



C# & ASP.NET Documentation

Detailed documentation of C# and dotnet for restiful apis

Table of Contents

C#	5
Introduction & Environment Setup	7
Introduction to C# Programming Language	12
How to Download and Install Visual Studio on Windows	15
Creating First Console Application using Visual Studio	23
C#.NET Basics	28
Methods and Properties of Console Class in C#	35
Var Keyword in C#	43
Const Keyword in C#	44
Data Types in C#	46
Value Data Types in C#	48
Predefined Data types	50
Numeric Numbers with Decimal (Single, Double, Decimal)	51
Integers Data types (Numbers without Decimal)	55
Char Data Type	59
Byte Data Type	60
Boolean Data Type	61
User Defined Data Type	67
Reference Data Types	75
Pointer Data Type	83
Type Casting in C#	89
Variables in C#	96
Operators in C#	113
Control Flow Statements in C#	128
Functions in C#	173
Object-Oriented Programming	184
Class & Objects	190
Constructors in C#	195
Access Specifiers	202
OOP Principles: Encapsulation	214
OOP Principles: Inheritance	221
OOP Principles: Abstraction	236
Application Development Process	247

Interface in C#	252
OOP Principles: Polymorphism	258
Method Overloading	261
Method Overriding	269
Method Hiding	272
OOP Real-Time Examples	276
OOP Real-Time Exercise Schenarios	310
Exception Handling	316
Custom Exceptions	329
Collections in C#	335
Array	338
ArrayList	349
Hashtable	360
Generic Collections	370
List<T>	374
HashSet<T>	378
SortedSet<T>	382
Stack<T>	386
File Handling	389
FileStream Class	392
StreamReader & StreamWriter	394
File Class	396
FileInfo Class	398
DirectoryInfo Class	400
Asynchronous Programming	402
AutoMapper	415
Lambda Expressions	442
LINQ	446
IEnumerable and IQueryable in C#	455
Fluent API in LINQ	462
IQueryable Workshop	467
Entity Framework	484
Entity Framework Architecture	488
DbContext in EF	491
Entities in Entity Framework	494
Entity States in Entity Framework	497

Development Approach with Entity Framework	499
Entity Framework Code First Approach	503
Default Code-First Conventions in Entity Framework	505
Configure Domain Classes in Entity Framework	510
Data Annotation Attributes in Entity Framework Code-First	515
Table Data Annotation Attribute in Entity Framework	518
Column Data Annotation Attribute in Entity Framework	520
Key Attribute in Entity Framework	526
ForeignKey Attribute in Entity Framework	529
Index Attribute in Entity Framework	532
Relationships in EF	536
EF Core CLI Database Commands	557
ASP.NET Core	560
Introduction & Env Setup	561
ASP.NET Web API	562
Basics of Web APIs	565
ASP.NET Core Web API – Basics	589
Controllers in ASP.NET Core Web API	604
Models in ASP.NET Core Web API	611
[Workshop] Controllers & Models : CRUD without dbs	615
Services in ASP.NET Core Web API	635
Dependency Injection in ASP.NET Core Web API	647
DI : Singleton vs Scoped vs Transient in ASP.NET Core Web API	654
[Workshop] DI in ShoppingCart API	658
Swagger API in ASP.NET Core Web API	676
Routing	682
Action Return Types in ASP.NET	699
Model Binding	715
Entity Framework Core	735
[Recap Workshop] Relationships	741
Lazy Loading in EF Core	767
Logging – ASP.NET Core Web API	773
Caching – ASP.NET Core Web API	774
In-Memory Caching in ASP.NET Core Web API	778
Redis Cache in ASP.NET Core Web API	780
Stored Procedures in EFCore	801

Fluent Validation	802
[Workshop] Fluent validation with manual validation	829
Filters – ASP.NET Core Web API	855
Security – ASP.NET Core Web API	856
JWT – ASP.NET Core Web API	857
API Versioning in ASP.NET Core	858
Repository Pattern in ASP.NET Core Web API	859
Unit Testing – ASP.NET Core Web API	860
Minimal API – ASP.NET Core	861
Microservices in ASP.NET Core	862
SQL Server	863

C#



image_282.png

C# (pronounced "C-Sharp") is a modern, general-purpose, object-oriented programming language developed by Microsoft as part of the .NET framework. It is a versatile, statically typed, and open-source language widely used for building a diverse range of applications.

Key Features of C#

- **Object-Oriented:** C# is fundamentally object-oriented, allowing for code reusability, a clear program structure, and high scalability.
- **Modern and Type-Safe:** It incorporates modern programming features, including functional programming patterns, and uses strong type checking and automatic memory management (garbage collection) to prevent common errors like memory leaks.
- **Cross-Platform:** While originally Windows-only, modern .NET allows C# applications to run on Windows, macOS, and Linux.
- **Syntax:** C# belongs to the C family of languages (its name is a musical reference to being a half-step above C++) and has a syntax familiar to developers who have used

C++, Java, or JavaScript.

- **Large Ecosystem & Community:** It benefits from extensive documentation and a large, active community of developers, which provides support and contributes to its open-source development on GitHub.

What C# Is Used For C# is a highly versatile language used across many industries and for various types of projects:

- **Game Development:** It is the primary language used with the popular Unity game engine for building games for PCs, consoles, and mobile devices.
- **Web Applications & Services:** Developers use the ASP.NET framework to create dynamic websites, web APIs, and robust back-end services.
- **Windows Desktop Applications:** As a Microsoft language, it is a standard choice for developing native Windows software with frameworks like Windows Presentation Foundation (WPF) and Windows Forms.
- **Mobile Applications:** With frameworks like Xamarin and .NET MAUI, C# can be used to build cross-platform mobile apps for iOS and Android.
- **Cloud Services:** C# is used extensively for building scalable cloud-based applications and microservices on platforms like Microsoft Azure.

Introduction & Environment Setup

What does .NET Represent?

NET stands for **Network Enabled Technology** (Internet). 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.

According to Microsoft, **.NET is a Free, Cross-Platform, Open-Source** developer platform for building many different types of applications. With .NET, we can use multiple languages (C#, VB, F#, etc.), Editors (Visual Studio, Visual Studio Code, Visual Studio for Mac, OmniSharp, JetBrains Rider, etc), and Libraries to build for Web, Mobile, Desktop, Games, IoT, and more.

Cross Platform: Whether you are working in C#, F#, or Visual Basic, your code will run on any compatible operating system. You can build many types of apps with .NET. Some are Cross-Platform, and some target a specific set of operating systems and devices.

Libraries: To extend functionality, Microsoft and others maintain a healthy .NET package ecosystem. NuGet (<https://www.nuget.org/>) is a package manager built specifically for .NET that contains over 100,000 packages.

What is a Framework?

A framework is a software. Or you can say a framework is a collection of many small technologies integrated together to develop applications that can be executed anywhere.

What does the .NET Framework Provide?

The DOT NET Framework provides two things as follows

1. **BCL** (Base Class Libraries)
2. **CLR** (Common Language Runtime)

What is BCL?

Base Class Libraries (BCL) are designed by Microsoft. Without BCL we can't write any code in .NET. So, BCL is also known as the basic building block of .NET Programs. 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. The physical location of BCL is **C:\Windows\assembly**

What is CLR?

CLR stands for Common Language Runtime and it is the core component under the .NET framework which is responsible for converting the MSIL (Microsoft Intermediate Language) code into native code. In our CLR session, we will discuss CLR in detail.

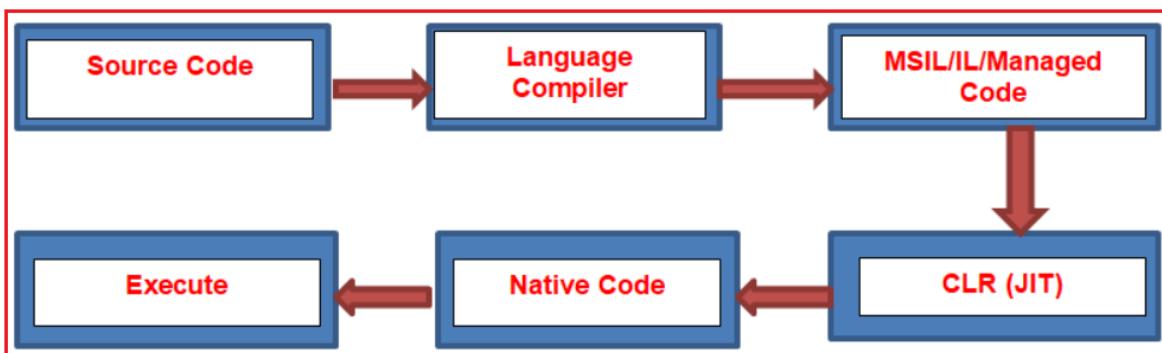
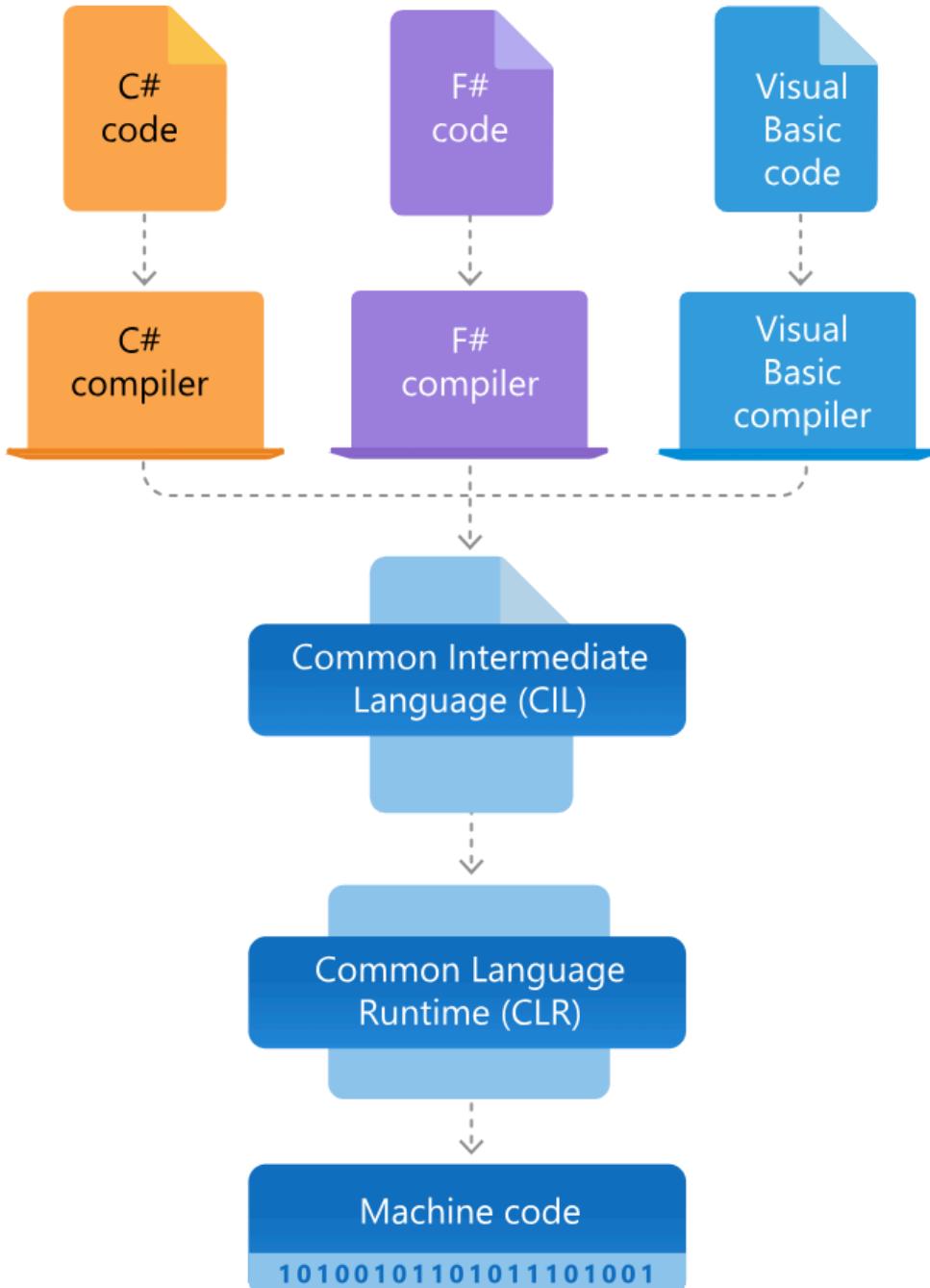


image.png

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



image_2.png

- In the 1st compilation, the source code is compiled by the respective language compiler and generates the intermediate code which is known as **MSIL (Microsoft Intermediate Language)** or **CIL (Common Intermediate language code)**, or Managed Code.

- In the 2nd compilation, **MSIL** is converted into **Native code/ Machine Code** (native code means code specific to the Operating system so that the code is executed by the Operating System) and this is done by **CLR**.

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

What is JIT?

JIT stands for the **Just-in-Time** compiler. It is the component of **CLR** that is responsible for *converting MSIL code into Native Code*. Native code is code that is directly understandable by the operating system.

Different types of .NET Framework

The .NET framework is available in many different flavors, but below are the common ones

- **.NET Framework:** .NET Framework is the original implementation of .NET. It supports running websites, services, desktop applications, and more on Windows OS Only.
- **.NET:** .NET is a cross-platform implementation for running websites, services, and console applications on Windows, Linux, and macOS. .NET is open source on GitHub and .NET was previously called .NET Core.
- **Xamarin/Mono:** Xamarin/Mono is a .NET implementation for running apps on all the major mobile operating systems, including iOS and Android.



Note: .NET Framework is **Platform-Dependent** while .NET or .NET Core is **Platform Independent**. Here, we are not talking about Web Applications. Web Applications are independent of Operating Systems.

What is Exactly .NET? .NET is a framework tool that supports many programming languages and many technologies. .NET support 60+ programming languages. Of 60+ programming languages, 9 are designed by Microsoft and the remaining are designed by non-Microsoft. Microsoft-designed programming languages are as follows:

1. VB.NET

2. C#.NET

3. VC++.NET

4. J#.NET

5. F#.NET

6. Jscript.NET

7. WindowsPowerShell

8. Iron phyton

9. Iron Ruby

Technologies supported by the .NET framework are as follows

1. **ASP.NET** (Active Server Pages.NET) – MVC, **Web API**, Core MVC, Core Web API, Core Blazor, etc.

2. ADO.NET (Active Data Object.NET)

3. WCF (Windows Communication Foundation)

4. WPF (Windows Presentation Foundation)

5. WWF (Windows Workflow Foundation)

6. AJAX (Asynchronous JavaScript and XML)

7. **LINQ** (Language Integrated Query) : It is a query language

8. **Entity Framework**: It is an ORM-based open-source framework that is used to work with a database using .NET objects.

Introduction to C# Programming Language



image_283.png

Why did C#.NET come to the market?

C#.NET Programming Language is mainly designed to overcome the disadvantages of C and C++ and to develop internet applications (web applications) by achieving platform independence.

Why C# is so much popular nowadays?

C#.NET is so much popular nowadays because of the following reasons.

- **C# is Simple and Familiar:** C# is simple because C# simplifies the programmer's job by avoiding certain features of C and C++. C# avoids explicit memory management. Memory management in C# is automatic. It is done by CLR.
- **C# is Portable:** Portability allows the programmer to write the same code for different machines (operating systems). C# provides portability in two ways

- Source Code Portability
- IL Code Portability (DLL and EXE)
- C# is Robust: Robust means Strong. C# is a strong type-checking language having strict type-checking during both compilation time and execution time which allows us to develop error-free applications and programs.
- C# is Multithreaded: A process is divided into several small parts known as threads or lightweight processes.
- C# is Dynamic: C# 4.0 introduced a new type called dynamic that avoids compile-time type checking.
- C# is Compiled and Interpreted: We know a programming language is either compiled or interpreted. But C# combines both approaches. That's why C# is called a two-stage system.
- C# is Object-Oriented: Except for the primitive data types, all elements in C# are objects.

Types of Applications Developed using C#:

With the help of the C# programming language, we can develop different types of secured and robust applications:

1. Window applications
2. Web applications
3. Distributed applications
4. Web service applications
5. Database applications
6. Mobile Applications, etc

C# History



Anders Hejlsberg

The history of the C# Programming Language is interesting to know. C# is pronounced as “C-Sharp”. It is an object-oriented programming language provided by Microsoft that runs on the .NET Framework, NET Core, or .NET. **Anders Hejlsberg** is known as the Founder of C# Programming Language.

C# (C# 1.0) was first introduced with .NET Framework 1.0 in the year 2002 and evolved much since then.

How to Download and Install Visual Studio on Windows



image_284.png

What is Visual Studio?

Microsoft Visual Studio is an Integrated Development Environment (IDE) from Microsoft. Visual Studio is a comprehensive tool for developing applications on the .NET platform. It's the go-to resource for all your programming needs. One can develop, debug and run applications using Visual Studio.

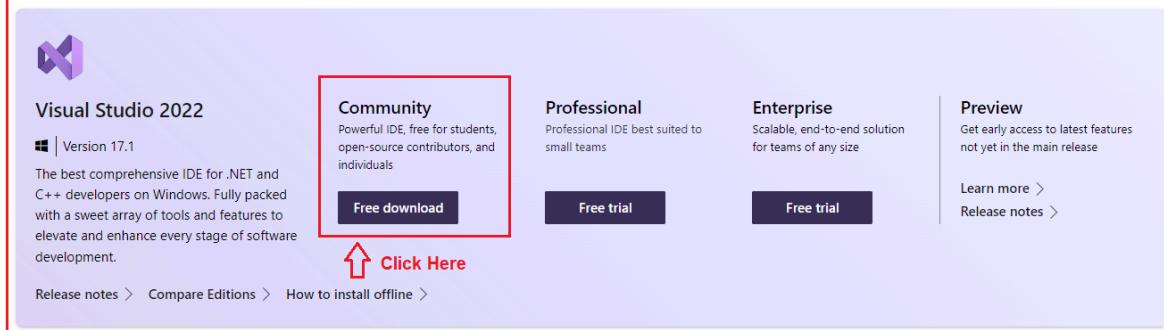
Download Visual Studio Latest Version

In order to download visual studio, please visit the following Visual Studio download link :

<https://visualstudio.microsoft.com/downloads/>

(<https://visualstudio.microsoft.com/downloads/>)

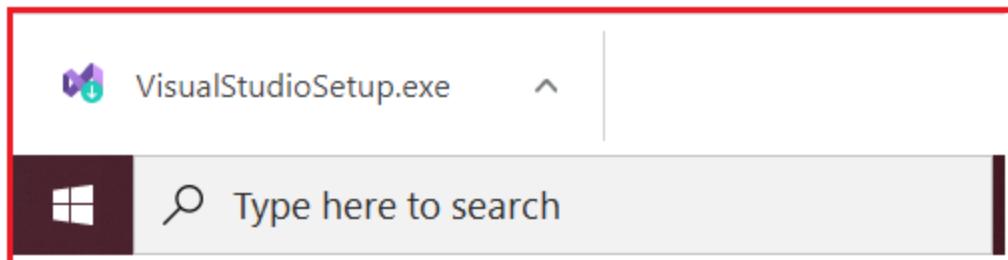
Downloads



image_4.png

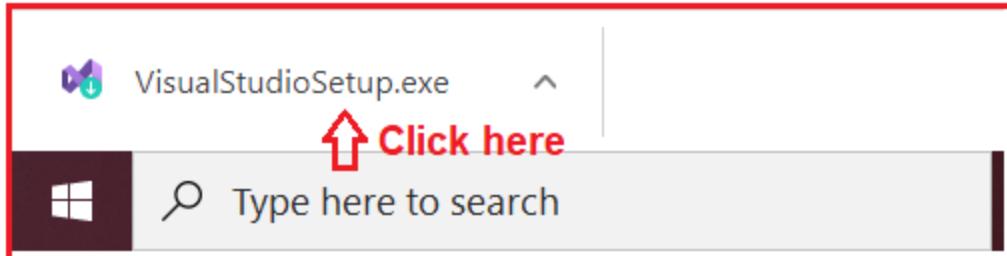
Here, you can select

1. Visual Studio 2022 Community Edition
2. Visual Studio 2022 Professional Edition (90-Day Free Trial)
3. Visual Studio 2022 Enterprise Edition (90-Day Free Trial) If you are a student, or you just want to practice C# Programming, then you can select the Community Edition which is absolutely free. I am selecting the Community Edition and simply clicking on the free download button as shown in the above image. Once you click on the free download button, it will download the Visual Studio 2022 Community Edition EXE file as shown in the below image.



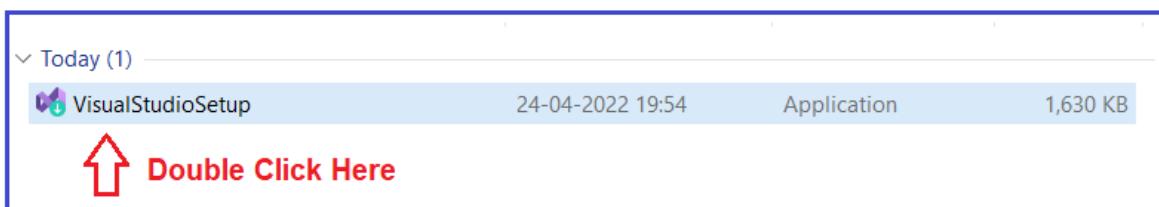
image_5.png

Open the .exe file Simply click on the downloaded EXE file as shown in the below image.



image_6.png

Or you can also double-click on the EXE file which is downloaded on your machine as shown in the below image.

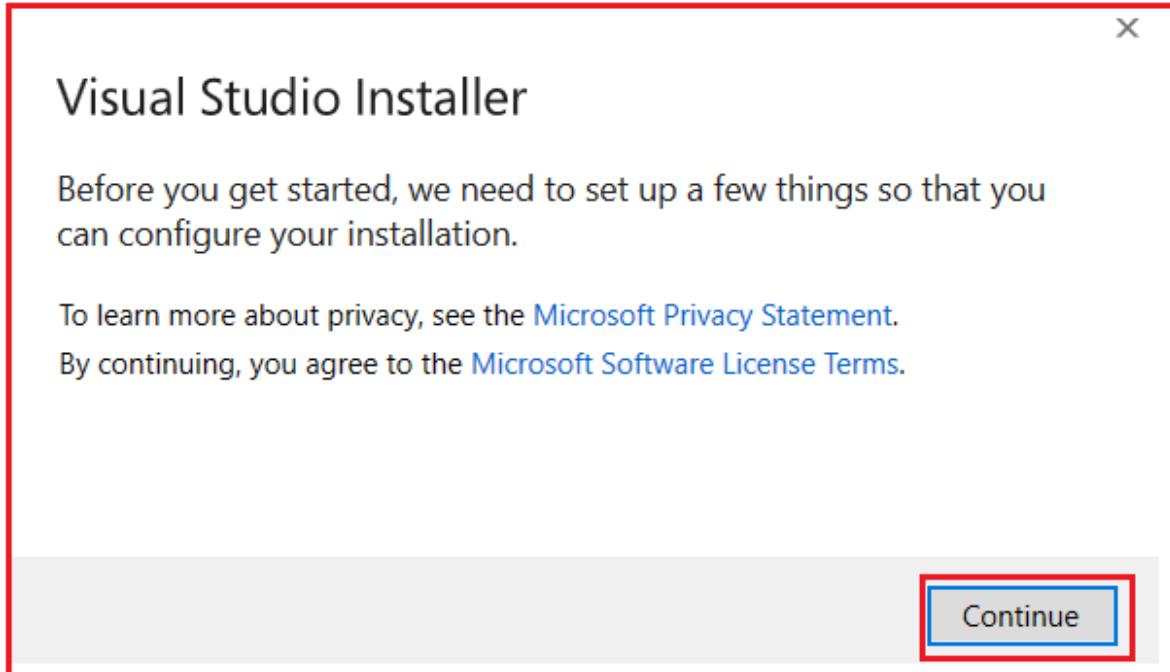


image_7.png

Once you double-click, it will open one popup asking do you want to allow this app to make changes to your device, simply click on the Yes button to continue.

