base class and including three constructors.

```
public class TutlaneCustomException : Exception
{
    public TutlaneCustomException()
    {
    }
    public TutlaneCustomException(string message)
        : base(message)
    {
    }
    public TutlaneCustomException(string message, Exception innerexcept
ion)
        : base(message, innerexception)
    {
    }
}
```

By using `throw` keyword, we can raise an exceptions using **TutlaneCustomException** class in our program based on our requirements.

C# Custom Exception Example

Following is the example of creating a custom exception class (**TutlaneCustomException**) and throwing exceptions using custom exception class in c#.

noteghar.com

```
using System;

namespace TutlaneExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = null;
            if (string.IsNullOrEmpty(name))
            {
```

```
                throw new TutlaneCustomException("Name is Empty");
            }
            else
            {
                Console.WriteLine("Name: " + name);
            }
            Console.ReadLine();
        }
    }
    public class TutlaneCustomException : Exception
    {
        public TutlaneCustomException()
        {
        }
        public TutlaneCustomException(string message)
            : base(message)
        {
        }
        public TutlaneCustomException(string message, Exception innerex
ception)
            : base(message, innerexception)
        {
        }
    }
}
```

If you observe the above example, we created a custom exception class **TutlaneCustomException** by deriving from `Exception` base class and we used a `throw` keyword to raise an exception.

When we execute the above example, we will get the result like as shown below.

```
Name is Empty
```

This is how we can create our own custom exception classes by deriving from `Exception` base class in c# based on our requirements.

noteghar.com
tlyghar.com

LINQ Introduction - What is LINQ & Why we use LINQ

Here we will learn introduction to linq, what is linq in c# / asp.net, why we use linq in c# / asp.net, linq architecture with diagram, advantages of linq, disadvantages of linq.

flyghar.com

What is LINQ?

The full form of LINQ is `Language Integrated Query` and it was introduced in **.NET Framework 3.5** to query the data from different data sources such as collections, generics, XML Documents, ADO.NET Datasets, SQL, Web Service, etc. in C# and VB.NET. The LINQ will provide a rich, standardized query syntax in .NET programming languages such as C# and VB.NET to allow the developers to interact with any type of data.

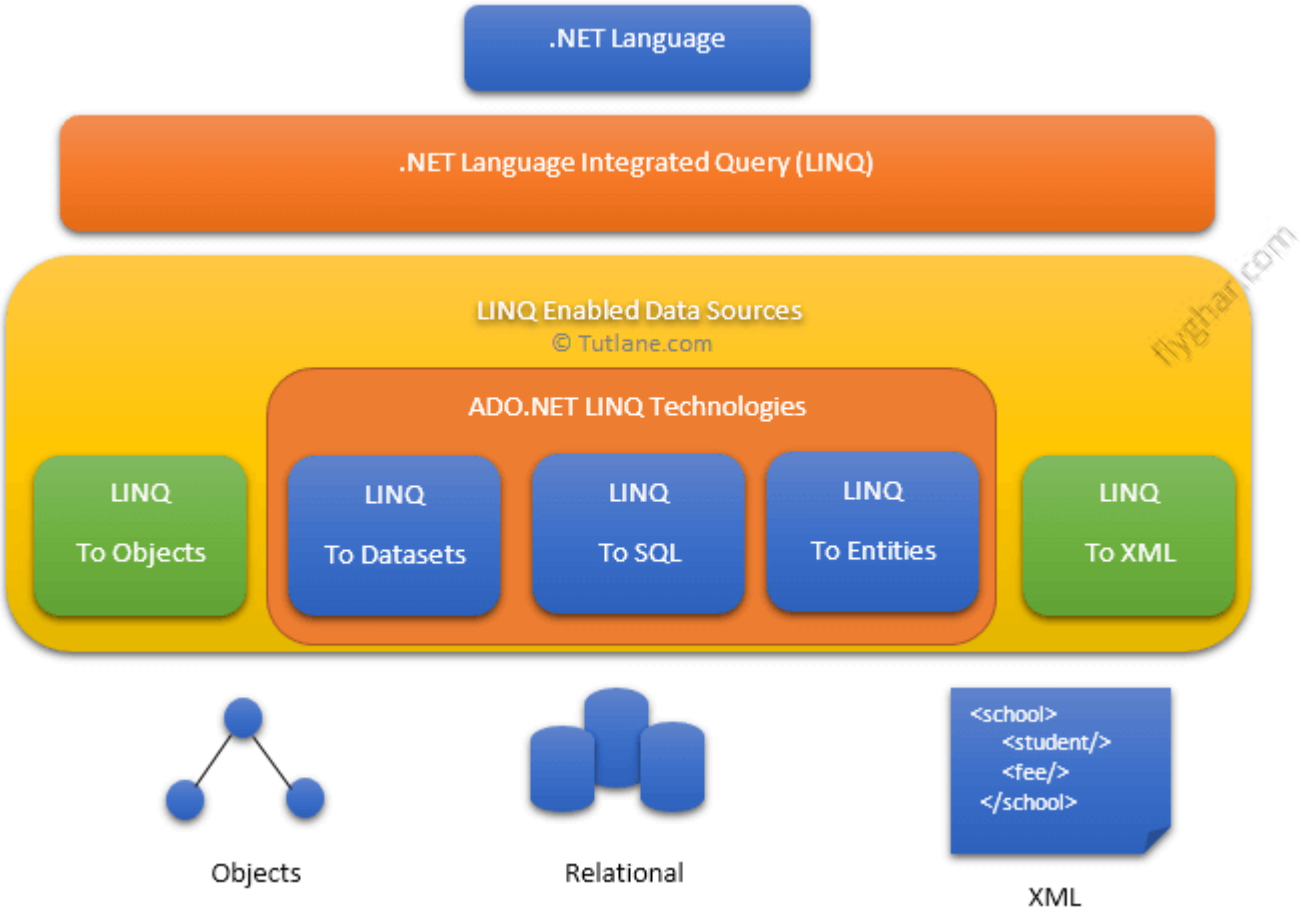
Generally, the LINQ will add a rich, standardized query syntax in .NET programming languages and it will allow the developers to write SQL style of queries within the code to interact with any type of data.