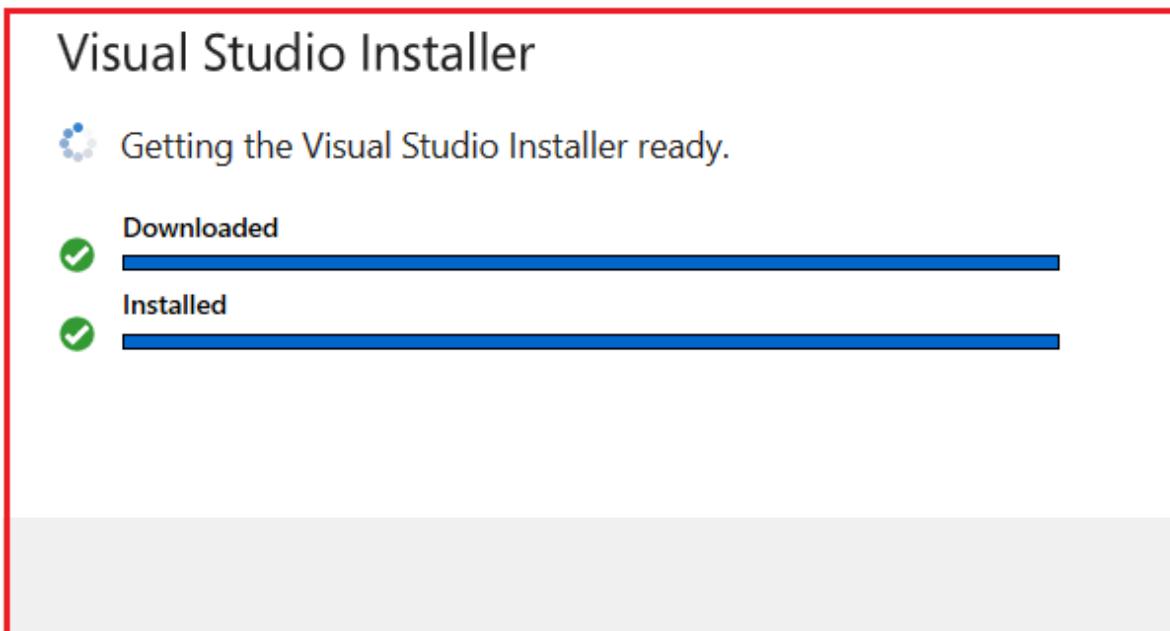
Start the Visual Studio Installation

Once you click on the Yes button in the previous step, it will open the following window. From this screen, simply click on the continue button to start the Visual Studio installation as shown in the below image.



image_8.png

After clicking on the Continue button, Visual Studio will begin downloading the necessary initial files. The download speed is subject to change depending on your internet connection. Let the installation complete as shown in the below image.



image_9.png

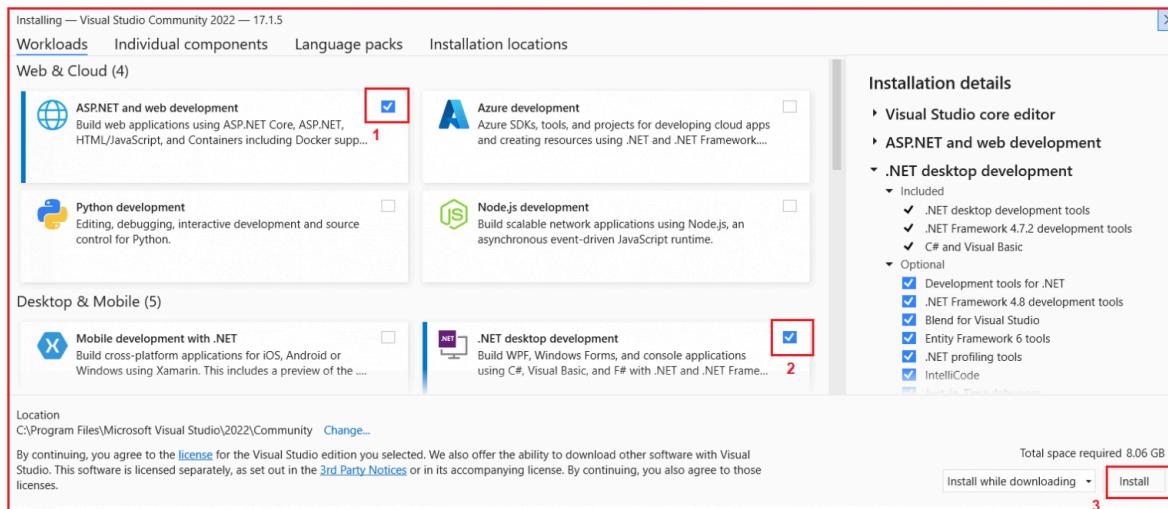
Select the Visual Studio Features:

Once the initial files are downloaded, then you will get the following Visual Studio installer screen. Here, I am selecting two checkboxes.

1. [x] ASP.NET and Web Development: For developing Web Based application

2. [x] .NET Desktop Development: For developing Console, Windows Form, WPF

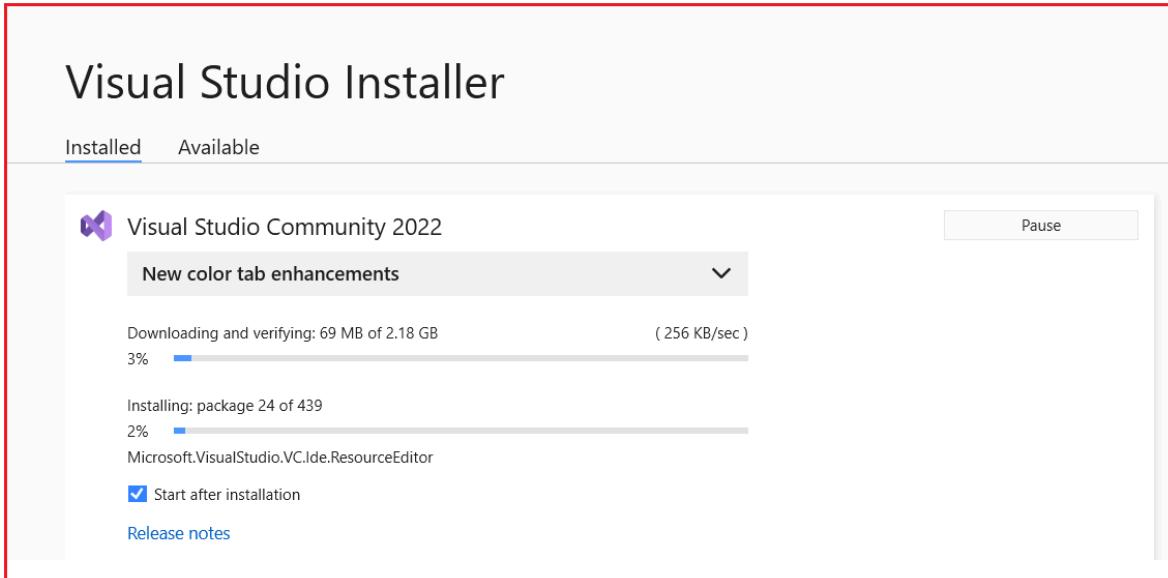
Applications Once you select the above two checkboxes from the workload section, simply click on the Install button as shown in the below image.



image_10.png

Wait for the files to be downloaded

Once you click on the Install button, Visual Studio will start downloading the relevant files based on the features you selected in the previous step. This will take some time to download the required files. You can monitor the same as shown in the below image.



image_11.png

Sign in to Visual Studio

Once the download is completed, you will get the screen asking you to sign in to Visual Studio. You can sign in or you can sign in later also.

Open Visual Studio

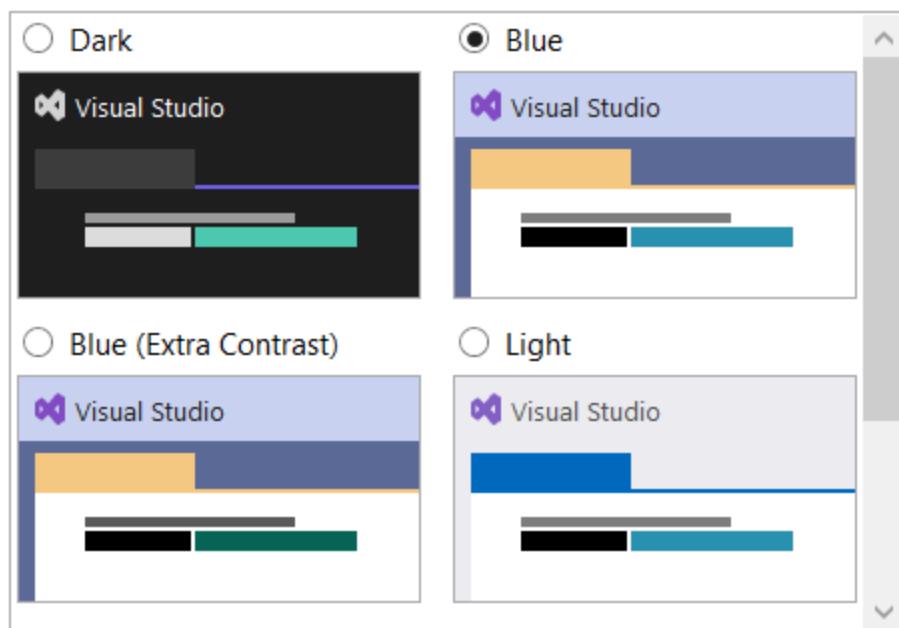
Open the Visual Studio IDE. As we are opening for the first time, it will ask you to select the theme as per your choice and then simply click on the Start Visual Studio Button as shown in the below image.

Visual Studio

Start with a familiar environment

Development Settings: General

Choose your color theme

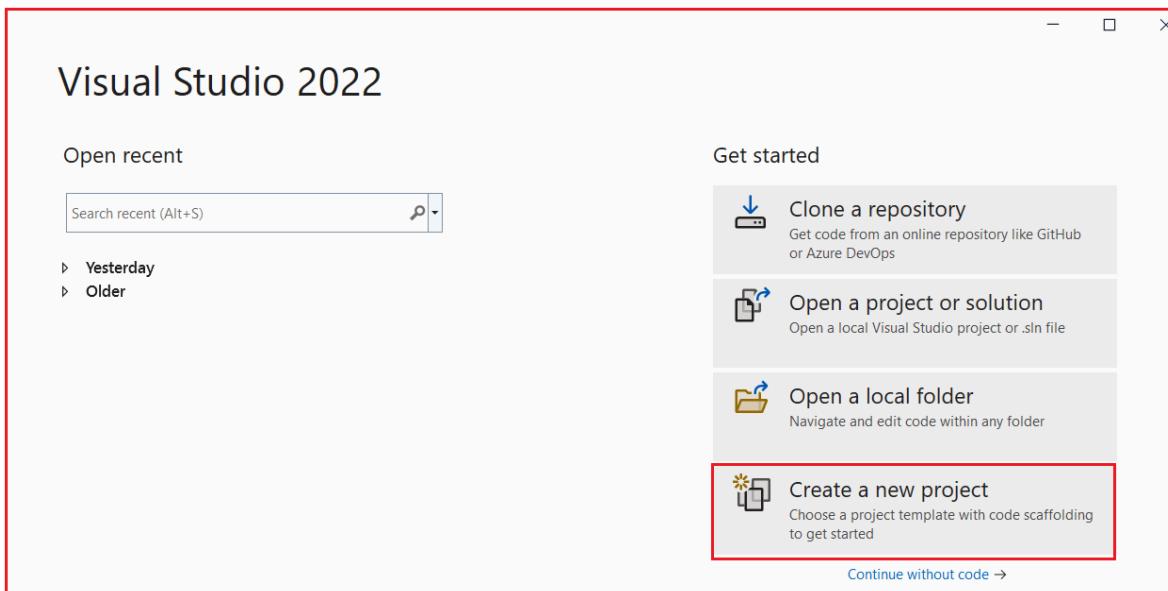


You can always change these settings later.

[Start Visual Studio](#)

image_12.png

That's it. Now you are ready to develop .NET Applications using C# language in Visual Studio. In Visual Studio IDE, you can create new C# applications by clicking on Create a new project option as shown in the below image.



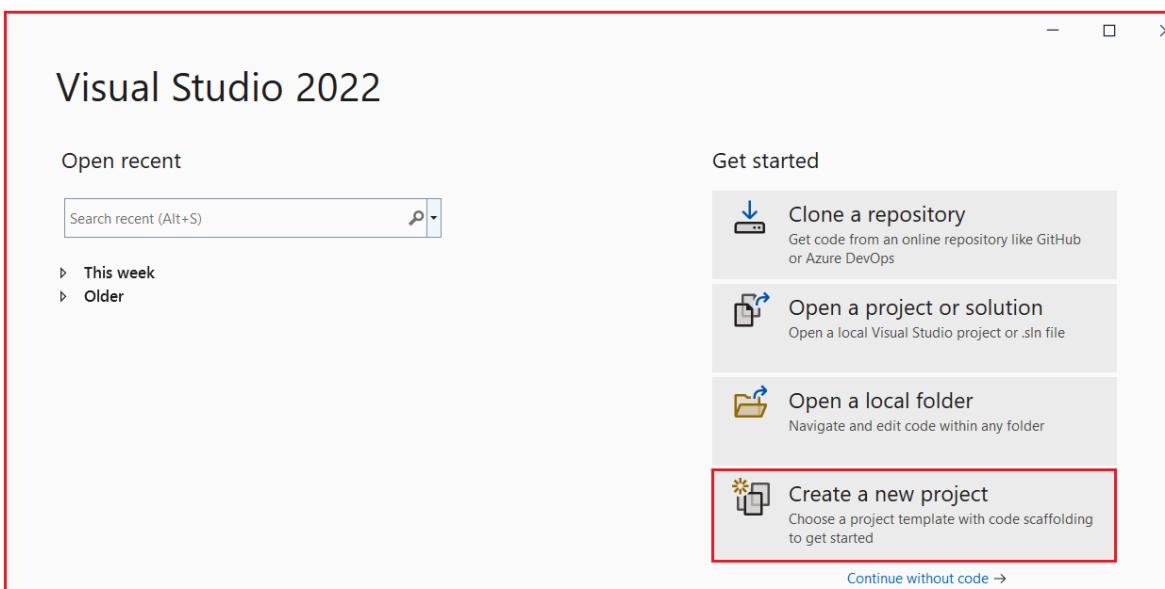
image_13.png

Creating First Console Application using Visual Studio

A console application is an application that can be run in the command prompt in Windows. For any beginner on .Net, building a console application is ideally the first step to learning the C# Language.

Step1

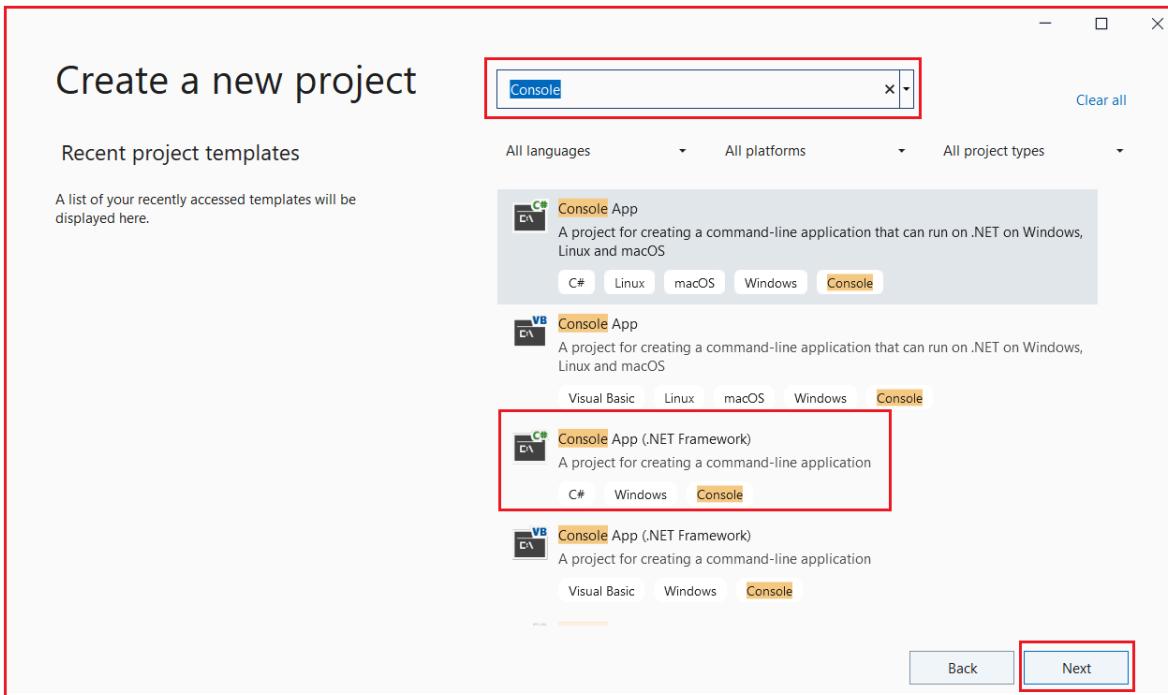
The first step involves the creation of a new project in Visual Studio. For that, open Visual Studio 2022 (the latest version at this point in time) and then click on the Create a New Project option as shown in the below image.



image_14.png

Step2

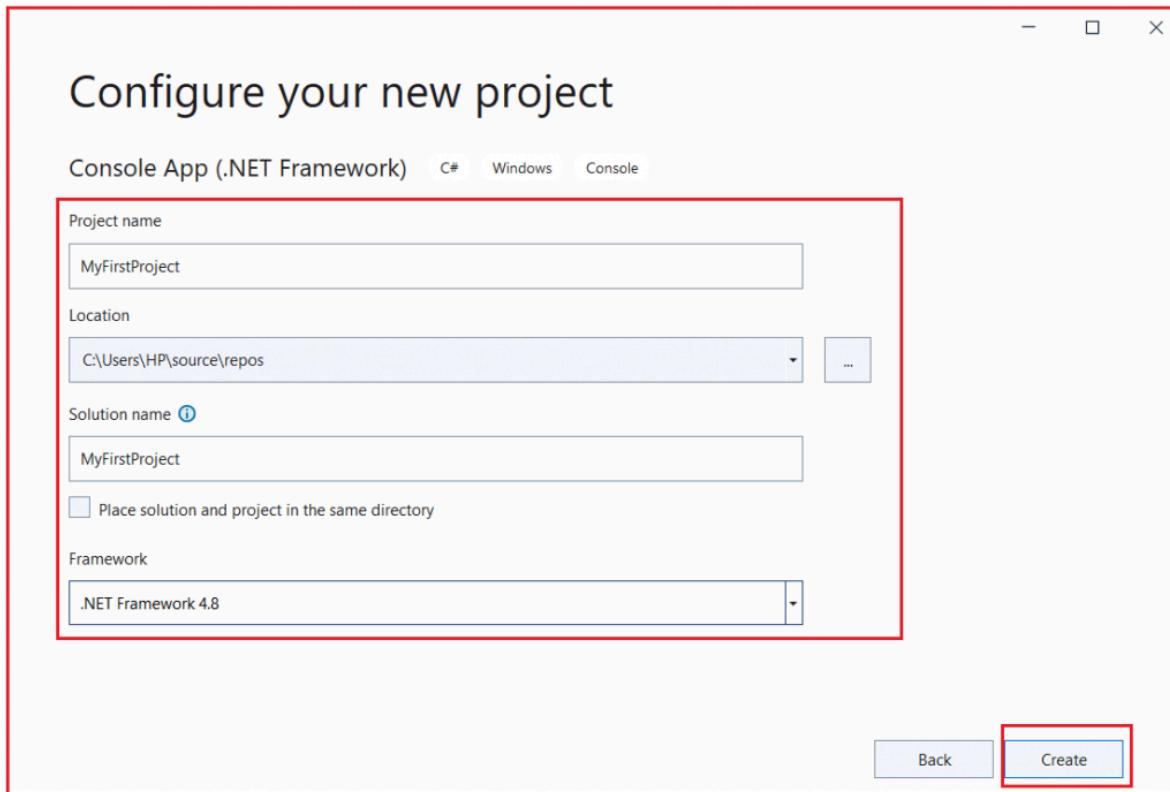
The next step is to choose the project type as a Console application. Type Console in the search bar and you can see different types of console applications using C#, VB, and F# languages using both .NET Framework and Core Framework. Here, I am selecting Console App (.NET Framework) using C# language and then clicking on the Next button as shown in the below image.



image_15.png

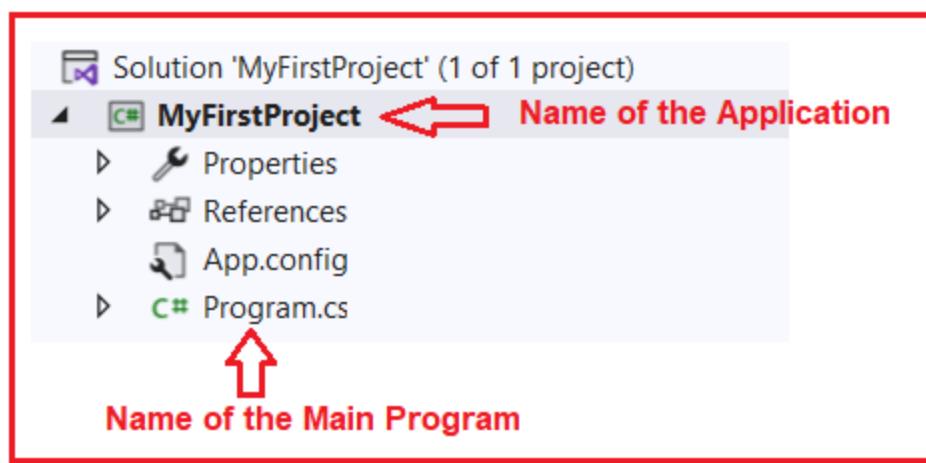
Step3

The next step is to configure the new project. Here, you need to provide the project name and solution name. You can also give the same name to both project and solution but it is not mandatory. You need to provide the location where you need to create the project. Here, you also need to provide the .NET Framework version you want to use. The latest version of the .NET Framework is 8.0 So, I am selecting .NET Framework 8.0 and then clicking on the Create button as shown in the below image.



image_16.png

Once you click on the Create button, visual studio will create the Console Application with the following structure.



image_17.png

A project called MYFirstProject will be created in Visual Studio. This project will contain all the necessary required files to run the Console application. The Main program called Program.cs is the default code file that is created when a new console application is

created in Visual Studio. This Program.cs class will contain the necessary code for our console application. So, if you look at the Program.cs class file, then you will see the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyFirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

image_18.png

Step4

Now let's write our code which will be used to display the message "Hello World" in the console application.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

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

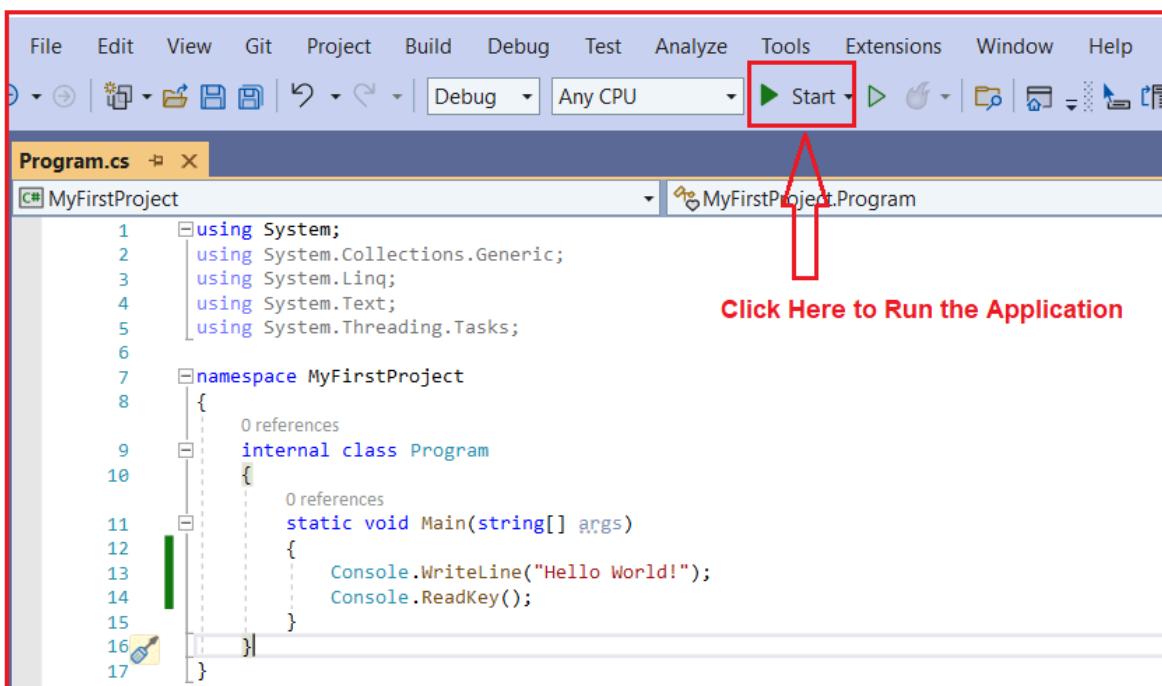
```

    {
        Console.WriteLine("Hello World!");
        Console.ReadKey();
    }
}

```

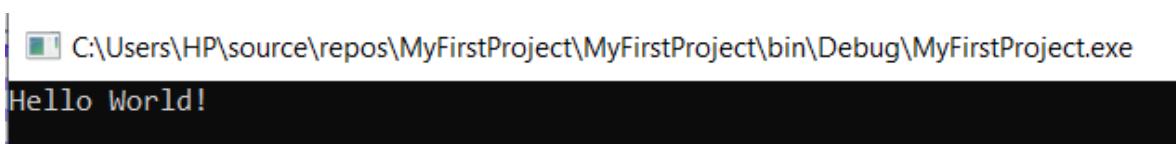
Step5

The next step is to run the .Net program. To run any program in Visual Studio, you need to click the Start button as shown in the below image.



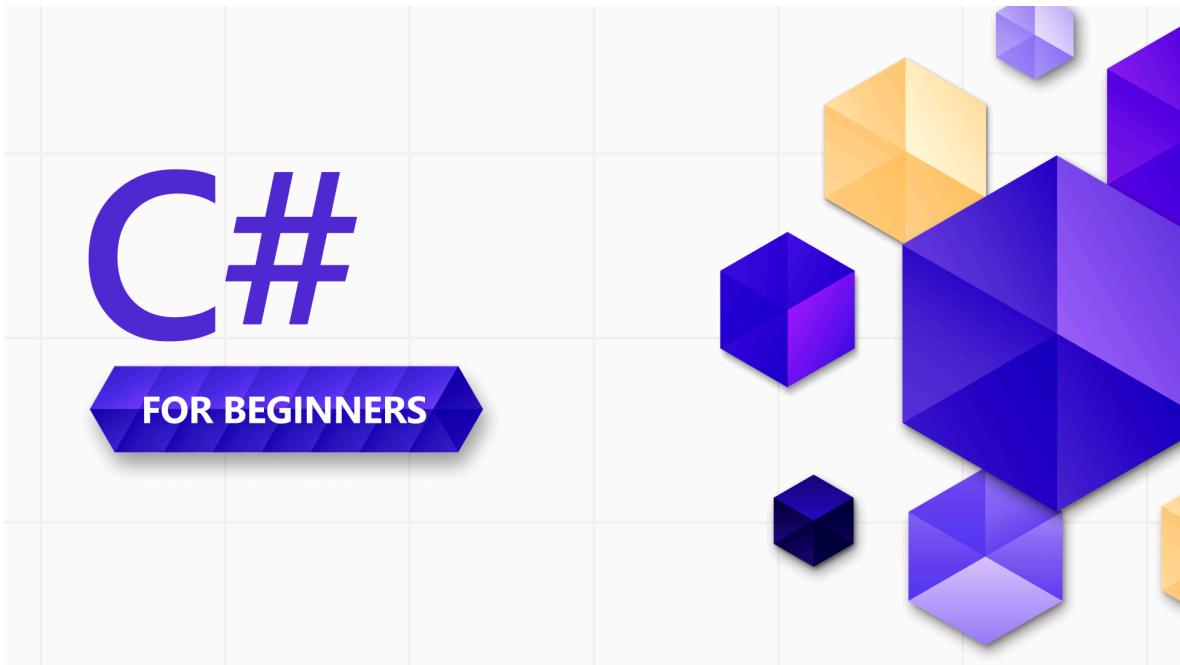
image_19.png

Once you click on the Start, you should get the following console window showing the message.



image_20.png

C#.NET Basics



image_285.png

Basic Structure of C# Program

Now, let's understand the Basic Structure of the C# Program using a Console Application.

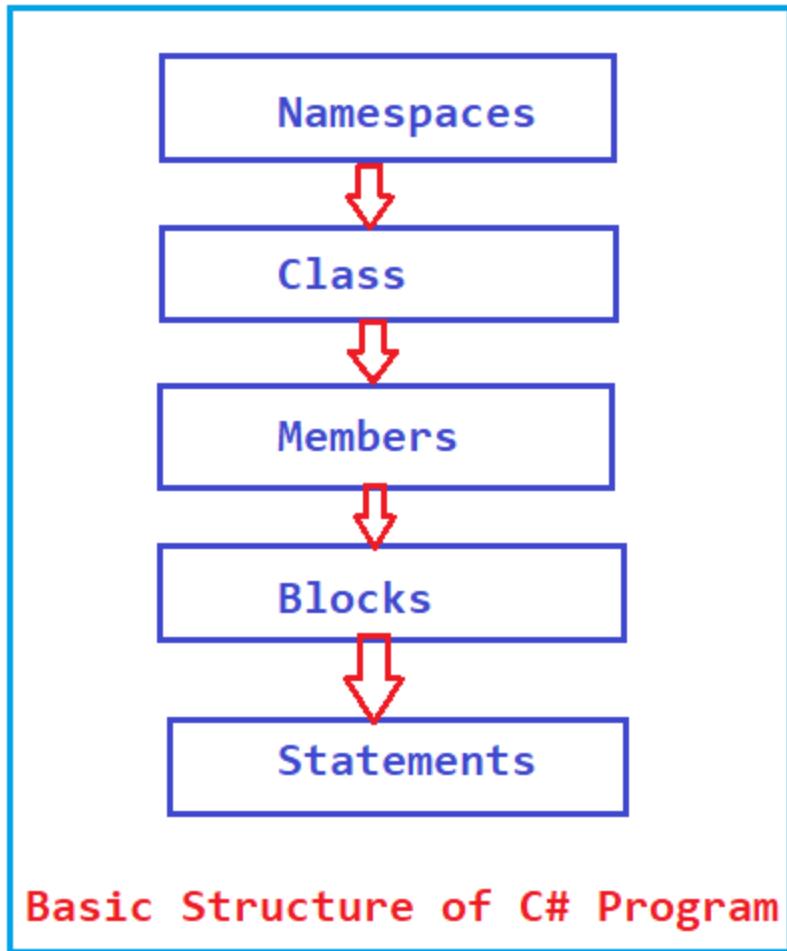
```
//Include classlibraries/namespaces
using System;

//Declare namespace
namespace namespacename
{
    //Class Declaration
    class ClassName
    {
        //Data Members
        int SomeMember1;
        float SomeMember2;

        //Functions/Methods/Blocks
        static void SomeBlock()
        {
            //Statements
            Console.WriteLine("Hello World");
        }
    }
}
```

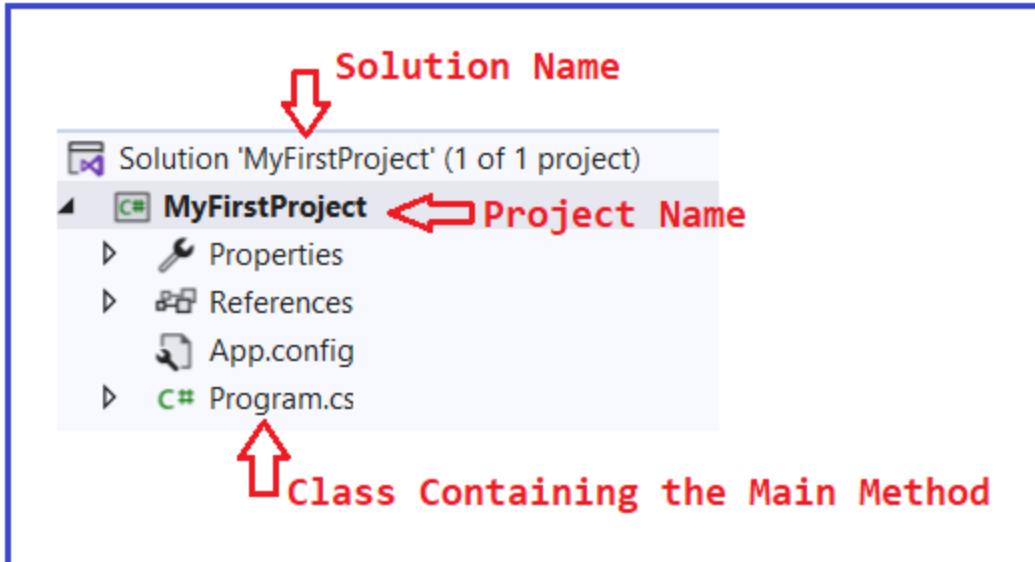
image_21.png

The above process is shown in the below diagram.



image_22.png

! Note: C#.NET is a Case-Sensitive Language and Every Statement in C# should end with a Semicolon.



image_23.png

A project called **MYFirstProject** will be created in Visual Studio. This project will contain all the necessary required files (Properties, References, App.Config files) to run the Console application. The Main program called **Program.cs** is the default code file that is created when a new console application is created in Visual Studio it contains the Main method by default and from that Main method our application is going to start its execution, but if you want you can also change this default behavior. So, if you look at the **Program.cs** class file, then you will see the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyFirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

image_24.png

Understanding the Code:

Using Visual Studio, if we are creating a console application (excluding .NET 6), then automatically we are getting four sections which are shown in the below image.

The diagram illustrates the four sections of a C# console application code, each highlighted with a red box and labeled with a blue arrow pointing to its description:

- Importing Namespaces Section:** Contains the code: `using System;`, `using System.Collections.Generic;`, `using System.Linq;`, `using System.Text;`, and `using System.Threading.Tasks;`.
- Namespace Declaration Section:** Contains the code: `namespace MyFirstProject` and its opening brace {.
- Class Declaration Section:** Contains the code: `internal class Program` and its opening brace {.
- Main Method Section:** Contains the code: `static void Main(string[] args)`, its opening brace {, the code `Console.WriteLine("Welcome to C#.NET");`, and the code `Console.ReadKey();`, all enclosed within a single large red box.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyFirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to C#.NET");
            Console.ReadKey();
        }
    }
}
```

image_25.png

Importing Namespace Section:

This section contains importing statements that are used to import the BCL (Base Class Libraries) as well as user-defined namespaces if required. This is similar to the included statements in the C programming language. Suppose you want to use some classes and interfaces in your code, then you have to include the namespace(s) from where these classes and interfaces are defined. For example, if you are going to use the Console class in your code, then you have to include the System namespace as the Console class belongs to the System namespace.

```
# Syntax: using NamespaceName;  
# Example:  
using System;
```

If the required namespace is a member of another namespace, we have to specify the parent and child namespaces separated by a dot as follows:

```
using System.Data;  
using System.IO;
```

⚠ Note: A namespace is a container that contains a group of related classes and interfaces, as well as, a namespace can also contain other namespaces.

Namespace Declaration Section:

Here a user-defined namespace is declared. In .NET applications, all classes and interfaces or any type related to the project should be declared inside some namespace. Generally, we put all the related classes under one namespace and in a project, we can create multiple namespaces.

```
# Syntax: namespace NamespaceName {}  
# Example:  
namespace MyFirstProject {}
```

Generally, the namespace name will be the same as the project name but it is not mandatory, you can give any user-defined name to the namespace.

Class Declaration Section:

For every Desktop Application in .NET, we need a start-up class file. For example, for every .NET Desktop Application like Console and Windows, there should be a Start-Up class that should have the Main method from where the program execution is going to start. When we create a Console Application using Visual Studio, by default, Visual Studio will Create the Start-Up class file with the name **Program.cs**

```
# Syntax:  
class ClassName  
{  
}
```

```
# Example:  
class Program  
{  
}
```

⚠ Note: You can change this default behavior. You can create your own class and you can also make that class as the Start-Up Class by including the Main method. In our upcoming articles, we will discuss this in detail.

Main() Method Section:

The main() method is the entry point or starting point of the application to start its execution. When the application starts executing, the main method will be the first block of the application to be executed. The Main method contains the main logic of the application.

What is using?

Using is a keyword. Using this keyword, we can refer to .NET BCL in C# Applications i.e. including the BCL namespaces as well as we can also include user-defined namespaces that we will discuss as we progress in this course. Apart from importing the namespace, other uses of using statements are there, which we will also discuss as progress in this course. For now, it is enough.

Methods and Properties of Console Class in C#

What is Console Class in C#?

In order to implement the user interface in console applications, Microsoft provided us with a class called `Console`. The `Console` class is available in the `System` namespace. This `Console` class provides some methods and properties using which we can implement the user interface in a console application.

In other words, if we want to work with the console window either for taking user input or to show the output, we are provided with the `Console` in C#.

According to Microsoft documentation the `Console` class represents the standard input, output, and error streams for console applications and this class cannot be inherited because it is a static class i.e. declared as static as shown in the below image.

Properties of Console Class in C#:

There are many properties available in the `Console` class. Some of them are as follows:

- **Title:** It gets or sets the title to display in the console title bar. It returns the string to be displayed in the title bar of the console. The maximum length of the title string is 24500 characters.
- **BackgroundColor:** It gets or sets the background color of the console. It returns a value that specifies the background color of the console; that is, the color that appears behind each character. The default is black.
- **ForegroundColor:** It gets or sets the foreground color of the console. It returns a `ConsoleColor` that specifies the foreground color of the console; that is, the color of each character that is displayed. The default is gray.
- **CursorPosition:** It gets or sets the height of the cursor within a character cell. It returns the size of the cursor expressed as a percentage of the height of a character cell. The property value ranges from 1 to 100.

Methods of Console Class in C#:

There are many methods available in the Console class. Some of them are as follows:

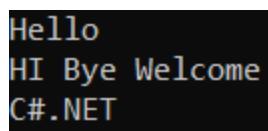
- **Clear():** It is used to clear the console buffer and corresponding console window of display information. In simple words, it is used to clear the screen.
- **Beep():** This method plays the sound of a beep through the console speaker. That means it plays a beep sound using a PC speaker at runtime.
- **ResetColor():** This method is used to set the foreground and background console colors to their defaults.
- **Write("string"):** This method is used to write the specified string value to the standard output stream.
- **WriteLine("string"):** This method is used to write the specified string value, followed by the current line terminator, to the standard output stream. That means this method same as the write method but automatically moves the cursor to the next line after printing the message.
- **Write(variable):** This method is used to write the value of the given variable to the standard output stream.
- **WriteLine(variable):** This method is used to write the value of the given variable to the standard output stream along with moving the cursor to the next line after printing the value of the variable.
- **Read():** This method read a single character from the keyboard and returns its ASCII value. The Datatype should be int as it returns the ASCII value.
- **ReadLine():** This method reads a string value from the keyboard and returns the entered value only. As it returns the entered string value so the DataType is going to be a string.
- **.ReadKey():** This method reads a single character from the keyboard and returns that character information like what key has been entered, and what its corresponding ASCII value is. The Datatype should be ConsoleKeyInfo which contains the entered key information

Example to show the use of the Write and WriteLine method in C#:

```
//Program to show the use of WriteLine and Write Method
//First Import the System namespace as the
//Console class belongs to System namespace
using System;
namespace MyFirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
            //We can access WriteLine and Write method using class name
            //as these methods are static

            //WriteLine Method Print the value and move the cursor to
            //the next line
            Console.WriteLine("Hello");
            //Write Method Print the value and stay in the same line
            Console.Write("HI ");
            //Write Method Print the value and stay in the same line
            Console.Write("Bye ");
            //WriteLine Method Print the value and move the cursor to
            //the next line
            Console.WriteLine("Welcome");
            //Write Method Print the value and stay in the same line
            Console.Write("C#.NET ");
            Console.ReadKey();
        }
    }
}
```

Output:



```
Hello
HI Bye Welcome
C#.NET
```

image_26.png

Example to show how to print the value of a variable in C#.

```
//Program to show how to print the value of a variable
using System;
namespace MyFirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string name = "Pranaya";
            Console.WriteLine(name);
            Console.WriteLine("Hello " + name);
            Console.Write($"Hello {name}");
            Console.ReadKey();
        }
    }
}
```

Output:



```
Pranaya
Hello Pranaya
Hello Pranaya
```

image_27.png

Reading Value from the user in C#:

```
//Program to show how to read value at runtime
using System;
namespace MyFirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
```

```

//Giving one message to the user to enter his name
Console.WriteLine("Enter Your Name");

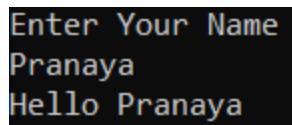
//ReadLine method reads a string value from the keyboard
string name = Console.ReadLine();

//Printing the entered string in the console
Console.WriteLine($"Hello {name}");
Console.ReadKey();
}

}

```

Output:



```

Enter Your Name
Pranaya
Hello Pranaya

```

image_28.png

How do you Read Integer Numbers in C# from the keyword?

Whenever we entered anything whether a string or numeric value from the keyword using the ReadLine method, the input stream is taking it as a string. So, we can directly store the input values in a string variable. If you want to store the input values in integer variables, then we need to convert the string values to integer values.

```

//Program to show how to read integer values
using System;
namespace MyFirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Eneter two Numbers:");
            //Converting string to Integer

```

```

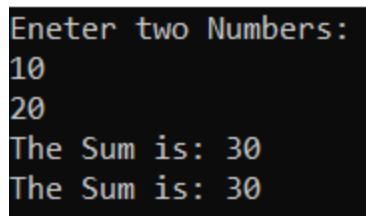
        int Number1 = Convert.ToInt32(Console.ReadLine());

        //Converting string to Integer
        int Number2 = Convert.ToInt32(Console.ReadLine());

        int Result = Number1 + Number2;
        Console.WriteLine($"The Sum is: {Result}");
        Console.WriteLine($"The Sum is: {Number1 + Number2}");
        Console.ReadKey();
    }
}

```

Output:



```

Eneter two Numbers:
10
20
The Sum is: 30
The Sum is: 30

```

image_29.png

Example to Show to Student Mark using Console Class Methods:

Write a Program to enter the Student Registration Number, Name, Mark1, Mark2, Mark3, and calculate the total mark and average marks and then print the student details in the console.