In c# or vb.net, the LINQ functionality can be achieved by importing `System.Linq` namespace in our application. Generally, the LINQ will contain a set of extension methods that allow you to query the data source objects directly in the code based on our requirements.

LINQ Architecture

Following diagram will represent the complete architectural view of LINQ.

noteghar.com



As mentioned in the above diagram following are the various types of LINQ objects available in c# and vb.net.

- LINQ to Objects
- LINQ to Datasets
- LINQ to SQL
- LINQ to Entities
- LINQ to XML

We would be covering these topics in detail in the later part of the series.

Why we need to use LINQ?

Now the question arise why should we use LINQ? What's so different in LINQ that everyone would want to use it in their application and not the SQL? Well the answer is quite simple as LINQ is simpler, tidier, and higher-level than SQL. When it comes to querying databases, In most cases, LINQ is more productive querying language than SQL.

Also we have the benefit of intellisense as LINQ query is written in code behind and moreover it has full type checking at compile time so we can catch any error in compile time itself. In c# or vb.net writing queries is more fun in LINQ but it has its advantages as well as disadvantages which are as follows:

Advantages of LINQ

Following are the advantages of using LINQ in our applications.

1. It has full type checking at compile time.
2. It has intellisense, so we can avoid silly errors.
3. Its query can be reused.
4. We can debug it using .NET debugger.
5. It supports filtering, sorting, ordering, grouping with much less effort.

tiyeghar.com

Disadvantages of LINQ

Following are the drawbacks of using LINQ in our applications.

1. Since it was written in the code we cannot make use of Cache execution plan which is a SQL feature as we do in stored procedure.
2. Writing complex queries in LINQ is a bit tedious as compared to SQL.
3. If the query is not written properly then performance is degraded.
4. We have to build the project and deploy the DLL every time some change is done in the query.

LINQ has its own advantages and disadvantages so based on requirements we can use linq in our applications.

In LINQ, we can write the queries in two ways.

- Using Query Syntax
- Using Method Syntax

noteghar.com

LINQ Query Syntax

In LINQ, we write LINQ query by following certain rules and the syntax is quite different from the SQL. To write a LINQ query, we need to follow the syntax hierarchy like as mentioned below.

```
from <variable> in <collection>
<where, joining, grouping, operators etc.> <lambda expression>
```

```
<select or groupBy operator> <format the results>
```

This order is to be followed while writing the queries in LINQ. "`from`" keyword will form the starting point of the LINQ query followed by a `user defined variable` followed by "`in`" which actually specifies our source collection or Data source followed by a `where` clause, if there is a certain condition in the query that can be used before the select to filter out the records and select is followed by group by and into the clause.