```

using System;
namespace MyFirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
            //Ask the user to Enter Student Details
            Console.WriteLine("Enter Student Details");
            Console.WriteLine("Enter Registration Number");

```

```

int RegdNumber = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Enter Name");
string Name = Console.ReadLine();
Console.WriteLine("Enter Marks of three Subjects:");
Console.WriteLine("Subject1");
int Mark1 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Subject2");
int Mark2 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Subject3");
int Mark3 = Convert.ToInt32(Console.ReadLine());

int TotalMarks = Mark1 + Mark2 + Mark3;
int AverageMark = TotalMarks / 3;

//Display the Student Details
Console.WriteLine("\nStudent Details are as Follows:");
Console.WriteLine($"Registration Number: {RegdNumber}");
Console.WriteLine($"Name: {Name}");
Console.WriteLine($"Total Marks : {TotalMarks}");
Console.WriteLine($"Average Mark: {AverageMark}");
Console.ReadKey();
}

}
}

```

Output:

```
Enter Student Details
Enter Registration Number
10001
Enter Name
Pranaya
Enter Marks of three Subjects:
Subject1
90
Subject2
60
Subject3
75

Student Details are as Follows:
Registration Number: 10001
Name: Pranaya
Total Marks : 225
Average Mark: 75
```

image_30.png

Var Keyword in C#

- **Type Inference:** `var` is used when you want the compiler to automatically *infer the type of a variable based on the value assigned to it.*
- **Cannot Be Changed:** Once a variable is declared using var, its type is fixed and cannot be changed later.
- **Must Be Initialized:** When using var, the variable must be initialized at the time of declaration, as the compiler infers the type from the initialization.

Examples:

```
var age = 25; // Inferred as int
var name = "John"; // Inferred as string
var list = new List<int>(); // Inferred as List<int>
```



`var` is not a data type: It's a keyword that tells the compiler to infer the type.



`Type Safety:` The type is still statically determined at compile-time, so var is type-safe.

Const Keyword in C#

- **Constant Value:** `const` is used to declare a constant value that cannot be changed after it is initialized.
- **Compile-time Constant:** The value of a `const` is determined at compile-time, and it must be assigned a value when declared.
- **Type:** A `const` can be of any data type (int, string, etc.), and its value must be assigned at the time of declaration.
- **Examples:**

```
const double Pi = 3.14;  
const int MaxValue = 100;  
const string Greeting = "Hello, World!";
```

Key Points:

- **const is implicitly static:** All constants are static by default, so you don't need to explicitly specify `static` when declaring them.
- **Cannot Be Modified:** Once a `const` value is set, it cannot be changed throughout the program.
- **Use Case:** Useful for values that are known at compile-time and should not change, such as mathematical constants or configuration values.

Comparison:

- **var:** Used for type inference, and the type is still determined at compile time. The value can change.
- **const:** Used to declare constants with a fixed value that cannot change after initialization.

Let me know if you'd like more details or examples!

Data Types in C#

.NET Developers use limited data types. See, you will see that most of the time as a .NET developer, we are acquainted to use int, bool, double, string, and Datetime data types. These five data types are mostly used by .NET Developers.

Why do we need Data Types in C#?

The Datatypes in C# are basically used to store the data temporarily in the computer through a program. In the real world, we have different types of data like integers, floating-point, characters, boolean, strings, etc. To store all these different kinds of data in a program to perform business-related operations, we need the data types.

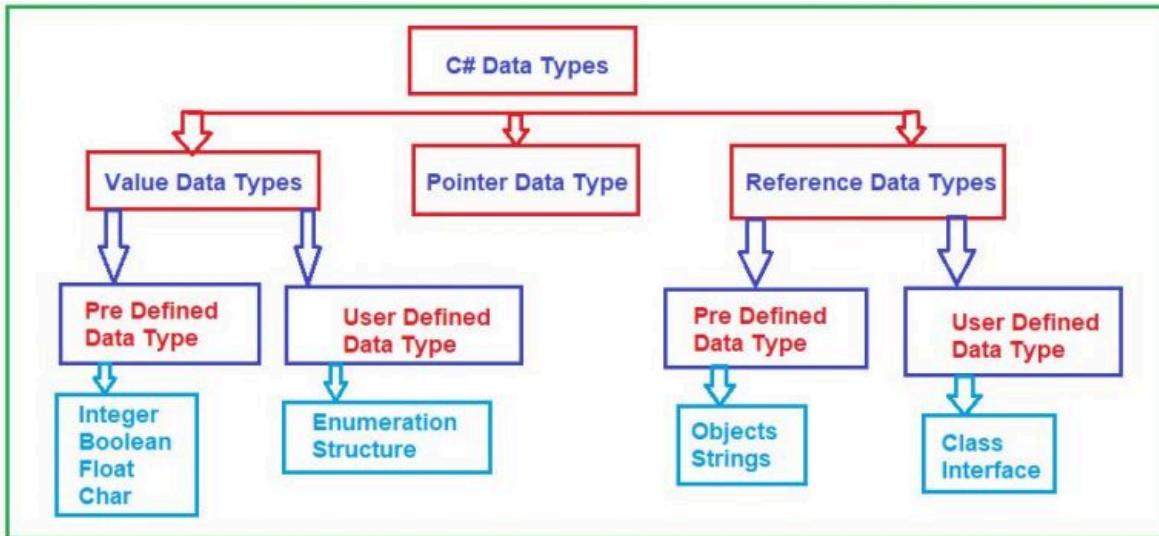
What is a Data Type in C#?

The Datatypes are something that gives information about

1. **Size** of the memory location.
2. The **Range of data** that can be stored inside that memory location
3. Possible **Legal Operations** that can be performed on that memory location.
4. What **Types of Results** come out from an expression when these types are used inside that expression? The keyword which gives all the above information is called the data type in C#.

What are the Different Types of Data types Available in C#?

A data type in C# specifies the type of data that a variable can store such as integer, floating, boolean, character, string, etc. The following diagram shows the different types of data types available in C#.



image_31.png

There are 3 types of data types available in the C# language.

1. Value Data Types: Byte, SByte, Char, String
2. Reference Data Types
3. Pointer Data Types

Value Data Types in C#

Value data types directly hold their data in memory. When you assign a value to a variable of a value type, a copy of the actual data is stored in the variable. There are two subcategories of value data types:

1. Predefined Data Types: These are built-in types provided by C#:

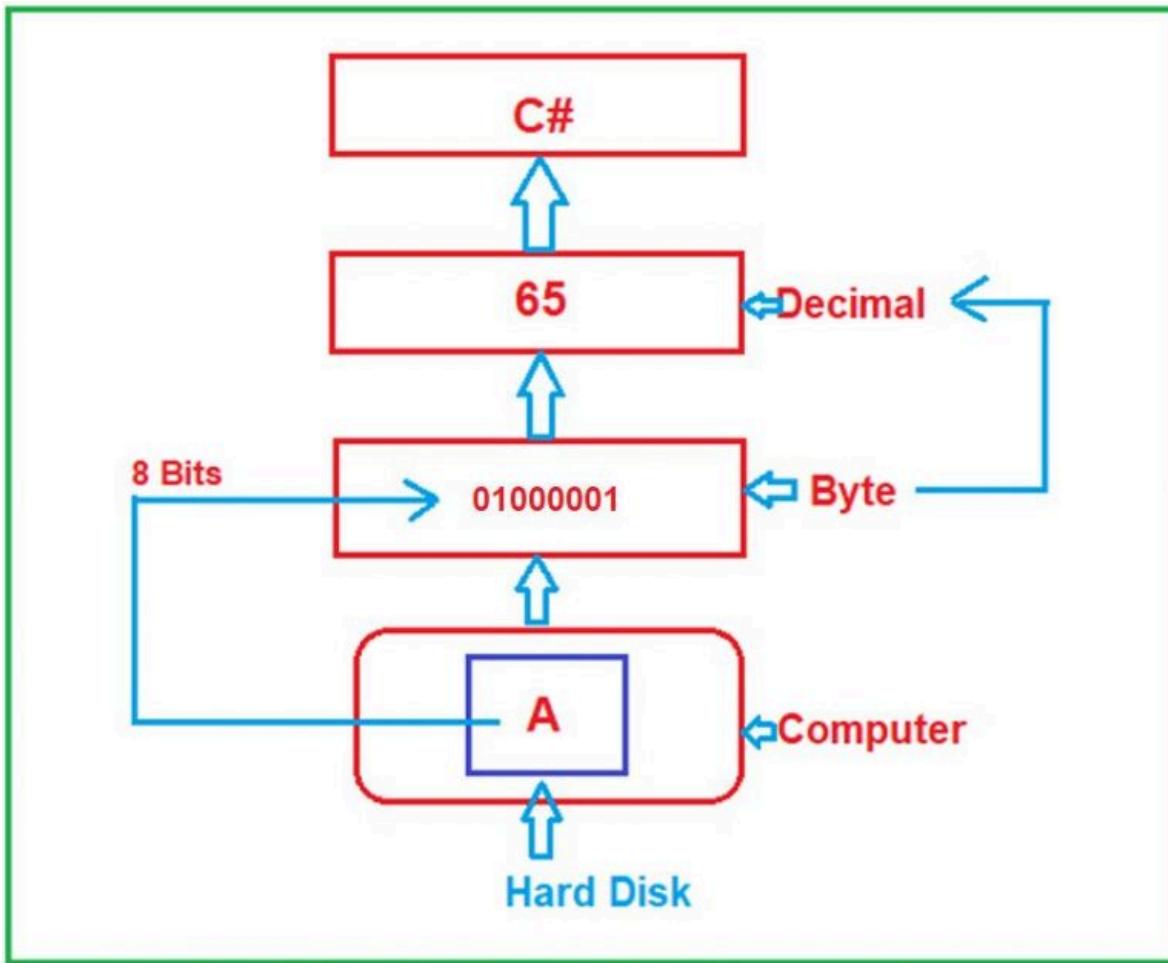
- Integer: Represents whole numbers (e.g., `int`, `long`).
- Boolean: Represents true or false values (`bool`).
- Float: Represents floating-point numbers with single precision (`float`).
- Char: Represents a single character (`char`).

2. User-Defined Data Types: These types are defined by the programmer. Two common user-defined value data types are:

- Enumeration (`enum`): Used to define a set of named integral constants. For example, representing the days of the week.
- Structure (`struct`): A value type that can contain multiple variables of different data types. For example, creating a structure to represent a point with x and y coordinates.

How Data is Represented in a Computer?

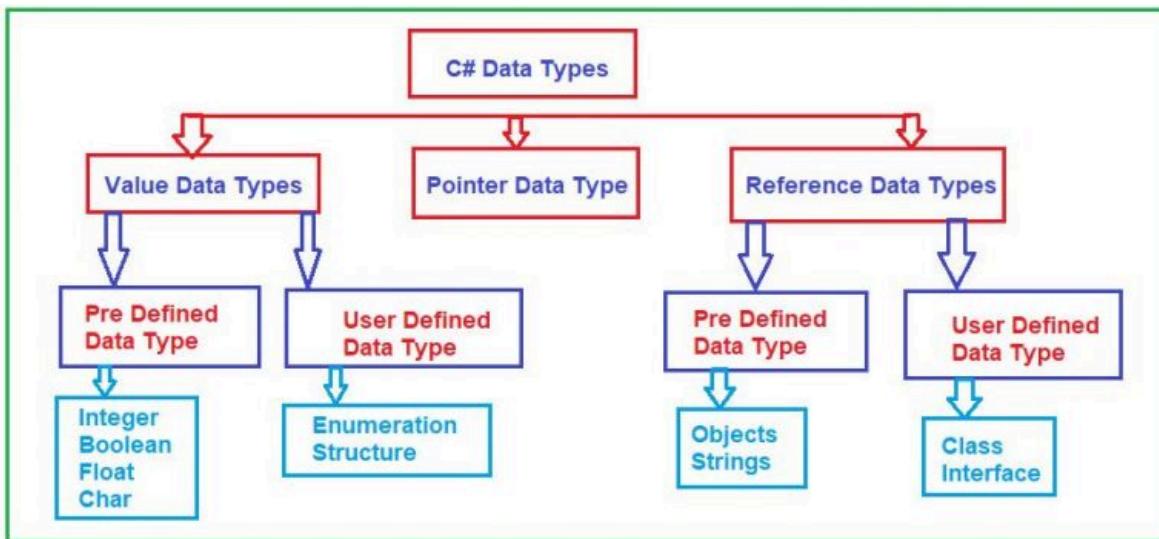
Please have a look at the below diagram



image_32.png

In essence, the computer stores and processes data using binary (0s and 1s), but we, as developers, often work with higher-level formats like decimal for easier understanding. In C#, we typically work with decimal numbers instead of binary. So, we convert the binary number to its decimal equivalent. Internally, the computer then converts this decimal back into binary (byte) format to process and store the data.

Predefined Data types



image_31.png

Numeric Numbers with Decimal (Single, Double, Decimal)

In C#, numeric values with decimals are represented by three main data types:

1. **Single** (single-precision floating-point number)
2. **Double** (double-precision floating-point number)
3. **Decimal** (represents a decimal floating-point number)

These data types differ primarily in the amount of memory they consume and their precision:

- **Single**: Takes 4 bytes of memory
- **Double**: Takes 8 bytes of memory
- **Decimal**: Takes 16 bytes of memory

Why Add f and m Suffixes?

In C#, when you write a decimal value, the compiler may default to interpreting it as a **double**. To avoid confusion and to specify the exact type, you can use suffixes:

- **f**: This suffix is used to indicate that the number is a **Single** (float) type. Without this suffix, the compiler will assume the value is a **double** by default.
- **m**: This suffix is used to indicate that the number is a **Decimal** type. Without this suffix, the compiler assumes the number is a **double**.

Example Code:

```
Single a = 1.123f; // Single (float) value with f suffix
Double b = 1.456; // Double value (default, no suffix)
Decimal c = 1.789M; // Decimal value with m suffix
```

```
Console.WriteLine($"Single Size: {sizeof(Single)} Byte");
Console.WriteLine($"Single Min Value: {Single.MinValue} and Max Value:
{Single.MaxValue}");
Console.WriteLine($"Double Size: {sizeof(Double)} Byte");
Console.WriteLine($"Double Min Value: {Double.MinValue} and Max Value:
{Double.MaxValue}");
Console.WriteLine($"Decimal Size: {sizeof(Decimal)} Byte");
Console.WriteLine($"Decimal Min Value: {Decimal.MinValue} and Max Value:
{Decimal.MaxValue}");
Console.ReadKey();
```

Output:

```
Single Size: 4 Byte
Single Min Value: -3.402823E+38 and Max Value: 3.402823E+38
Double Size: 8 Byte
Double Min Value: -1.7976931348623157E+308 and Max Value:
1.7976931348623157E+308
Decimal Size: 16 Byte
Decimal Min Value: -79228162514264337593543950335 and Max Value:
79228162514264337593543950335
```

Explanation of the Code:

- Single a = 1.123f;: The f suffix specifies that a is a Single (float).
- Double b = 1.456;: No suffix is used, so this is interpreted as a Double by default.
- Decimal c = 1.789M;: The M suffix specifies that c is a Decimal.

Short-Hand Names:

Instead of using Single, Double, and Decimal, you can use the short-hand names for these types in C#:

- float for Single
- double for Double

- **decimal** for Decimal

Here's how you can use the shorthand names:

```
float a = 1.123f; // float (Single) type
double b = 1.456; // double (Double) type
decimal c = 1.789M; // decimal (Decimal) type

Console.WriteLine($"float Size: {sizeof(float)} Byte");
Console.WriteLine($"float Min Value: {float.MinValue} and Max Value:
{float.MaxValue}");
Console.WriteLine($"double Size: {sizeof(double)} Byte");
Console.WriteLine($"double Min Value: {double.MinValue} and Max Value:
{double.MaxValue}");
Console.WriteLine($"decimal Size: {sizeof(decimal)} Byte");
Console.WriteLine($"decimal Min Value: {decimal.MinValue} and Max Value:
{decimal.MaxValue}");
Console.ReadKey();
```

Output will be the same as the earlier example, but using short-hand names for types.

Summary:

Data Type	Size	Range (Positive)	Range (Negative)	Default Suffix
Single	4 bytes	0 to 3.402823E+38	-3.402823E+38 to -1.401298E-45	f
Double	8 bytes	0 to 1.7976931348623157E+308	-1.7976931348623157E+308 to -4.9406564584124654E-324	No suffix
Decimal	16 bytes	0 to 79228162514264337593543950335	-79228162514264337593543950335	m



- **Use f for Single:** When specifying a `Single` (float) value.
- **Use m for Decimal:** When specifying a `Decimal` value.
- **No suffix means Double:** If no suffix is provided, the value defaults to `double`.
- **Choosing the Right Type:** Use `Single` for less memory usage and moderate precision, `Double` for larger numbers and higher precision, and `Decimal` when you need very high precision (such as for financial calculations).

Integers Data types (Numbers without Decimal)

A Integers are Numbers without Decimal

In this category, the .NET Framework provides three kinds of integer data types. They are as follows:

1. **16-Bit Signed Numeric:** Example: `Int16`
2. **32-Bit Signed Numeric:** Example: `Int32`
3. **64-Bit Signed Numeric:** Example: `Int64`

As the above data types are signed, they can store both positive and negative numbers. The size of the number they can hold varies based on the data type.

16-Bit Signed Numeric (`Int16`)

The `Int16` data type is a 16-bit signed integer. It can store 65,536 values. Since it is signed, half of these values are negative, and half are positive.

- **Positive Range:** 0 to 32,767
- **Negative Range:** -1 to -32,768

```
const Int16 MaxInt16Value = 32767;
const Int16 MinInt16Value = -32768;

Console.WriteLine(MaxInt16Value);
Console.WriteLine(MinInt16Value);
Console.WriteLine($"the type of Int16: {typeof(Int16)}");
```

Output:

```
32767  
-32768  
the type of Int16 : System.Int16
```

32-Bit Signed Numeric (Int32)

The `Int32` data type is a 32-bit signed integer, which allows it to store a wider range of values than `Int16`. It can represent both positive and negative numbers.

- **Positive Range:** 0 to 2,147,483,647
- **Negative Range:** -1 to -2,147,483,648

```
const Int32 MaxInt32Value = 2147483647;  
const Int32 MinInt32Value = -2147483648;  
  
Console.WriteLine(MaxInt32Value);  
Console.WriteLine(MinInt32Value);  
Console.WriteLine($"the type of Int32: {typeof(Int32)}");
```

Output:

```
2147483647  
-2147483648  
the type of Int32 : System.Int32
```

64-Bit Signed Numeric (Int64)

The `Int64` data type is a 64-bit signed integer. This data type has an even larger range, making it suitable for situations where a large number must be stored.

- **Positive Range:** 0 to 9,223,372,036,854,775,807
- **Negative Range:** -1 to -9,223,372,036,854,775,808

```

const Int64 MaxInt64Value = 9223372036854775807;
const Int64 MinInt64Value = -9223372036854775808;

Console.WriteLine(MaxInt64Value);
Console.WriteLine(MinInt64Value);
Console.WriteLine($"the type of Int64: {typeof(Int64)}");

```

Output:

```

9223372036854775807
-9223372036854775808
the type of Int64 : System.Int64

```

Summary of Integer Data Types in C\#

Data Type	Size	Range (Positive)	Range (Negative)
Int16	16-bit	0 to 32,767	-1 to -32,768
Int32	32-bit	0 to 2,147,483,647	-1 to -2,147,483,648
Int64	64-bit	0 to 9,223,372,036,854,775,807	-1 to -9,223,372,036,854,775,808

Notes:

- Choosing the Right Data Type:** When working with integers in C#, choose the appropriate data type based on the range of numbers you need to store. For most applications, `Int32` or `Int64` will suffice, but for very large values, `Int64` is typically used.
- Signed vs. Unsigned:** All of these integer types are signed, meaning they can store both positive and negative values. If you only need to store positive values, you can

use unsigned versions like `UInt16`, `UInt32`, and `UInt64`.

This is the updated documentation focusing on the three main integer types in C#. Let me know if you'd like further details!

Char Data Type

Char is a 2-Byte length data type that can contain Unicode data. **What is Unicode?**

Unicode is a standard for character encoding and decoding for computers. We can use various Unicode encodings formats such as

- UTF-8(8-bit)
- UTF-16(16-bit),

and so on. As per the definition of char, it represents a character as a UTF-16 code unit. UTF-16 means 16-bit length which is nothing but 2-Bytes.

As char is 2-Byte length, so it will contain 216 numbers i.e. 65536. So, the minimum number is 0 and the maximum number is 65535. For a better understanding, please have a look at the below example.

```
char ch = 'B';
Console.WriteLine($"Char: {ch}");
Console.WriteLine($"Equivalent Number: {(byte)ch}");
Console.WriteLine($"Char Minimum: {(int)char.MinValue} and Maximum:
{(int)char.MaxValue}");
Console.WriteLine($"Char Size: {sizeof(char)} Byte");
Console.ReadKey();
```

output will be

```
Char: B
Equivalent Number: 66
Char Minimum: 0 and Maximum: 65535
Char Size: 2 Byte
```

Byte Data Type

Is used to represent an 8-Bit unsigned integer.

 Unsigned means only positive values

```
const Byte MaxValue = 255;  
const Byte MinValue = 0;  
  
Console.WriteLine(MaxValue);  
Console.WriteLine(MinValue);
```

Now, if you want to store both positive and negative values, then you need to use the signed byte data type i.e. SByte

SByte Data Type in C#

SByte data type represents an 8-bit signed integer. Then what will be the maximum and minimum values? Remember when a data type is signed, then it can hold both positive and negative values. In that case, the maximum needs to be divided by two i.e. $256/2$ which is 128. So, it will store 128 positive numbers and 128 negative numbers

```
const SByte MaxValue = 127;  
const SByte MinValue = -128;  
  
Console.WriteLine(MaxValue);  
Console.WriteLine(MinValue);
```

Boolean Data Type

The **Boolean** data type in C# is used to represent **truth values**, which can either be **true** or **false**. It is one of the simplest and most fundamental data types, and it is primarily used for conditional logic in programs.

The **Boolean** data type in C# is represented by the keyword **bool**.

Key Points:

1. **Value:** A **bool** can only hold two values: **true** or **false**.
2. **Default Value:** By default, a **bool** variable is initialized to **false**.
3. **Size:** A **bool** typically takes 1 byte of memory, though the exact size might vary based on the platform and implementation.
4. **Usage:** It is commonly used in conditions, logical operations, and control flow (e.g., **if**, **while** statements).

Declaring a Boolean Variable:

```
bool isActive = true; // Declaration and initialization  
bool isCompleted = false;
```

Basic Boolean Operation

```
bool isDay = true;  
bool isNight = false;  
  
if (isDay)  
{  
    Console.WriteLine("It's day time!");  
}  
else  
{
```

```
        Console.WriteLine("It's night time!");
    }

// Output: It's day time!
```

Explanation:

- `isDay` is set to `true`, so the program prints "It's day time!".
- If the condition `isDay` were `false`, it would print "It's night time!".

Using Boolean in Conditions

```
bool isRaining = true;

if (isRaining)
{
    Console.WriteLine("You need an umbrella!");
}
else
{
    Console.WriteLine("No umbrella needed.");
}

// Output: You need an umbrella!
```

Explanation:

- The boolean variable `isRaining` is used in an `if` statement to decide whether to suggest bringing an umbrella.

Boolean Expressions and Logical Operations

Boolean values are often the result of **logical expressions** and can be combined using logical operators like **AND** (`&&`), **OR** (`||`), and **NOT** (`!`).

```

bool isSunny = true;
bool isWeekend = false;

// AND operator (both conditions must be true)
bool canGoToPark = isSunny && isWeekend;
Console.WriteLine($"Can go to the park: {canGoToPark}"); // Output:
False

// OR operator (at least one condition must be true)
bool isHoliday = isSunny || isWeekend;
Console.WriteLine($"It's a holiday: {isHoliday}"); // Output: True

// NOT operator (reverses the value)
bool isNotRaining = !isSunny;
Console.WriteLine($"It is not raining: {isNotRaining}"); // Output:
False

```

Explanation:

- The `&&` (AND) operator checks if both conditions are true.
- The `||` (OR) operator checks if at least one condition is true.
- The `!` (NOT) operator negates the boolean value.

Boolean as Return Type in Methods

You can use the `bool` type as a return value in methods to indicate success, failure, or whether a certain condition is met.

```

static bool IsAdult(int age)
{
    return age >= 18; // Returns true if age is 18 or greater
}

```

Explanation:

- The method `IsAdult` takes an integer `age` and returns `true` if the age is 18 or older, otherwise `false`.

Boolean in Loops

The `bool` type is commonly used in loops to control when the loop should stop or continue.

```
bool isRunning = true;
int count = 0;

while (isRunning)
{
    Console.WriteLine($"Count: {count}");
    count++;

    if (count >= 5)
    {
        isRunning = false; // Exit the loop when count reaches 5
    }
}

// Output:
// Count: 0
// Count: 1
// Count: 2
// Count: 3
// Count: 4
```

Explanation:

- The `while` loop continues to run as long as `isRunning` is `true`. When the `count` reaches 5, `isRunning` is set to `false`, causing the loop to exit.

Summary of Boolean Data Type

Feature	bool (Boolean)		
Value	true or false		
Default Value	false (when not explicitly initialized)		
Size	1 byte		
Use Cases	Conditions, logical operations, flags, control flow		
Supported Operations	AND (&&), OR (), NOT (!)		

When to Use Boolean:

- **Control Flow:** Used to control the flow of execution with conditional statements (**if**, **else**, **switch**).
- **Flags:** Used as flags to track the state of certain conditions (e.g., **isValid**, **isRunning**).
- **Logical Decisions:** Used in logical expressions to combine multiple conditions and make decisions based on them.

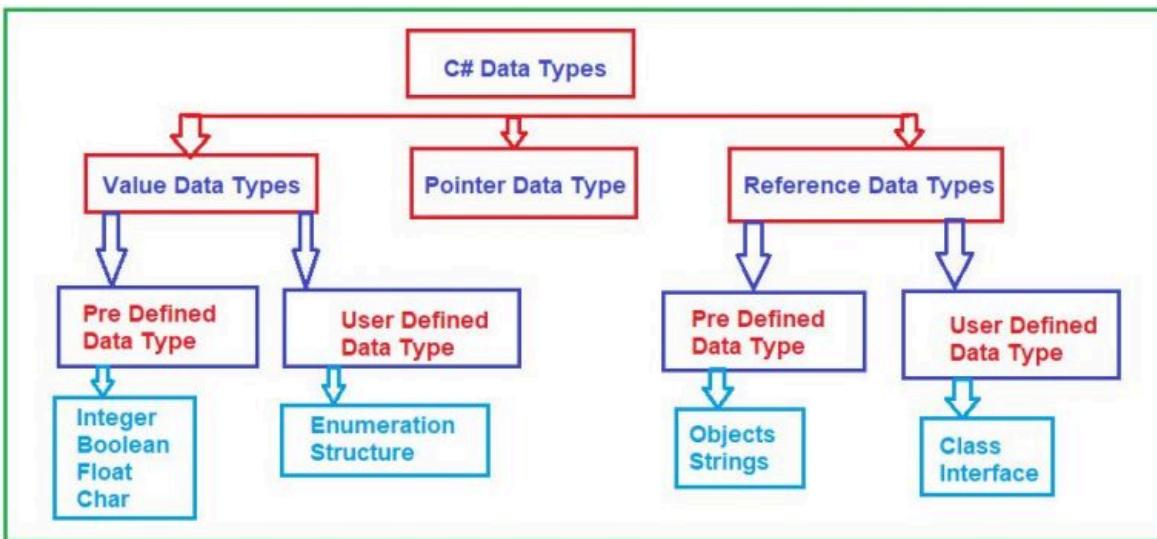
Please Note

The **bool** data type is a fundamental part of decision-making and logic in C#. It is widely used for:



- Conditional checks
- Controlling the flow of the program
- Representing binary true/false states

User Defined Data Type



image_31.png

These types are defined by the programmer. Two common user-defined value data types are:

- **Enumeration** (enum)
- **Structure** (struct)

Enums in C#

What is an Enum?

An **Enum** (short for "enumeration") is a special value type in C# that allows you to define a set of named constants. These constants are typically used to represent a collection of related values in a more readable and maintainable manner.

Enums are ideal for use cases where you need to define a collection of options or states that are represented by a numeric value, but you want to use descriptive names instead of raw numbers.

Declaration of Enum

An enum is declared using the `enum` keyword followed by the name of the enum and the list of constant values it will hold. By default, the underlying type of the enum is `int`, but

you can specify other integral types such as byte, short, long, etc.

Syntax:

```
enum EnumName : EnumType
{
    Value1,
    Value2,
    Value3,
    ...
}
```

- **EnumName**: The name of the enumeration.
- **EnumType**: The underlying data type (optional; defaults to **int**).
- **Value1, Value2, Value3, ...**: The named constants within the enum.

Example of Enum:

```
// Defining an Enum for Days of the Week
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday }

class Program
{
    static void Main(string[] args)
    {
        // Assigning an Enum value
        Day today = Day.Monday;

        // Using Enum in a switch case
        switch (today)
        {
            case Day.Monday:
                Console.WriteLine("Start of the work week.");
                break;
            case Day.Friday:
```

```

        Console.WriteLine("End of the work week.");
        break;
    default:
        Console.WriteLine("It's a regular day.");
        break;
    }

    // Getting the numeric value of the Enum
    Console.WriteLine($"Numeric value of {today}: {(int)today}");
}
}

```

Output:

```

Start of the work week.
Numeric value of Monday: 0

```

Enum Values and Numeric Representation

By default, the first value in an enum is assigned 0, and each subsequent value is incremented by 1. However, you can explicitly assign values to the enum members.

Example with Explicit Values:

```

enum Day { Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday
= 5 }

class Program
{
    static void Main(string[] args)
    {
        Day today = Day.Wednesday;
        Console.WriteLine($"Numeric value of {today}: {(int)today}");
    }
}

```

Output:

```
Numeric value of Wednesday: 3
```

Why Use Enums?

- **Improves Code Readability:** Enums allow you to use descriptive names instead of arbitrary numbers.
- **Prevents Invalid Values:** Enums limit the possible values to a predefined set of constants, preventing the use of invalid data.

Structs in C#

What is a Struct?

A **Struct** in C# is a value type that allows you to define a complex data type. Structs are similar to classes but have a few key differences. They are often used when you need to group related data together, and you want that data to be passed around by value (copying the entire struct).

Unlike classes, which are reference types, structs are **value types**, meaning they are stored on the stack and passed by value. Structs can contain fields, methods, properties, and constructors.

Declaration of Struct

A struct is defined using the `struct` keyword, and it can contain members such as fields, methods, properties, and constructors.

Syntax:

```
struct StructName
{
    public DataType Field1;
    public DataType Field2;
    ...

    // Constructor
    public StructName(DataType value1, DataType value2)
    {
```

```
    Field1 = value1;
    Field2 = value2;
}
}
```

- StructName: The name of the struct.
- Field1, Field2, etc.: The fields (data members) of the struct.
- The constructor (optional) initializes the fields when a new instance of the struct is created.

Example of Struct:

```
// Defining a Struct to represent a Point in 2D space
struct Point
{
    public int X;
    public int Y;

    // Constructor for initialization
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Method to display point coordinates
    public void DisplayPoint()
    {
        Console.WriteLine($"Point coordinates: ({X}, {Y})");
    }
}

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

```
// Creating an instance of the struct
Point p = new Point(10, 20);
p.DisplayPoint(); // Output: Point coordinates: (10, 20)
}
}
```

Output:

```
Point coordinates: (10, 20)
```

Struct vs Class

- **Value Type:** Structs are value types, meaning when you assign a struct variable to another, a copy of the data is made.
- **Reference Type:** Classes are reference types, meaning they store references to the data and not the actual data itself.

Why Use Structs?

- **Performance:** Structs are more memory-efficient when dealing with small, simple data types that don't require inheritance or complex functionality.
- **No Inheritance:** Unlike classes, structs cannot be inherited or derive from another struct or class.

Differences Between Enums and Structs

Feature	Enum	Struct
Type	Special data type for named constants	User-defined value type
Purpose	To define a set of related constants	To group data into a single entity
Memory	Stores the numeric value of each constant	Stores values for each member (fields)
Default Type	int (default underlying type)	Can contain multiple fields of any type
Inheritance	Cannot inherit from other enums or types	Cannot inherit from another struct or class
Use Case	Days of the week, months of the year	Point, Rectangle, Customer information



- **Enums** are used to define a set of related constants with readable names instead of numbers, improving code clarity and maintainability.
- **Structs** are lightweight, value types used to group related data together, often used for simple data structures like points, rectangles, or any data that requires passing by value.

```
// Enumeration (User-defined value type)
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday }
```

```
// Structure (User-defined value type)
```

```
struct Point
{
    public int X;
    public int Y;
}

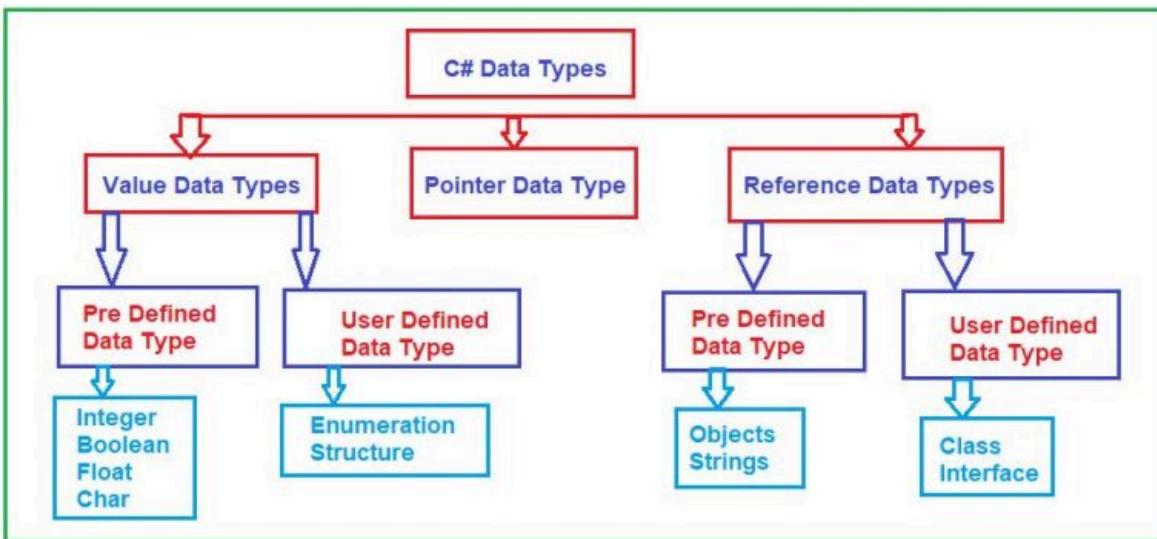
class Program
{
    static void Main(string[] args)
    {
        // Using Enum
        Day today = Day.Monday;
        Console.WriteLine("Today is: " + today);

        // Using Structure
        Point p = new Point();
        p.X = 10;
        p.Y = 20;
        Console.WriteLine($"Point: ({p.X}, {p.Y})");
    }
}
```

Output

```
Today is: Monday
Point: (10, 20)
```

Reference Data Types



image_31.png

Reference Data Types in C#

In C#, **reference data types** are types that store references (or pointers) to the actual data rather than the data itself. When you assign a reference type variable to another variable, they both point to the same memory location, meaning they share the same data.

Reference types are stored in the **heap** memory, and variables of reference types hold the memory address (reference) of where the data is stored, not the actual data. This is different from **value types** (like `int`, `float`, etc.), which store the actual data in memory.

Types of Reference Data Types in C#

1. Objects
2. Strings
3. Classes
4. Interfaces
5. Arrays

1. Objects in C#

- **Object** is the base type for all data types in C#. Every type, whether it's a built-in type or user-defined class, is derived from `System.Object`.
- **Object** can store any data type, including value types and reference types.

Example:

```
using System;

class Program
{
    static void Main()
    {
        // Storing a value type in an object
        int number = 10;
        object obj = number; // Boxing: value type stored in object

        // Storing a reference type in an object
        string message = "Hello, C#!";
        obj = message; // obj now holds a reference to the string

        Console.WriteLine(obj); // Output: Hello, C!
    }
}
```

Explanation:

- The `obj` variable can hold any type of data (here, an `int` and a `string`). It's a generic reference type that can store any object, but it requires casting to work with the original type.

2. Strings in C#

Strings in C# are reference types. They are immutable, which means once a string is created, it cannot be changed. Any operation that modifies a string will result in the creation of a new string.

Example:

```
using System;

class Program
{
    static void Main()
    {
        string str1 = "Hello";
        string str2 = str1;

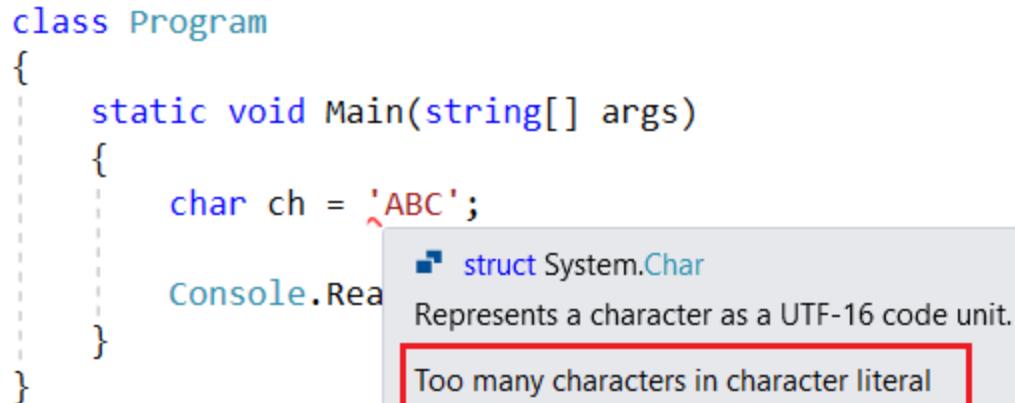
        str1 = "Goodbye"; // Reassigning str1

        Console.WriteLine(str1); // Output: Goodbye
        Console.WriteLine(str2); // Output: Hello
    }
}
```

Explanation:

- Even though `str2` was assigned `str1`, changing `str1` does not affect `str2`. This is because strings are immutable, and `str2` holds a reference to the original string `"Hello"`, while `str1` now points to a new string `"Goodbye"`.

In the previous Chapter, we discussed the `char` data type where we are storing a single character in it. Now, if I try to add multiple characters to a `char` data type, then I will get a compile time error as shown in the below image.



image_33.png

As you can see, here we are getting the error Too many characters in character literal. This means you cannot store multiple characters in the character literal. If you want to store multiple characters, then you need to use the string data type in C# as shown in the below example,

```
string str = "ABC";

Console.WriteLine(str);

Console.WriteLine($"The type of str is : {typeof(string)}");

Console.ReadKey();
```

output

```
ABC
The type of str is : System.String
```

3. Classes in C#

A **class** is a reference type that defines the structure of objects. When you create an instance of a class, the object is stored in the heap, and the variable holds a reference to that object.

Example:

```

using System;

class Person
{
    public string Name;
    public int Age;
}

class Program
{
    static void Main()
    {
        // Creating an instance of the Person class
        Person person1 = new Person();
        person1.Name = "Alice";
        person1.Age = 25;

        // Assigning reference of person1 to person2
        Person person2 = person1;

        // Modifying the object through person2
        person2.Name = "Bob";

        // Both person1 and person2 refer to the same object
        Console.WriteLine(person1.Name); // Output: Bob
        Console.WriteLine(person2.Name); // Output: Bob
    }
}

```

Explanation:

- `person1` and `person2` both refer to the same object in memory. Modifying the object through `person2` also affects `person1`, as they both point to the same memory location.

4. Interfaces in C#

An **interface** defines a contract that classes can implement. It is a reference type that specifies a set of methods or properties that a class must implement.

Example:

```
using System;

interface IDriveable
{
    void Drive();
}

class Car : IDriveable
{
    public void Drive()
    {
        Console.WriteLine("The car is driving.");
    }
}

class Program
{
    static void Main()
    {
        IDriveable myCar = new Car();
        myCar.Drive(); // Output: The car is driving.
    }
}
```

Explanation:

- The `Car` class implements the `IDriveable` interface, meaning it must provide an implementation for the `Drive` method. The `myCar` variable holds a reference to a `Car` object, and calling `Drive` on it triggers the `Drive` method of the `Car` class.

5. Arrays in C#

Arrays in C# are also reference types. They store multiple elements of the same type and are allocated on the heap. When you assign an array to another variable, both variables point to the same array in memory.

Example:

```
using System;

class Program
{
    static void Main()
    {
        int[] arr1 = { 1, 2, 3, 4 };
        int[] arr2 = arr1; // arr2 points to the same array as arr1

        arr2[0] = 10; // Modify the first element of the array

        Console.WriteLine(arr1[0]); // Output: 10 (arr1 and arr2 are
        pointing to the same array)
        Console.WriteLine(arr2[0]); // Output: 10
    }
}
```

Explanation:

- Both `arr1` and `arr2` reference the same array. Changing an element in `arr2` also changes it in `arr1` because they both point to the same memory location.

Summary of Reference Data Types

Type	Memory Location	Example Usage
Object	Heap	Can store any data type (e.g., int, string, etc.)
String	Heap	Represents a sequence of characters, immutable
Class	Heap	Defines objects with data and behavior
Interface	Heap	Defines a contract that classes can implement
Array	Heap	A collection of elements of the same type

Key Points:



- **Heap Memory:** Reference types are stored in heap memory, and variables hold a reference to the memory location.
- **Shared Data:** Assigning one reference type variable to another makes them both reference the same object in memory.
- **Object Types:** Classes, interfaces, and arrays are examples of reference types in C#.
- **Mutability:** Reference types allow the data they point to be modified, and changes are reflected across all references to that object.

Pointer Data Type

In C#, pointer types are used to store memory addresses, which directly point to the memory location where the data is stored.

⚠ However, C# is a managed language, meaning it handles memory management automatically.

- As a result, pointers are not commonly used in everyday C# programming.
- They are typically used in unsafe code contexts (i.e., code that runs outside the safety checks of the runtime) and are often utilized when performing low-level programming, such as interacting with memory directly or working with interop (calling native code).

Using Pointers in C#

To use pointers in C#, you need to:

1. Use the `unsafe` keyword to allow pointers.
2. Declare pointer types using the `*` operator.
3. Enable unsafe code in the project settings (in most IDEs, like Visual Studio, you must allow unsafe code in the project properties).

Syntax for Pointers:

```
unsafe
{
    // Pointer declaration
    int* ptr;
    int value = 10;

    // Storing address of a variable in the pointer
```

```
ptr = &value;

// Accessing value through the pointer
Console.WriteLine(*ptr); // Output: 10
}
```

Here, `&` is used to get the address of a variable, and `*` is used to dereference the pointer (i.e., access the value stored at the memory address).

Enabling Unsafe Code

In order to use pointers, you need to enable **unsafe code** in your project. If you're using Visual Studio:

1. Right-click on your project.
2. Go to **Properties**.
3. Under the **Build** tab, check **Allow unsafe code**.

Example 1: Using Pointers with Primitive Types

```
using System;

class Program
{
    static unsafe void Main()
    {
        int num = 10;
        int* ptr = &num; // Store the address of num in ptr

        Console.WriteLine($"Value of num: {num}");
        Console.WriteLine($"Address of num: {(int)ptr}");
        Console.WriteLine($"Value at pointer: {*ptr}"); // Dereferencing the pointer
    }
}
```

Output:

```
Value of num: 10
Address of num: <address_value>
Value at pointer: 10
```

Explanation:

- We use `&num` to get the memory address of the `num` variable.
- The pointer `ptr` holds that memory address.
- We dereference the pointer `*ptr` to get the value stored at the address.

Example 2: Modifying Data Through a Pointer

Pointers allow you to directly modify the value of a variable by accessing its memory address.

```
using System;

class Program
{
    static unsafe void Main()
    {
        int num = 5;
        int* ptr = &num; // Pointer to num

        Console.WriteLine($"Before: {num}"); // Output: 5

        // Modifying the value of num via the pointer
        *ptr = 20;

        Console.WriteLine($"After: {num}"); // Output: 20
    }
}
```

Output:

Before: 5
After: 20

Explanation:

- By dereferencing the pointer `*ptr`, we modify the value of `num` directly through its memory address.

Example 3: Using Pointers with Arrays

You can use pointers with arrays to work directly with the elements in memory.

```
using System;

class Program
{
    static unsafe void Main()
    {
        int[] arr = { 1, 2, 3, 4, 5 };

        // Getting the address of the first element
        fixed (int* ptr = arr)
        {
            // Accessing array elements using pointer arithmetic
            Console.WriteLine($"First element: {*ptr}"); // Output: 1
            ptr++; // Move the pointer to the next element
            Console.WriteLine($"Second element: {*ptr}"); // Output: 2
        }
    }
}
```

Output:

```
First element: 1
Second element: 2
```

Explanation:

- The `fixed` keyword is used to pin the memory address of the array so that the garbage collector doesn't move it during pointer operations.
- Pointer arithmetic (`ptr++`) allows us to move through the array elements in memory.

Important Notes on Pointers in C#

- **Unsafe Code:** Since pointers work directly with memory, this feature is considered unsafe. To use pointers in C#, you must declare the method or code block as `unsafe`.
- **Memory Management:** Unlike in languages like C or C++, C# handles memory management through garbage collection, which automatically handles the memory allocation and deallocation for reference types. As a result, pointers are generally not needed for regular C# programming.
- **Interop:** Pointers are often used in **interop scenarios** where C# needs to interact with unmanaged code or C libraries that require direct memory access.

When to Use Pointers in C#

Pointers are useful in specific cases, such as:

- Interacting with **unmanaged code** (e.g., Windows API).
- Optimizing **performance** in scenarios where direct memory manipulation is necessary.
- Working with **low-level system programming** or **hardware interfaces**.

However, in everyday applications, C#'s managed memory model and higher-level constructs like collections, classes, and structs are usually sufficient and much safer.

Please Note



Pointers are a powerful feature of C#, but they should be used carefully due to the potential for unsafe memory access. In most cases, you don't need pointers

for general programming. However, for **performance optimization** or when working with **unmanaged code**, pointers can be helpful.

- ⚠ By using pointers, you gain direct access to memory, which can result in faster execution for certain tasks, but it also bypasses some of the safety features of the .NET runtime.

Type Casting in C#

Here is the updated Type Casting in C# documentation with the requested code blocks removed:

Type Casting in C#

In simple terms, **Type Casting** or **Type Conversion** in C# is the process of converting a value from one data type to another. Type conversion is possible only when the source and target data types are compatible with each other; otherwise, it results in a **compile-time error** saying `cannot implicitly convert one type to another`.

Let us better understand this concept with an example. When we declare an integer variable, we cannot implicitly assign a string value to it. For instance:

```
int a;  
a = "Hello"; // Error CS0029: cannot implicitly convert type string to  
int
```

In this case, the compiler throws an error because a string cannot be automatically converted to an integer.

Types of Type Casting in C#

Type casting in C# is of two types:

1. **Implicit Type Casting** (also called **Automatic Type Conversion**)
2. **Explicit Type Casting**

1. Implicit Type Casting (Automatic Type Conversion)

Implicit Type Casting is automatically done by the C# compiler when converting from a smaller data type to a larger one. This type of conversion is safe and doesn't lose any data. The compiler performs the conversion without requiring the developer's intervention.

In implicit conversion, the conversion is performed **automatically** because the target

data type has enough capacity to store the source data type's value.

Example:

```
int num = 10; // int is 32-bit
double result = num; // Implicit conversion from int to double

Console.WriteLine("Value of result: " + result); // Output: 10.0
```

Explanation:

- In the above example, the `int` value `num` is automatically converted to `double` without any explicit cast.
- The `int` type (32-bit) can be safely converted to `double` (64-bit) because `double` has more capacity, so there is no data loss.

2. Explicit Type Casting (Manual Type Conversion)

Explicit Type Casting occurs when we manually convert one data type to another using casting operators. This is necessary when converting from a larger data type to a smaller one, as data loss or truncation may occur. Since this type of casting could potentially lose data, the compiler does not allow it implicitly and expects the developer to handle the conversion explicitly.

Syntax for Explicit Casting:

```
(targetType) value;
```

Example:

```
double num = 10.75; // double is 64-bit
int result = (int)num; // Explicit conversion from double to int

Console.WriteLine("Value of result: " + result); // Output: 10
```

Explanation:

- The double value num is explicitly converted to int using the cast (int). This operation discards the decimal portion, resulting in 10.

Difference Between Implicit and Explicit Type Casting

Aspect	Implicit Type Casting	Explicit Type Casting
When it occurs	Automatically performed by the compiler.	Performed manually by the developer.
Conversion direction	Smaller data types to larger data types.	Larger data types to smaller data types.
Risk of data loss	No risk of data loss or truncation.	May result in data loss or truncation (e.g., from double to int).
Syntax	No special syntax required.	Requires a cast operator (e.g., (int), (float), etc.).
Example	int num = 10; double result = num;	double num = 10.75; int result = (int)num;

Type Casting with Different Data Types

1. Implicit Conversion Example (int to long)

```
int a = 10;
long b = a; // Implicit conversion from int to long

Console.WriteLine("Value of b: " + b); // Output: 10
```

Explanation:

- Converting from int to long is an implicit conversion because long (64-bit) has more capacity than int (32-bit).