The order of the clauses in LINQ query will be like as shown below.

flyghar.com

Clause	Description
From	[Identifier]
In	[Source Collection]
Let	[Expression]
Where	[Boolean Expression]
order by	[Expression]
Select	[Expression]
group by	[Expression]
Into	[Expression]

The following are the code snippets of linq query syntax in c# and vb.net.

LINQ Query Syntax in C#

noteghar.com

```
int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
IEnumerable<int> result = from numbers in Num
                          where numbers >3
                          select numbers;
```

LINQ Query Syntax in VB.NET

```
Dim Num AsInteger() = {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
Dim result AsIEnumerable(OfInteger) = From numbers In Num
                                      Where numbers > 3
                                      Select numbers
```

Following are the examples that will show how to use the LINQ query syntax in our applications. Here, we are using an integer array which is having nine elements and we will print the array elements whose value is greater than **3**.

LINQ Query Syntax Example in C#

flyghar.com

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace Linqtutorials
{
class Program
{
static void Main(string[] args)
{
int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
//LINQ Query Syntax to Print Numbers Greater than 3
IEnumerable<int> result = from numbers in Num
                          where numbers >3
                          select numbers;
foreach (var item in result)
{
Console.WriteLine(item);
}
Console.ReadLine();
}
}
}
```

LINQ Query Syntax Example in VB.NET

noteghar.com

```
Module Module1
Sub Main()
Dim Num As Integer() = {1, 2, 3, 4, 5, 6, 7, 8, 9}
'LINQ Query Syntax to Print Numbers Greater than 3
Dim result As IEnumerable(Of Integer) = From numbers In Num Where numbe
rs > 3 Select numbers
For Each item In result
Console.WriteLine(item)
Next
Console.ReadLine()
End Sub
End Module
```

In the above example, we define an integer array (**Num**) with nine elements and printing only the array elements whose value is greater than **3**.

If you observe the above examples, we used "**from**" identifier at the beginning of the query (Refer Ordering table) and declared **numbers** variable which is user-defined and added **where** condition to get the elements whose value is greater than **3** and we used **numbers** variable with a select clause to retrieve the result set.

And of course, we will get the elements from **Num** integer array where **numbers** value greater than **3** which is our data source in this program using "**in**" keyword.

Since the resultant output in LINQ is of type IEnumerable<int> so we declared the result type as IEnumerable<int> and to access elements from the resultant object we used a foreach loop to get the desired output.

When we execute the above examples, we will get the result as shown below.

```
4

5
6
7
8
9
```

LINQ Method Syntax

In LINQ, method syntaxes will use the extension methods of Enumerable or **Queryable** static classes.

The following are the code snippet of using method syntaxes in c# and vb.net.

LINQ Method Syntax in C#

```
int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
//LINQ Method Syntax to Print Numbers Greater than 3
IEnumerable<int> result = Num.Where(n => n > 3).ToList();
```

LINQ Methos Syntax in VB.NET

```
Dim Num As Integer() = {1, 2, 3, 4, 5, 6, 7, 8, 9}
'LINQ Method Syntax to Print Numbers Greater than 3
Dim result As IEnumerable(Of Integer) = Num.Where(Function(n) n > 3).To
List()
```

If you observe above code snippets we are using extension methods **Where** and **ToList()** of IEnumerable class.

LINQ Method Syntax Example

The following are the examples of using method syntax query with IEnumerable<int> in c# and vb.net.

LINQ Method Syntax Example in C#

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LINQExamples
{
class Program
{
static void Main(string[] args)
{
int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
//LINQ Method Syntax to Print Numbers Greater than 3
IEnumerable<int> result = Num.Where(n => n > 3).ToList();
foreach (var item in result)
{
Console.WriteLine(item);
}
Console.ReadLine();
}
}
}
```

If you observe the above example, we used IEnumerable collection extension methods (**Where** and **ToList()**) to query the data based on our requirements.

When we execute the above program, we will get the result as shown below.

```
4

5
6
7
8
9
```

This is how we can write the queries in LINQ using query syntax and method syntax based on our requirements.

LINQ Lambda Expressions

In LINQ, **Lambda expression** is a function without a name. It makes the syntax short and clear. Though it is not as readable as a LINQ query it is equally important as the LINQ query and gets

convert to lambda internally. Its scope is limited to where it is used as an expression. We cannot reuse it afterward.