2. Explicit Conversion Example (double to int)

```
double a = 9.99;  
int b = (int)a; // Explicit conversion from double to int  
  
Console.WriteLine("Value of b: " + b); // Output: 9
```

Explanation:

- The double value `a` is explicitly converted to `int` using the cast `(int)`. This operation discards the decimal portion, resulting in `9`.

3. Using Convert Class for Explicit Type Casting

Another way to explicitly convert types is by using the `Convert` class, which handles many conversions.

```
string str = "123";  
int num = Convert.ToInt32(str); // Convert string to int  
  
Console.WriteLine("Converted number: " + num); // Output: 123
```

Explanation:

- The `Convert.ToInt32()` method is used to convert a string value to an integer. If the string is not a valid number, this method will throw an exception.

Common Errors in Type Casting

1. Invalid Type Casting:

Trying to convert incompatible types (e.g., `string` to `int` directly).

```
string text = "Hello";  
int number = (int)text; // Error: Cannot convert string to int  
directly.
```

2. **Data Loss:** Converting a larger data type to a smaller one without explicitly handling the potential loss of data.

```
double largeValue = 9999999.99;  
int smallValue = (int)largeValue; // Data loss due to truncation.  
Console.WriteLine(smallValue); // Output: 9999999
```

Table of Compatible Data Types for Type Casting in C#

Source Type	Target Type	Conversion Type	Description
int	long	Implicit	int can be implicitly converted to long
int	double	Implicit	int can be implicitly converted to double
int	float	Implicit	int can be implicitly converted to float
long	float	Explicit	long needs explicit casting to float
long	double	Implicit	long can be implicitly converted to double
float	double	Implicit	float can be implicitly converted to double
double	float	Explicit	double needs explicit casting to float
double	long	Explicit	double needs explicit casting to long
char	int	Implicit	char can be implicitly converted to int
char	long	Implicit	char can be implicitly converted to long
char	double	Implicit	char can be implicitly converted to double
byte	short	Implicit	byte can be implicitly converted to short
byte	int	Implicit	byte can be implicitly converted to int
byte	long	Implicit	byte can be implicitly converted to long
byte	float	Implicit	byte can be implicitly converted to float

byte	double	Implicit	<code>byte</code> can be implicitly converted to <code>double</code>
short	int	Implicit	<code>short</code> can be implicitly converted to <code>int</code>
short	long	Implicit	<code>short</code> can be implicitly converted to <code>long</code>
short	float	Implicit	<code>short</code> can be implicitly converted to <code>float</code>
short	double	Implicit	<code>short</code> can be implicitly converted to <code>double</code>
decimal	double	Explicit	<code>decimal</code> needs explicit casting to <code>double</code>
decimal	float	Explicit	<code>decimal</code> needs explicit casting to <code>float</code>
bool	(other types)	Not allowed	<code>bool</code> cannot be implicitly or explicitly cast to other types

Conclusion

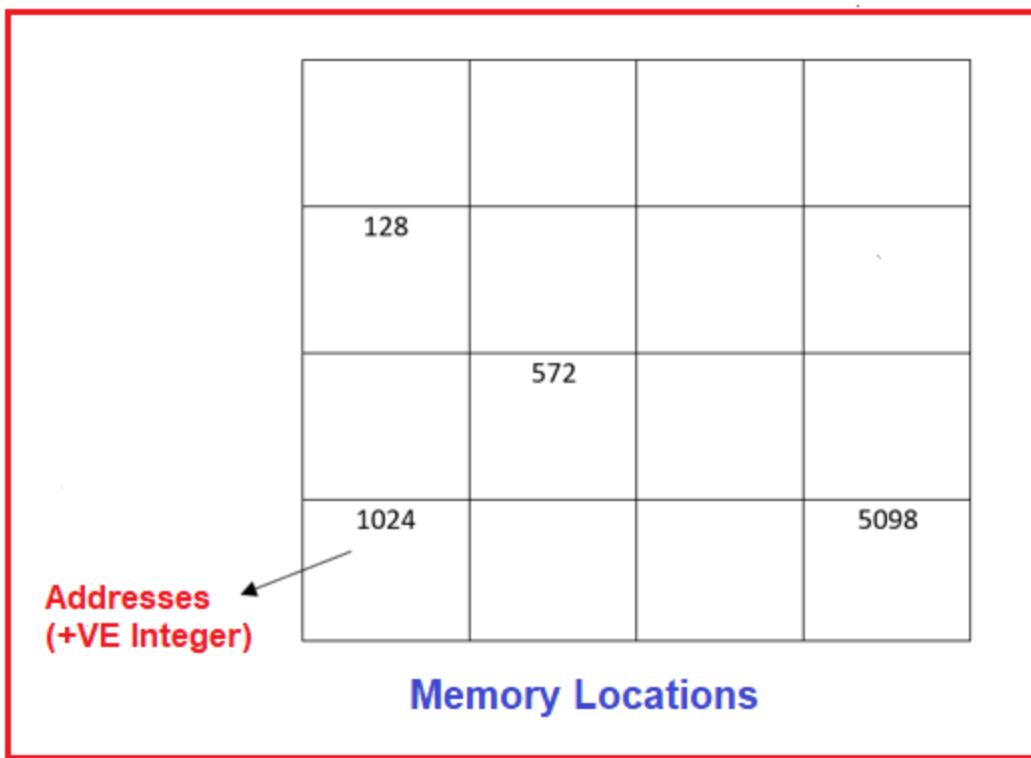
- **Implicit Type Casting:** Done automatically by the compiler when converting from a smaller data type to a larger one. It is safe and causes no data loss.
- **Explicit Type Casting:** Done manually by the developer when converting from a larger data type to a smaller one. This type of casting can lead to data loss or truncation if not handled properly.

Understanding the difference between implicit and explicit type casting is crucial in C# to avoid errors and ensure proper data manipulation.

Variables in C#

Whenever we are processing the data or information, the data or information must be at some location. And we call that location a Memory Location. Every computer has memory locations, and every memory location is identified by an address. Just consider in a movie hall, the seating arrangement as memory locations.

So, every memory location in the computer is identified by an address. For a better understanding, please have a look at the below image. As you can see in the below image, 128, 572, 1024, 5098, etc. are one-one memory addresses. We can treat all the addresses are positive integer values.

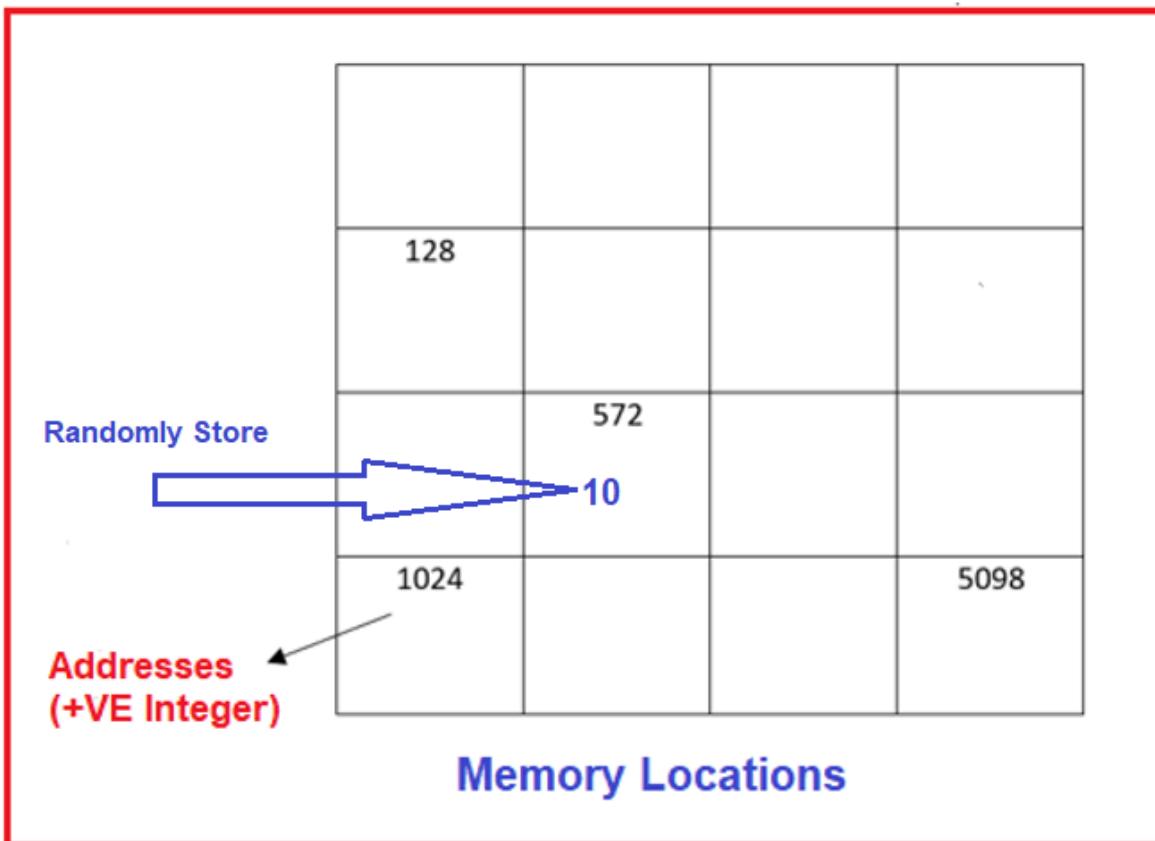


image_34.png

What is the relation between variable and memory locations?

Suppose I want to store a value of 10 in the computer memory locations. Just consider a classroom, there is no restriction on the students where they can sit. That means the students will go and sit randomly at different locations. In the same way, the value 10 that we want to store in the computer memory locations will also go and be stored randomly.

at a particular memory location. For a better understanding, please have a look at the below image.



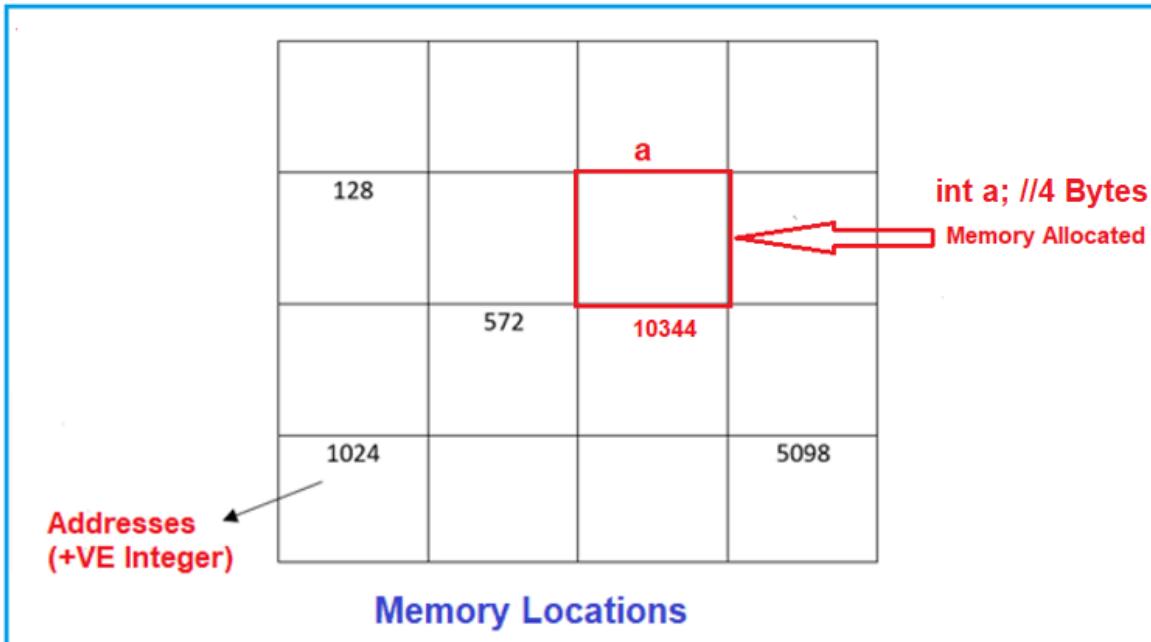
image_35.png

How we can set Identity to Memory Locations?

We can set the identity of the memory location by using variables or you can say identifiers. The following is the syntax to declare a variable by setting the identity of the memory location in the C# language. First, we need to write the data type followed by the identifier.

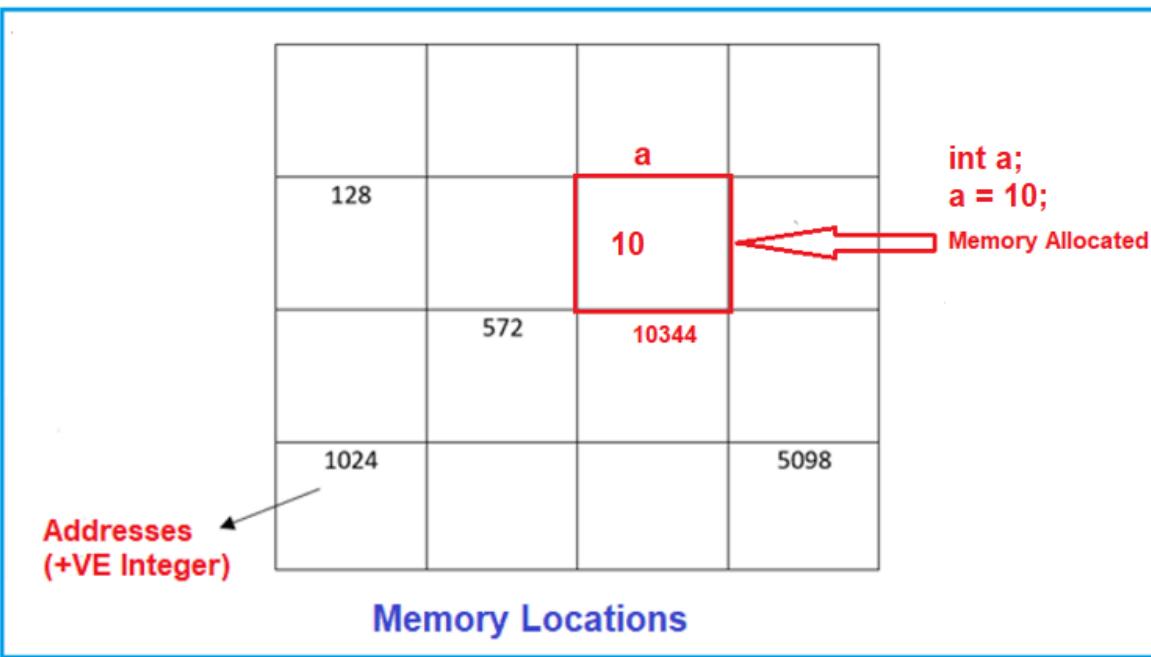
```
## Syntax: data_type Identifier;  
## Example:  
int a;  
/* Here int is the data type and the identifier can be any name and here  
we set it as a.  
So, whenever we declare a variable, it gets memory allocated.
```

To one memory location, the identity is set as shown in the below image.
*/



image_36.png

Here “a” is a named memory location to location 10344. Later we can store an element in that memory location that is identified by the identifier “a” as follows.



image_37.png

What is a Variable in C# Language?



A name that is given for any computer memory location is called a variable. The purpose of the variable is to provide some name to a memory location where we store some data. The user will access the data by the variable name and the compiler will access the data by the memory address. So, the variable is a named location in the computer memory where a program can store the data.

Rules for variable declaration in C#:

1. A variable name must begin with a letter or underscore.
2. Variables in C# are case sensitive
3. They can be constructed with digits and letters.
4. No special symbols are allowed other than underscores.
5. sum, Height, _value, and abc123, etc. are some examples of the variable name

How to declare a variable in C#?

```
## Syntax:  
data_type variable_name;  
  
## Example:  
int age;
```

Here, data_type is the type of data to be stored in the variable, and variable_name is the name given to that variable.

How to initialize a Variable in C#?

The Syntax for initializing a variable in C# is as follows:

```
## Syntax:  
data_type variable_name = value;  
  
## Example:  
int age = 20;
```

Here, `data_type` is the type of data to be stored in the variable, `variable_name` is the name given to the variable and `value` is the initial value stored in the variable.

Types of Variables in a Class in C#:

let us understand the different kinds of variables a class can have and their behavior.

Basically, there are four types of variables that we can declare inside a class in C#. They are as follows:

1. Non-Static/Instance Variable
2. Static Variable
3. Constant Variable
4. Readonly Variable

Static and Non-Static Variables in C#

⚠ If we declare a variable explicitly by using the `static` modifier, we call it a static variable, and the rest of all are non-static variables.

⚠ Again, if we declare a variable inside a static block, then also that variable is a static variable.

⚠ And if we declare a variable inside a non-static block, then that becomes a non-static variable.

For a better understanding, please have a look at the following example. In the below example, we have declared three variables.

```
using System;
namespace TypesOfVariables
{
    internal class Program
    {
        static int x; //Static Variable
        int y; //Non-Static or Instance Variable
        static void Main(string[] args)
        {
            int z; //Static Variable
        }
    }
}
```

⚠ Now, let us try to print the value of x and y inside the Main method. Let us initialize the x value to 100 and the y value to 200. Here, you can print the value of x directly inside the Main method. But you cannot print the value of y directly inside the Main method.

```
using System;
namespace TypesOfVariables
{
    internal class Program
    {
        static int x = 100; //Static Variable
        int y = 200; //Non-Static or Instance Variable
        static void Main(string[] args)
        {
            Console.WriteLine($"x value: {x}");
            Console.Read();
        }
    }
}
```

```
    }  
}
```

Output: x value: 100

Now, let us try to print the y value also directly. If we try to print the y value directly, then we will get a compile-time error saying an object reference is required for the non-static field, method, or property 'Program.y'.

```
using System;  
namespace TypesOfVariables  
{  
    internal class Program  
    {  
        static int x = 100; //Static Variable  
        int y = 200; //Non-Static or Instance Variable  
        static void Main(string[] args)  
        {  
            Console.WriteLine($"x value: {x}");  
            Console.WriteLine($"x value: {y}");  
            Console.Read();  
        }  
    }  
}
```

When you try to run the above code, you will get the following Compile Time Error.

Code	Description
CS0120	An object reference is required for the non-static field, method, or property 'Program.y'

image_38.png

A This is because the memory for the variable y is going to be created only when we create an instance of the class Program and for each instance.

⚠ But x does not require an instance of the class. The reason is a static variable is initialized immediately once the execution of the class starts.

⚠ So, until and unless we created the instance of the Program class, the memory will not be allocated for the variable y and as long as the memory is not allocated for the variable y, we cannot access it. So, once we create the instance of the Program class, the memory for variable y will be allocated, and then only we can access the variable y.

In the below example, we are creating an instance of the Program class, and using that instance we are accessing the y variable. But we are accessing directly the x variable.

```
using System;
namespace TypesOfVariables
{
    internal class Program
    {
        static int x = 100; //Static Variable
        int y = 200; //Non-Static or Instance Variable
        static void Main(string[] args)
        {
            Console.WriteLine($"x value: {x}");
            Program obj = new Program();
            Console.WriteLine($"y value: {obj.y}");
            Console.Read();
        }
    }
}
```

output :

```
x value: 100
y value: 200
```

Difference Between Static and Non-Static Variables in C#

- In the case of an Instance Variable, each object will have its own copy whereas We can only have one copy of a static variable irrespective of how many objects we create.
- In C#, the Changes made to the instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of static variables, changes made in one object will be reflected in other objects as static variables are common to all objects of a class.
- We can access the instance variables through object references whereas the Static Variables can be accessed directly by using the class name in C#.
- In the life cycle of a class, a static variable is initialized only once, whereas instance variables are initialized for 0 times if no instance is created and n times if n number of instances are created.

Instance/Non-Static Variables in C#

- Scope of Instance Variable: Throughout the class except in static methods.
- The lifetime of Instance Variable: Until the object is available in the memory.

Static Variables in C#

- Scope of the Static Variable: Throughout the class.
- The Lifetime of Static Variable: Until the end of the program.

Constant Variables in C#:

In C#, if we declare a variable by using the const keyword, then it is a constant variable and the value of the constant variable can't be modified once after its declaration.

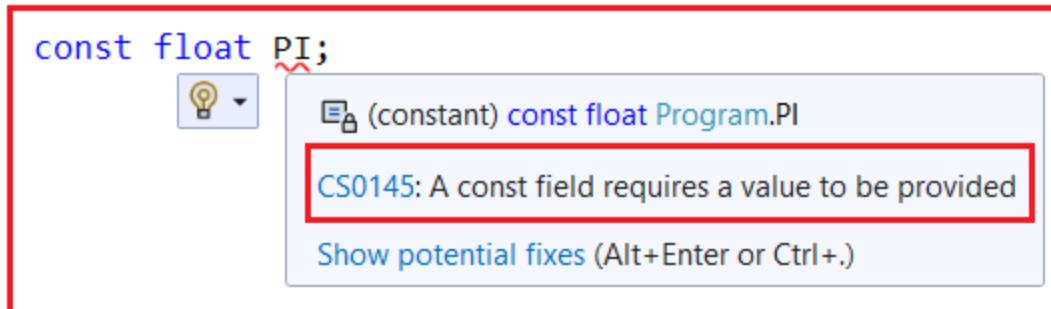


So, it is mandatory to initialize the constant variable at the time of its declaration only.

Suppose, you want to declare a constant PI in your program, then you can declare the constant as follows:

```
const float PI = 3.14f;
```

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



image_39.png

As you can see it saying a const field requires a value to be provided which means while declaring a constant it is mandatory to initialize the constant variable.

```
using System;
namespace TypesOfVariables
{
    internal class Program
    {
        const float PI = 3.14f; //Constant Variable
        static int x = 100; //Static Variable
        //We are going to initialize variable y through constructor
        int y; //Non-Static or Instance Variable

        //Constructor
        public Program(int a)
        {
            //Initializing non-static variable
            y = a;
        }

        static void Main(string[] args)
        {
            //Accessing the static variable without instance
            Console.WriteLine($"x value: {x}");
        }
}
```

```

//Accessing the constant variable without instance
Console.WriteLine($"PI value: {PI}");

Program obj1 = new Program(300);
Program obj2 = new Program(400);
//Accessing Non-Static variable using instance
Console.WriteLine($"obj1 y value: {obj1.y}");
Console.WriteLine($"obj2 y value: {obj2.y}");
Console.Read();
}

}
}

```

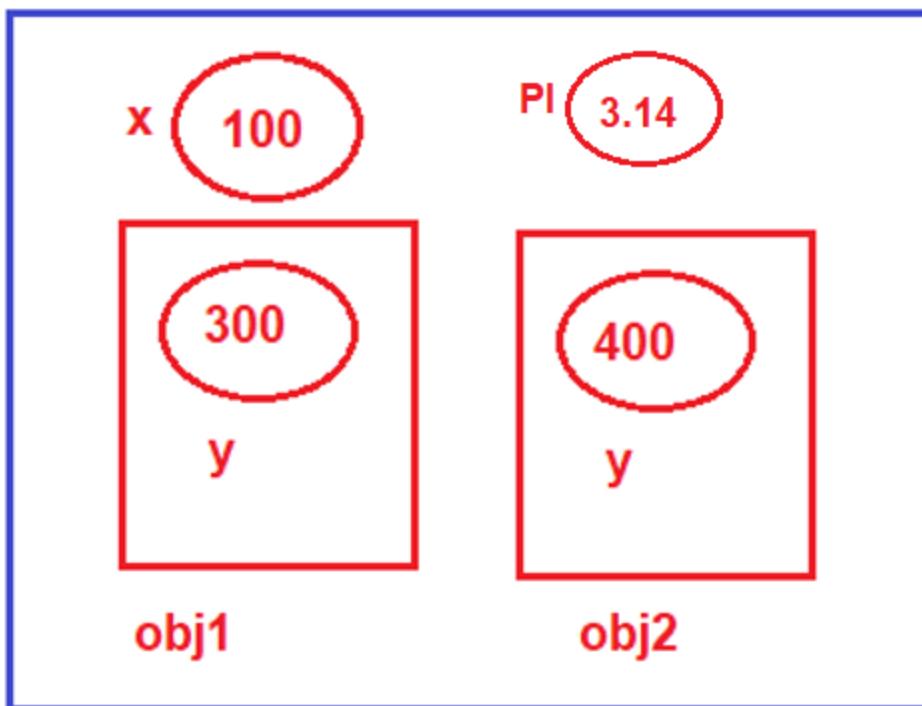
Output:

```

x value: 100
PI value: 3.14
obj1 y value: 300
obj2 y value: 400

```

The following diagram shows the memory representation of the above example.



image_40.png

Now, you might have one question, if both static and constant are behaving in the same way, then what are the differences between them?

Read-only Variable in C#

When we declare a variable by using the `readonly` keyword, then it is known as a read-only variable and these variables can't be modified like constants but after initialization.

- ⚠ That means it is not mandatory to initialize a read-only variable at the time of its declaration, they can also be initialized under the constructor.
- ⚠ That means we can modify the read-only variable value only within a constructor.

Example to Understand Read-Only Variables in C#:

In the below example, the read-only variable `z` is not initialized with any value but when we print the value of the variable, the default value of int i.e. 0 will be displayed.

```
using System;
namespace TypesOfVariables
{
    internal class Program
    {
        const float PI = 3.14f; //Constant Variable
        static int x = 100; //Static Variable
        //We are going to initialize variable y through constructor
        int y; //Non-Static or Instance Variable
        readonly int z; //Readonly Variable

        //Constructor
        public Program(int a)
        {
            //Initializing non-static variable
            y = a;
        }
    }
}
```

```

static void Main(string[] args)
{
    //Accessing the static variable without instance
    Console.WriteLine($"x value: {x}");
    //Accessing the constant variable without instance
    Console.WriteLine($"PI value: {PI}");

    Program obj1 = new Program(300);
    Program obj2 = new Program(400);
    //Accessing Non-Static variable using instance
    Console.WriteLine($"obj1 y value: {obj1.y} and Readonly z
value: {obj1.z}");
    Console.WriteLine($"obj2 y value: {obj2.y} and Readonly z
value: {obj2.z}");
    Console.Read();
}
}

```

Output:

```

x value: 100
PI value: 3.14
obj1 y value: 300 and Readonly z value: 0
obj2 y value: 400 and Readonly z value: 0

```

In the below example, we are initializing the readonly variable through the class constructor. Now, the constructor takes two parameters. The first parameter will initialize the non-static variable and the second parameter will initialize the readonly variable. So, while creating the instance, we need to pass two integer values to the constructor function.

```

using System;
namespace TypesOfVariables
{
    internal class Program

```

```

{
    const float PI = 3.14f; //Constant Variable
    static int x = 100; //Static Variable
    //We are going to initialize variable y through constructor
    int y; //Non-Static or Instance Variable
    readonly int z; // Readonly Variable

    //Constructor
    public Program(int a, int b)
    {
        //Initializing non-static variable
        y = a;
        //Initializing Readonly variable
        z = b;
    }

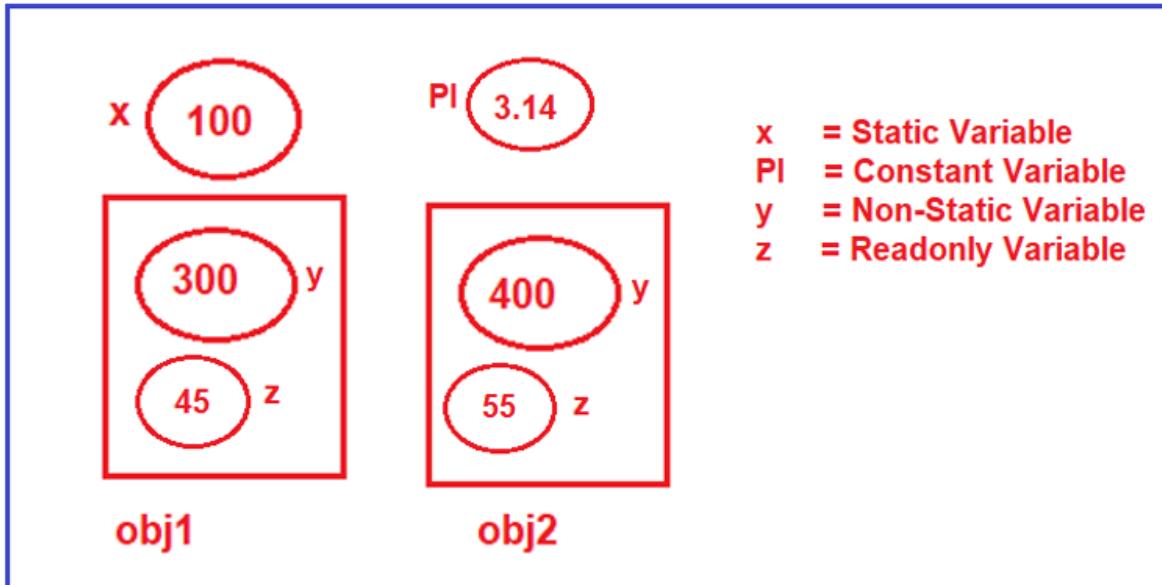
    static void Main(string[] args)
    {
        //Accessing the static variable without instance
        Console.WriteLine($"x value: {x}");
        //Accessing the constant variable without instance
        Console.WriteLine($"PI value: {PI}");

        Program obj1 = new Program(300, 45);
        Program obj2 = new Program(400, 55);
        //Accessing Non-Static variable using instance
        Console.WriteLine($"obj1 y value: {obj1.y} and Readonly z
value: {obj1.z}");
        Console.WriteLine($"obj2 y value: {obj2.y} and Readonly z
value: {obj2.z}");
        Console.Read();
    }
}

```

Output:

```
x value: 100  
PI value: 3.14  
obj1 y value: 300 and Readonly z value: 45  
obj2 y value: 400 and Readonly z value: 55
```



image_41.png

Difference Between Non-Static and Readonly in C#:

- ⚠ The only difference between a non-static and readonly variable is that after initialization, you can modify the non-static variable value but you cannot modify the readonly variable value.

In the below example, after creating the first instance we are trying to modify the non-static y and readonly z variable value.

```
using System;  
namespace TypesOfVariables  
{  
    internal class Program  
    {  
        const float PI = 3.14f; //Constant Variable  
        static int x = 100; //Static Variable
```

```

//We are going to initialize variable y through constructor
int y; //Non-Static or Instance Variable
readonly int z; // Readonly Variable

//Constructor
public Program(int a, int b)
{
    //Initializing non-static variable
    y = a;
    //Initializing Readonly variable
    z = b;
}

static void Main(string[] args)
{
    //Accessing the static variable without instance
    Console.WriteLine($"x value: {x}");
    //Accessing the constant variable without instance
    Console.WriteLine($"PI value: {PI}");

    Program obj1 = new Program(300, 45);
    //Accessing Non-Static variable using instance
    Console.WriteLine($"obj1 y value: {obj1.y} and Readonly z
value: {obj1.z}");

    obj1.y = 500; //Modifying Non-Static Variable
    obj1.z = 400; //Trying to Modify Readonly Variable, Getting
Error

    Console.Read();
}
}

```

When you try to execute the above code, you will get the following Compilation Error.

	Code	Description
✖	CS0191	A readonly field cannot be assigned to (except in a constructor or init-only setter of the type in which the field is defined or a variable initializer)

image_42.png

- ⚠ A readonly field cannot be assigned to (except in a constructor or init-only setter of the type in which the field is defined or a variable initializer).

Operators in C#

Operators in C# are symbols that are used to perform operations on operands. For example, consider the expression $2 + 3 = 5$, here **2** and **3** are **operands**, and **+** and **=** are called **operators**. So, the Operators in C# are used to manipulate the variables and values in a program.

Types of Operators in C#:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Unary Operators or
7. Ternary Operator or Conditional Operator

In C#, the Operators can also be categorized based on the Number of Operands:

Operator	Type
+, -, *, /, %	Arithmetic Operators
<, <=, >, >=, ==, !=	Relational Operators
&&, , !	Logical Operators
&, , <<, >>, ~, ^	Bitwise Operators
=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator → ++, --	Unary Operators
Ternary Operator → ?:	Ternary Operator or Conditional Operator

image_43.png

- 1. Unary Operator:** The Operator that requires one operand (variable or value) to perform the operation is called Unary Operator.
- 2. Binary Operator:** Then Operator that requires two operands (variables or values) to perform the operation is called Binary Operator.
- 3. Ternary Operator:** The Operator that requires three operands (variables or values) to perform the operation is called Ternary Operator. The Ternary Operator is also called Conditional Operator.

Arithmetic Operators in C#

The Arithmetic Operators in C# are used to perform arithmetic/mathematical operations like addition, subtraction, multiplication, division, etc. on operands. The following Operators are falling into this category.

Addition Operator (+):

```
int a = 10;
int b = 5;
```

```
int c = a + b; //15, Here, it will add the a and b operand values i.e.  
10 + 5
```

Subtraction Operator (-):

```
int a = 10;  
int b = 5;  
int c = a - b; //5, Here, it will subtract b from a i.e. 10 - 5
```

Multiplication Operator (*):

```
int a = 10;  
int b = 5;  
int c = a * b; //50, Here, it will multiply a with b i.e. 10 * 5
```

Division Operator (/):

```
int a = 10;  
int b = 5;  
int c = a / b; //2, Here, it will divide 10 / 5
```

Modulus Operator (%):

The % (Modulos) operator returns the remainder when the first operand is divided by the second. As this operator works with two operands, so, this % (Modulos) operator belongs to the category of the binary operator

```
int a = 10;  
int b = 5;  
int c=a % b; //0, Here, it will divide 10 / 5 and it will return the  
remainder which is 0 in this case
```

Example to Understand Arithmetic Operators in C#:

```
using System;  
namespace OperatorsDemo  
{
```

```

class Program
{
    static void Main(string[] args)
    {
        int Result;
        int Num1 = 20, Num2 = 10;

        // Addition Operation
        Result = (Num1 + Num2);
        Console.WriteLine($"Addition Operator: {Result}");

        // Subtraction Operation
        Result = (Num1 - Num2);
        Console.WriteLine($"Subtraction Operator: {Result}");

        // Multiplication Operation
        Result = (Num1 * Num2);
        Console.WriteLine($"Multiplication Operator: {Result}");

        // Division Operation
        Result = (Num1 / Num2);
        Console.WriteLine($"Division Operator: {Result}");

        // Modulo Operation
        Result = (Num1 % Num2);
        Console.WriteLine($"Modulo Operator: {Result}");

        Console.ReadKey();
    }
}

```

Assignment Operators in C#:

The Assignment Operators in C# are used to assign a value to a variable. The left-hand side operand of the assignment operator is a variable and the right-hand side operand of

the assignment operator can be a value or an expression that must return some value and that value is going to assign to the left-hand side variable.

Simple Assignment (=):

```
int a=10;  
int b=20;
```

Add Assignment (+=):

This operator is the combination of + and = operators. It is used to add the left-hand side operand value with the right-hand side operand value and then assign the result to the left-hand side variable.

```
int a=5;  
int b=6;  
a += b; //a=a+b; That means (a += b) can be written as (a = a + b)
```

Subtract Assignment (-=):

This operator is the combination of – and = operators. It is used to subtract the right-hand side operand value from the left-hand side operand value and then assign the result to the left-hand side variable.

```
int a=10;  
int b=5;  
a -= b; //a=a-b; That means (a -= b) can be written as (a = a - b)
```

Multiply Assignment (*=):

This operator is the combination of * and = operators. It is used to multiply the left-hand side operand value with the right-hand side operand value and then assign the result to the left-hand side variable.

```
int a=10;  
int b=5;  
a *= b; //a=a*b; That means (a *= b) can be written as (a = a * b)
```

Division Assignment (/=):

This operator is the combination of / and = operators. It is used to divide the left-hand side operand value with the right-hand side operand value and then assign the result to the left-hand side variable.

```
int a=10;
int b=5;
a /= b; //a=a/b; That means (a /= b) can be written as (a = a / b)
```

Modulus Assignment (%=):

This operator is the combination of % and = operators. It is used to divide the left-hand side operand value with the right-hand side operand value and then assigns the remainder of this division to the left-hand side variable.

```
int a=10;
int b=5;
a %= b; //a=a%b; That means (a %= b) can be written as (a = a % b)
```

Example to Understand Assignment Operators in C#:

```
using System;
namespace OperatorsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Initialize variable x using Simple Assignment Operator
            "="

            int x = 15;

            x += 10; //It means x = x + 10 i.e. 15 + 10 = 25
            Console.WriteLine($"Add Assignment Operator: {x}");

            // initialize variable x again
            x = 20;
            x -= 5; //It means x = x - 5 i.e. 20 - 5 = 15
```

```

        Console.WriteLine($"Subtract Assignment Operator: {x}");

        // initialize variable x again
        x = 15;
        x *= 5; //It means x = x * 5 i.e. 15 * 5 = 75
        Console.WriteLine($"Multiply Assignment Operator: {x}");

        // initialize variable x again
        x = 25;
        x /= 5; //It means x = x / 5 i.e. 25 / 5 = 5
        Console.WriteLine($"Division Assignment Operator: {x}");

        // initialize variable x again
        x = 25;
        x %= 5; //It means x = x % 5 i.e. 25 % 5 = 0
        Console.WriteLine($"Modulo Assignment Operator: {x}");

        Console.ReadKey();
    }
}
}

```

Relational Operators in C#:

The Relational Operators in C# are also known as Comparison Operators. It determines the relationship between two operands and returns the Boolean results, i.e. true or false after the comparison. The Different Types of Relational Operators supported by C# are as follows.

Equal to (==):

This Operator is used to return true if the left-hand side operand value is equal to the right-hand side operand value. For example, $5 == 3$ is evaluated to be false. So, this Equal to (==) operator will check whether the two given operand values are equal or not. If equal returns true else returns false.

Not Equal to (!=):

This Operator is used to return true if the left-hand side operand value is not equal to the right-hand side operand value. For example, $5 != 3$ is evaluated to be true. So, this Not

Equal to (!=) operator will check whether the two given operand values are equal or not. If equal returns false else returns true

Less than (<):

This Operator is used to return true if the left-hand side operand value is less than the right-hand side operand value. For example, $5 < 3$ is evaluated to be false. So, this Less than (<) operator will check whether the first operand value is less than the second operand value or not.

Less than or equal to (<=):

This Operator is used to return true if the left-hand side operand value is less than or equal to the right-hand side operand value. For example, $5 \leq 5$ is evaluated to be true. So, this Less than or equal to (<=) operator will check whether the first operand value is less than or equal to the second operand value. If so returns true else returns false.

Greater than (>):

This Operator is used to return true if the left-hand side operand value is greater than the right-hand side operand value. For example, $5 > 3$ is evaluated to be true. So, this Greater than (>) operator will check whether the first operand value is greater than the second operand value. If so, returns true else return false.

Greater than or Equal to (>=):

This Operator is used to return true if the left-hand side operand value is greater than or equal to the right-hand side operand value. For example, $5 \geq 5$ is evaluated to be true. So, this Greater than or Equal to (>=) operator will check whether the first operand value is greater than or equal to the second operand value. If so, returns true else returns false.

Example to Understand Relational Operators in C#:

```
using System;
namespace OperatorsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
```

```

        bool Result;
        int Num1 = 5, Num2 = 10;

        // Equal to Operator
        Result = (Num1 == Num2);
        Console.WriteLine("Equal (=) to Operator: " + Result);

        // Greater than Operator
        Result = (Num1 > Num2);
        Console.WriteLine("Greater (<) than Operator: " + Result);

        // Less than Operator
        Result = (Num1 < Num2);
        Console.WriteLine("Less than (>) Operator: " + Result);

        // Greater than Equal to Operator
        Result = (Num1 >= Num2);
        Console.WriteLine("Greater than or Equal to (>=) Operator: "
+ Result);

        // Less than Equal to Operator
        Result = (Num1 <= Num2);
        Console.WriteLine("Lesser than or Equal to (<=) Operator: "
+ Result);

        // Not Equal To Operator
        Result = (Num1 != Num2);
        Console.WriteLine("Not Equal to (!=) Operator: " + Result);

        Console.ReadKey();
    }
}

```

Output:

```
Equal (=) to Operator: False  
Greater (<) than Operator: False  
Less than (>) Operator: True  
Greater than or Equal to (>=) Operator: False  
Lesser than or Equal to (<=) Operator: True  
Not Equal to (!=) Operator: True
```

Logical Operators in C#:

The Logical Operators are mainly used in conditional statements and loops for evaluating a condition. These operators are going to work with boolean expressions. The different types of Logical Operators supported in C# are as follows:

Logical OR (||):

This operator is used to return true if either of the Boolean expressions is true. For example, false || true is evaluated to be true. That means the Logical OR (||) operator returns true when one (or both) of the conditions in the expression is satisfied. Otherwise, it will return false. For example, a || b returns true if either a or b is true. Also, it returns true when both a and b are true.

Logical AND (&&):

This operator is used to return true if all the Boolean Expressions are true. For example, false && true is evaluated to be false. That means the Logical AND (&&) operator returns true when both the conditions in the expression are satisfied. Otherwise, it will return false. For example, a && b return true only when both a and b are true.

Logical NOT (!):

This operator is used to return true if the condition in the expression is not satisfied. Otherwise, it will return false. For example, !a returns true if a is false.

Example to Understand Logical Operators in C#:

```
using System;  
namespace OperatorsDemo  
{  
    class Program
```

```

{
    static void Main(string[] args)
    {
        bool x = true, y = false, z;

        //Logical AND operator
        z = x && y;
        Console.WriteLine("Logical AND Operator (&&) : " + z);

        //Logical OR operator
        z = x || y;
        Console.WriteLine("Logical OR Operator (||) : " + z);

        //Logical NOT operator
        z = !x;
        Console.WriteLine("Logical NOT Operator (!) : " + z);

        Console.ReadKey();
    }
}

```

Output:

```

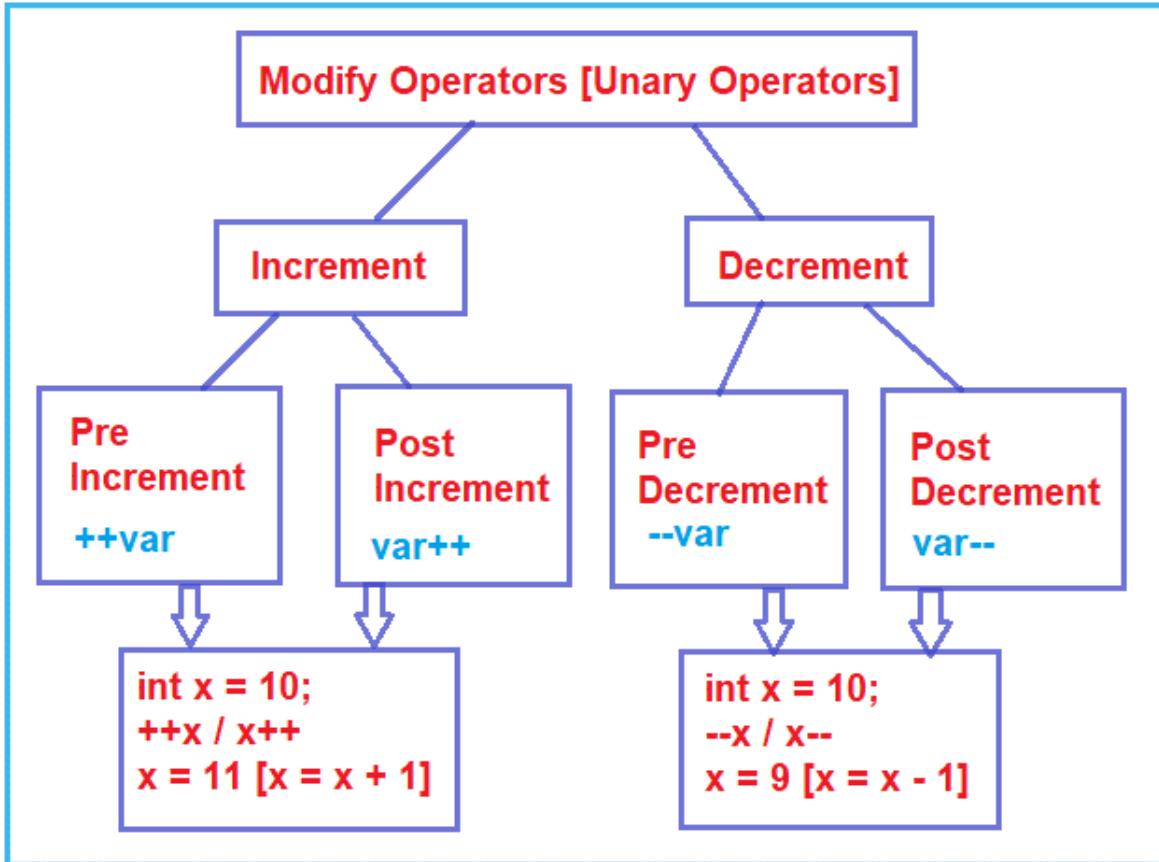
Logical AND Operator (&&) : False
Logical OR Operator (||) : True
Logical NOT Operator (!) : False

```

Unary/Modify Operators in C#:

The Unary Operators in C# need only one operand. They are used to increment or decrement a value. There are two types of Unary Operators. They are as follows:

- **Increment operators (++):** Example: (++x, x++)
- **Decrement operators (–):** Example: (–x, x–)



image_44.png



Increment Operator means to increment the value of the variable by 1 and Decrement Operator means to decrement the value of the variable by 1.

Increment Operator (++) in C# Language:

The Increment Operator (++) is a unary operator. It operates on a single operand only. Again, it is classified into two types:

1. Post-Increment Operator
2. Pre-Increment Operator

Post Increment Operators:

The Post Increment Operators are the operators that are used as a suffix to its variable. It is placed after the variable. For example, a++ will also increase the value of the variable a

by 1.

Syntax: Variable`++`; Example: `x++;`

Pre-Increment Operators:

The Pre-Increment Operators are the operators which are used as a prefix to its variable. It is placed before the variable. For example, `++a` will increase the value of the variable `a` by 1.

Syntax: `++Variable`; Example: `++x;`

Decrement Operators in C# Language:

The Decrement Operator (`-`) is a unary operator. It takes one value at a time. It is again classified into two types. They are as follows:

- Post Decrement Operator
- Pre-Decrement Operator

Post Decrement Operators:

The Post Decrement Operators are the operators that are used as a suffix to its variable. It is placed after the variable. For example, `a-` will also decrease the value of the variable `a` by 1.

Syntax: Variable`--`; Example: `x--;`

Pre-Decrement Operators:

The Pre-Decrement Operators are the operators that are a prefix to its variable. It is placed before the variable. For example, `-a` will decrease the value of the variable `a` by 1.