LINQ Lambda Expressions Syntax

Following is the syntax of defining lambda expressions in LINQ.

```
(Input Parameter) => Method Expression
```

Lambda Expression is dynamic and decides the type in compile time. In the above lambda expression on the left-hand side, we have a bracket **()** containing the Input parameter.

The name of the parameter can be anything and ahead of this parameter (`=>`) is an **Equal to** (`=`) followed by a **greater than** (`>`) symbol which is used to send or pass the input parameter from left to right side and on the right-hand side the operation is performed using the input parameter passed from the left-hand side parameters. This whole syntax forms a `Lambda Expression`.

Here we have an example to demonstrate what we have read so far. Let's say we have an expression like as shown below.

```
X => X + 10
```

Here `X` is an input parameter followed by `=>` operator and next to the operator there is an expression that adds numeric `10` to the input variable (`X`). So the output would increment the numeric `10` to the `X` variable which was the input parameter on the left-hand side of the expression.

The following are examples of using LINQ lambda expressions in c# and vb.net.

LINQ Lambda Expressions Example C#

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LINQExamples
{
class Program
{
static void Main(string[] args)
{
List<string> countries = new List<string>();
countries.Add("India");
countries.Add("US");
countries.Add("Australia");
countries.Add("Russia");
IEnumerable<string> result = countries.Select(x => x);
foreach (var item in result)
{
```

```
Console.WriteLine(item);
}
Console.ReadLine();
}
}
}
```

LINQ Lambda Expressions Example VB.NET

flyghar.com

```
Module Module1
Sub Main()
Dim countries As New List(Of String)()
countries.Add("India")
countries.Add("US")
countries.Add("Australia")
countries.Add("Russia")
Dim result As IEnumerable(Of String) = countries.[Select](Function(x) x
)
For Each item In result
Console.WriteLine(item)
Next
Console.ReadLine()
End Sub
End Module
```

If you observe the above examples, we created a list object (**countries**) with country names and we are using lambda expression to select the countries from list object (**countries**).

Here `Select` is the property that we use to select from a list. And `x` is the input parameter that is on the left-hand side of the expression followed by `=>` operator. On the right-hand side of the expression is the same input parameter which denotes that we just want to display the parameter without performing any operation on it. So we are not specifying any condition on it.

When we run the above linq lambda expressions example, we will get the result like as shown below.

```
India

US
Australia
Russia
```

We will create one more example to dig further into the linq Lambda Expressions. In the following example, we have an integer array having elements from 1 to 10 and we will display only even elements from the array.

noteghar.com

LINQ Lambda Expressions Example2 C#

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace LINQExamples
{
class Program
{
static void Main(string[] args)
{
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
Console.WriteLine("Below are the even numbers");
Console.WriteLine("");
IEnumerable<int> evennumber = numbers.Where(x => x % 2 == 0);
foreach (var item in evennumber)
{
Console.WriteLine(item + " is an even number");
}
Console.WriteLine("-------------------------------------------------")
;
Console.WriteLine("Below are the Odd numbers");
Console.WriteLine("");
IEnumerable<int> oddnumber = numbers.Where(x => x % 2 != 0);
foreach (var item in oddnumber)
{
Console.WriteLine(item + " is an odd number");
}
Console.ReadLine();
}
}
}
```

In the above examples, we are trying to display the Even and Odd numbers separately from the array list (**numbers**). Here, "Where" clause is used as a method in the list to specify a desired condition and it is supplied by using **Lambda expression**. We just name an input parameter on left hand side and following the `=>` operator, and specified the condition to select only even numbers. Once the numbers are selected into the variable named "**evennumber**" of type IEnumerable<int>, we loop through the collection using a foreach loop to get the result.

Similarly, we are finding the Odd numbers by just modifying the condition in our Lambda Expression and storing the result in the variable named **oddnumber** of type IEnumerable<int> and looping further to get the desired result.

When we execute the above examples, we will get the result as shown below.

```
Below are the even numbers

2 is an even number
4 is an even number
6 is an even number
8 is an even number
10 is an even number
-------------------------------------------------
Below are the Odd numbers
```

```
1 is an odd number
3 is an odd number
5 is an odd number
7 is an odd number
9 is an odd number
```

This is how we can use the Lambda Expressions in our applications to stores the result in a variable of type IEnumerable<int> and we can loop through it to display the values based on our requirements.

flyghar.com

noteghar.com