Syntax: `--Variable`; Example: `--x;`

Example to Understand Increment Operators in C# Language:

```
using System;
namespace OperatorsDemo
```

```

{
    class Program
    {
        static void Main(string[] args)
        {
            // Post-Increment
            int x = 10;
            // Result1 is assigned 10 only,
            // x is not updated yet
            int Result1 = x++;
            //x becomes 11 now
            Console.WriteLine("x is {0} and Result1 is {1}", x,
Result1);

            // Pre-Increment
            int y = 10;
            int Result2 = ++y;
            //y and Result2 have same values = 11
            Console.WriteLine("y is {0} and Result2 is {1}", y,
Result2);

            Console.ReadKey();
        }
    }
}

```

output:

```

x is 11 and Result1 is 10
y is 11 and Result2 is 11

```

Example to understand Decrement Operators in C# Language:

```

using System;
namespace OperatorsDemo
{
    class Program

```

```

{
    static void Main(string[] args)
    {
        // Post-Decrement
        int x = 10;
        // Result1 is assigned 10 only,
        // x is not yet updated
        int Result1 = x--;
        //x becomes 9 now
        Console.WriteLine("x is {0} and Result1 is {1}", x,
Result1);

        // Pre-Decrement
        int y = 10;
        int Result2 = --y;
        //y and Result2 have same values i.e. 9
        Console.WriteLine("y is {0} and Result2 is {1}", y,
Result2);

        Console.ReadKey();
    }
}

```

output:

```

x is 9 and Result1 is 10
y is 9 and Result2 is 9

```

Control Flow Statements in C#

The Control Flow Statements in C# are the statements that Alter the Flow of Program Execution and provide better control to the programmer on the flow of execution. The Control Flow Statements are useful to write better and more complex programs. A program executes from top to bottom except when we use control statements. We can control the order of execution of the program, based on logic and values.

By default, when we write statements in a program, the statements are going to be executed sequentially from top to bottom line by line. For example, in the below program we have written five statements. Now, if you execute the below program, the statements are going to be executed one by one from the top to bottom. That means, first it will execute statement1, then statement2, then statement3, then statement4, and finally statement5.

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Executing Statement1");
            Console.WriteLine("Executing Statement2");
            Console.WriteLine("Executing Statement3");
            Console.WriteLine("Executing Statement4");
            Console.WriteLine("Executing Statement5");
            Console.ReadKey();
        }
    }
}
```

Output:

```
Executing Statement1
Executing Statement2
Executing Statement3
```

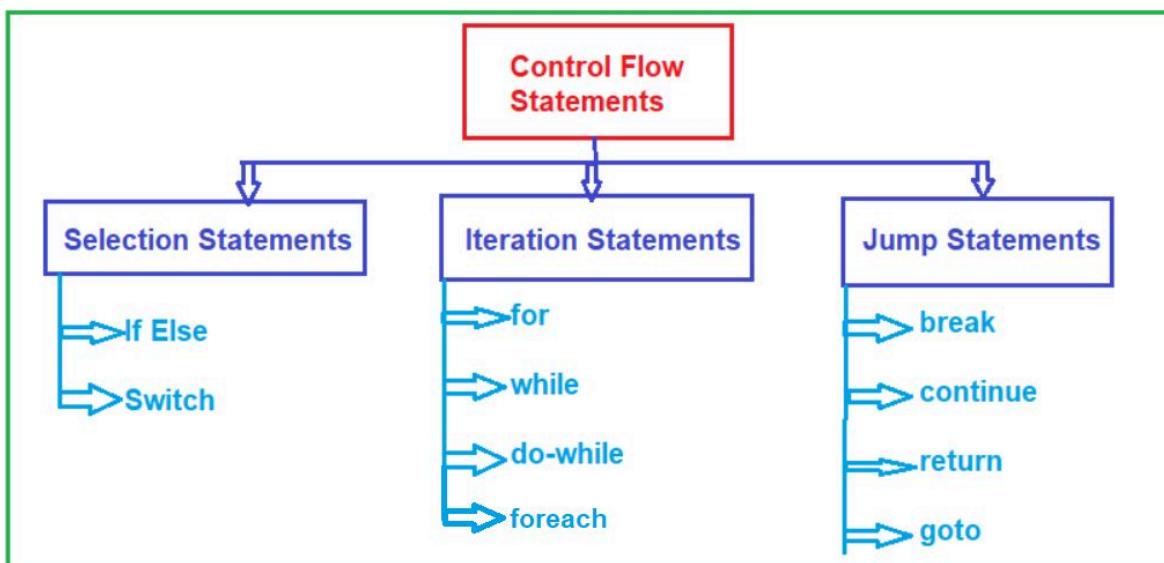
Executing Statement4

Executing Statement5

Types of Control Flow Statements in C#:

In C#, the control flow statements are divided into the following three categories:

1. **Selection Statements or Branching Statements:** (Examples: if-else, switch case, nested if-else, if-else ladder)
2. **Iteration Statements or Looping Statements:** (Examples: while loop, do-while loop, for-loop, and foreach loop)
3. **Jumping Statements:** (Examples: break, continue, return, goto)



image_45.png

Selection Statements

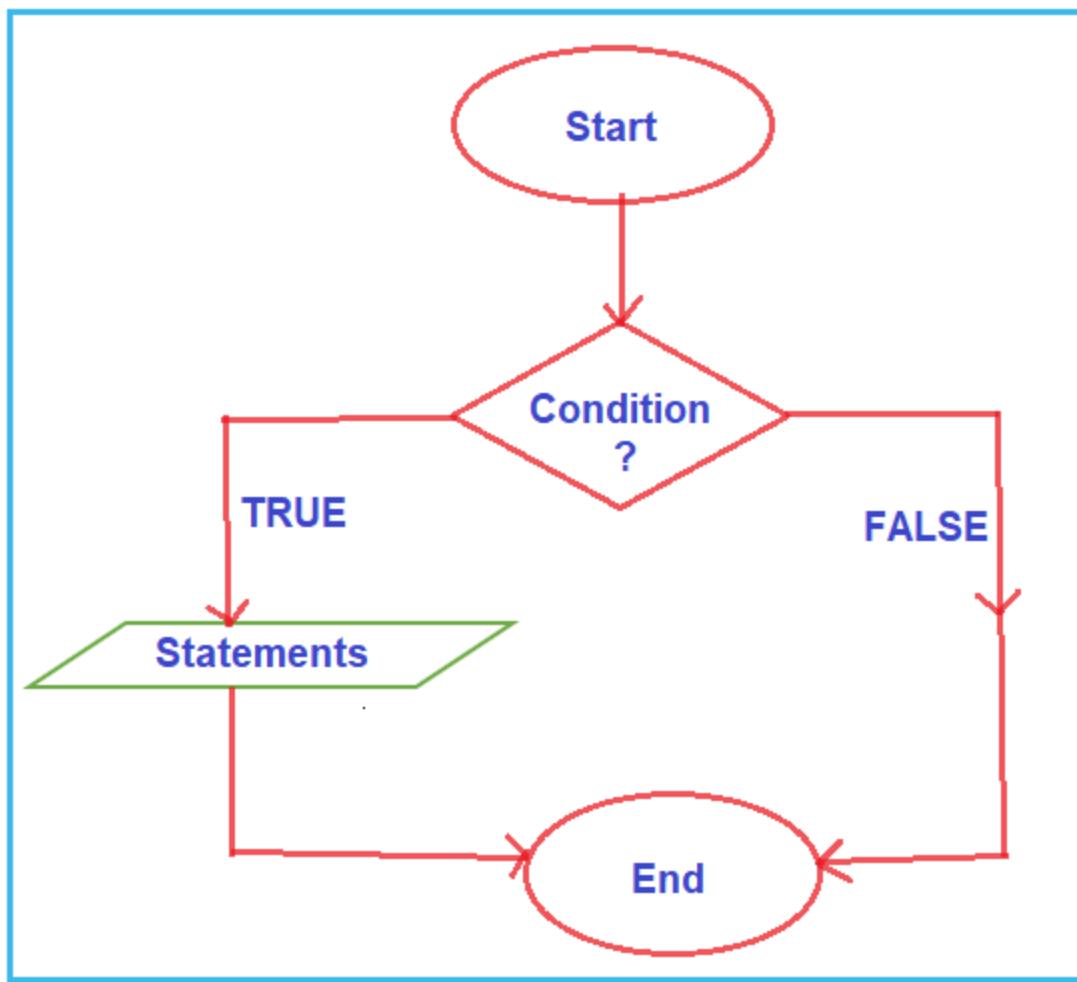
If Statements in C#

It executes a block of statements (one or more instructions) when the condition in the if block is true and when the condition is false, it will skip the execution of the if block. Using else block is optional in C#. Following is the syntax to use the if block in the C# language.

```
if(condition)
{
    Statements;
}
```

image_46.png

Flow Chart of If Block:



image_47.png

Here, the block of statements executes only when the condition is true. And if the condition is false, then it will skip the execution of the statements.

Example to Understand If Block in C#:

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int number;
            Console.WriteLine("Enter a Number: ");
            number = Convert.ToInt32(Console.ReadLine());
            if (number > 10)
            {
                Console.WriteLine($"{number} is greater than 10 ");
                Console.WriteLine("End of if block");
            }
            Console.WriteLine("End of Main Method");
            Console.ReadKey();
        }
    }
}
```

Output: If we take 43 as an input, $43 > 10$ means the condition is true, then if block statement gets executed.

```
Enter a Number:
43
43 is greater than 10
End of if block
End of Main Method
```

If we take 5 as an input, $5 > 10$ means the condition is false, then the if block statements will be skipped

```
Enter a Number:
5
End of Main Method
```

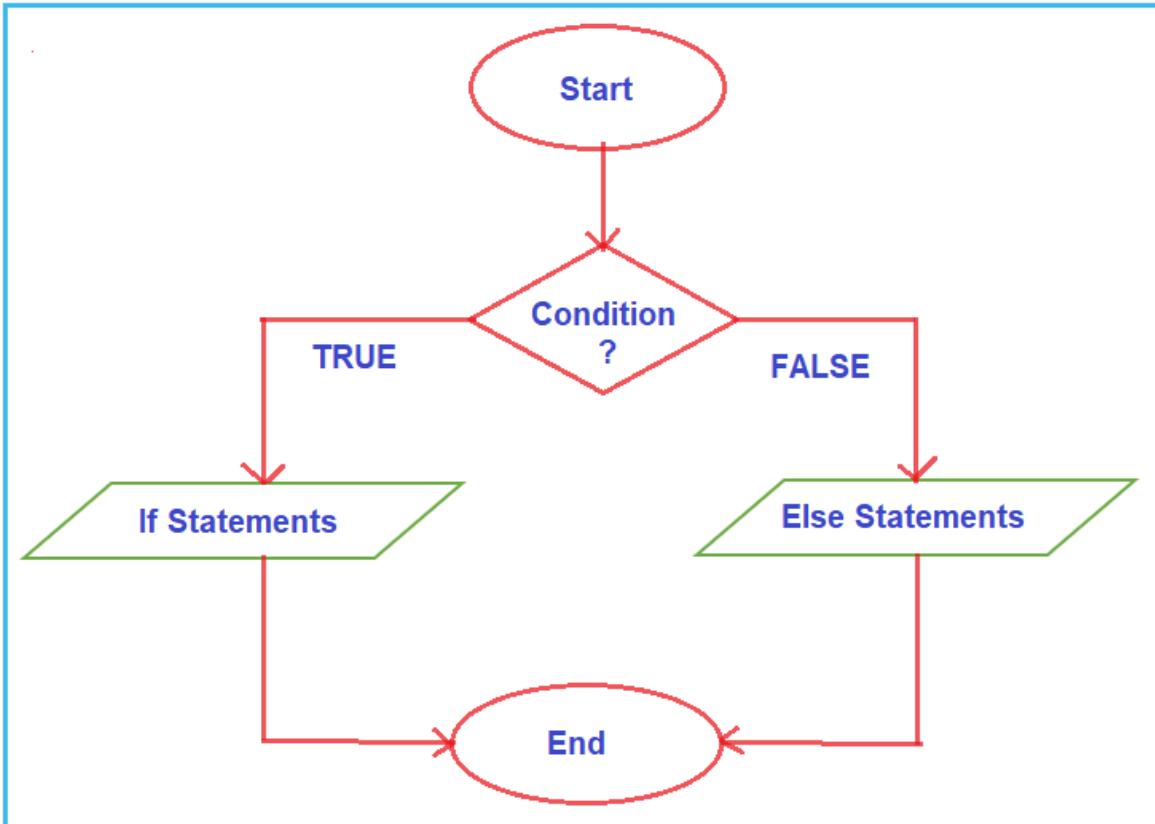
If Else Statements in C# Language:

The If-Else block in C# Language is used to provide some optional information whenever the given condition is FALSE in the if block. That means if the condition is true, then the if block statements will be executed, and if the condition is false, then the else block statement will execute.

```
If (Condition)
{
    if statements;
}
else
{
    else statements;
}
```

image_48.png

Flow Chart of If-Else Block:



image_49.png

⚠ In C# Programming Language, if and else are reserved words. That means you cannot use these two keywords for the naming of any variables, properties, class, methods, etc. The expressions or conditions specified in the if block can be a Relational or Boolean expression or condition that evaluates to TRUE or FALSE. Now let us see some examples to understand the if-else conditional statements

Example to Understand IF-ELSE Statement in C#:

Let us write a Program to Check Whether a Number is Even or Odd using If Else Statements in C# Language. Here we will take the input number from the user and then we will check whether that number is even or odd using the if-else statement in C# Language. The following program exactly does the same.

```

using System;
namespace ControlFlowDemo
{
  
```

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Enter a Number: ");
        int number = Convert.ToInt32(Console.ReadLine());
        if (number % 2 == 0)
        {
            Console.WriteLine($"{number} is an Even Number");
        }
        else
        {
            Console.WriteLine($"{number} is an Odd Number");
        }

        Console.ReadKey();
    }
}

```

If we take 16 as an input, $16 \% 2 == 0$ means the condition is true, then the if block statement gets executed. And the output will be 16 is an Even Number.

```

Enter a Number:
16
16 is an Even Number

```

If we take 13 as an input, $13 \% 2 == 0$ means the condition is false, then the else block statements get executed. And the output will be 13 is an Odd Number.

```

Enter a Number:
13
13 is an Odd Number

```

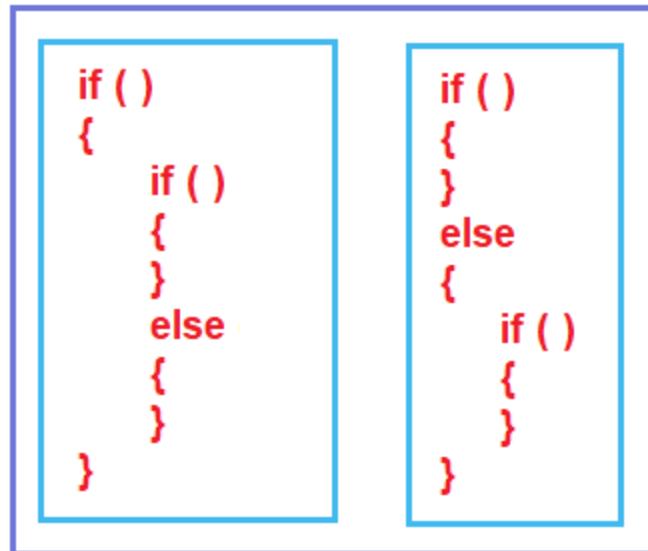
Nested If-Else Statements in C# Language:

When an if-else statement is present inside the body of another if or else then this is called nested if-else. Nested IF-ELSE statements are used when we want to check for a

condition only when the previous dependent condition is true or false

What is the Nested If block?

Nested if block means defining if block inside another if block. We can also define the if block inside the else blocks. Depending on our logic requirements, we can use nested if block either inside the if block or inside the else.



image_50.png

Now, we will take one example and try to understand the flow chart. We are taking the following syntax. Here, we have an if-else block inside the if block, as well as, an if-else block inside the else block.

```

if(Condition/Expression) ← Outer If Block
{
    if(Condition/Expression) ← Inner If Block
    {
        Outer if and Inner If Statements;
    }
    else ← Inner Else Block
    {
        Outer if and inner else statements;
    }
}
else ← Outer Else Block
{
    if(Condition/Expression) ← Inner If Block
    {
        Outer else and inner if statements;
    }
    else ← Inner Else Block
    {
        Outer else and inner else statements;
    }
}

```

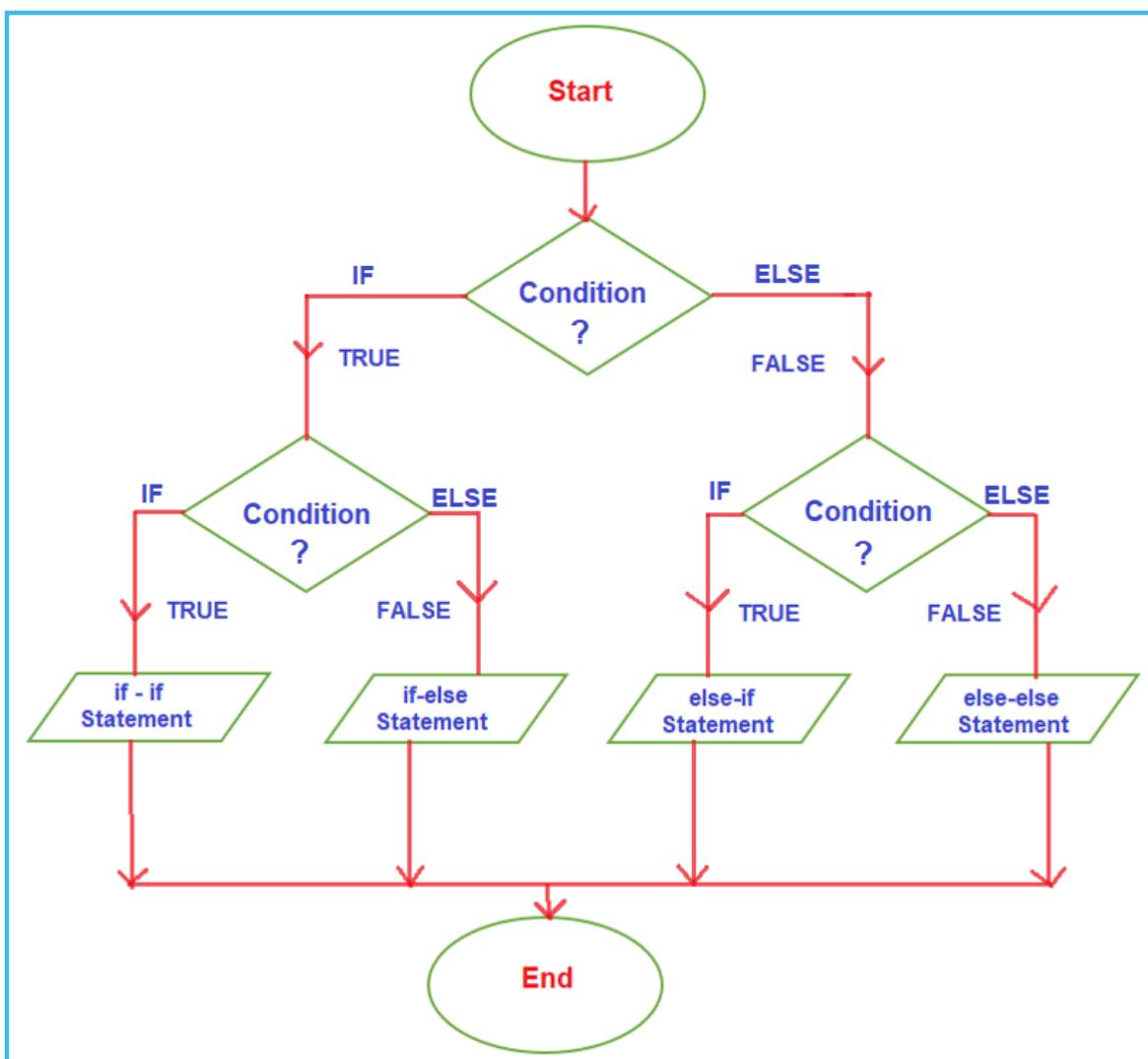
image_51.png

Let us understand the above code.

- Condition1: First, it will check the first if condition i.e. the outer if condition and if it is true, then the outer else block will be terminated. So, the control moves inside the first or the outer if block. Then again it checks the inner if condition and if the inner if condition is true, then the inner else block gets terminated. So, in this case, the outer if and inner if block statements get executed.
- Condition2: Now, if the outer if condition is true, but the inner if condition is false, then the inner if block gets terminated. So, in this case, the outer if and inner else block statements get executed.

- Condition3: Now, if the outer if condition is false, then the outer if block gets terminated and control moves to the outer else block. And inside the outer else block, again it checks the inner if condition, and if the inner if condition is true, then the inner else block gets terminated. So, in this case, the outer else and inner if block statements get executed.
- Condition4: Now, if the outer if condition is false as well as the if condition inside the outer else blocks also failed, then the if block gets terminated. And in this case, the outer else and inner else block statements get executed. This is how statements get executed in Nested if. Now we will see the flow chart of nested if blocks.

Flow Chart of Nested If – Else Block in C# Language:



image_52.png

Example to understand Nested IF-ELSE Statements in C# Language:

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 15, b = 25, c = 10;
            int LargestNumber = 0;

            if (a > b)
            {
                Console.WriteLine($"Outer IF Block");
                if (a > c)
                {
                    Console.WriteLine($"Outer IF - Inner IF Block");
                    LargestNumber = a;
                }
                else
                {
                    Console.WriteLine($"Outer IF - Inner ELSE Block");
                    LargestNumber = c;
                }
            }
            else
            {
                Console.WriteLine($"Outer ELSE Block");
                if (b > c)
                {
                    Console.WriteLine($"Outer ELSE - Inner IF Block");
                    LargestNumber = b;
                }
                else
                {
                    Console.WriteLine($"Outer ELSE - Inner ELSE Block");
                    LargestNumber = c;
                }
            }
        }
    }
}
```

```
        }
    }

    Console.WriteLine($"The Largest Number is:
{LargestNumber}");

    Console.ReadKey();
}
}

}
```

Output:

```
Outer ELSE Block
Outer ELSE - Inner IF Block
The Largest Number is: 25
```

Ladder if-else statements in C# Language:

In Ladder if-else statements one of the statements will be executed depending upon the truth or false of the conditions. If the condition1 is true then Statement 1 will be executed, and if condition2 is true then statement 2 will be executed, and so on. But if all conditions are false, then the last statement i.e. else block statement will be executed. The C# if-else statements are executed from top to bottom. As soon as one of the conditions controlling the if is true, the statement associated with that if block is going to be executed, and the rest of the C# else-if ladder is bypassed. If none of the conditions are true, then the final else statement will be executed.

```
if (condition)
    Statement 1;
else if (condition)
    Statement 2;
.
.
.
.
else
    Statement n;
```

image_53.png

Example to understand Ladder If-Else Statements in C# Language:

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 20;
            if (i == 10)
            {
                Console.WriteLine("i is 10");
            }
            else if (i == 15)
            {
                Console.WriteLine("i is 15");
            }
            else if (i == 20)
            {
                Console.WriteLine("i is 20");
            }
            else
```

```
    {
        Console.WriteLine("i is not present");
    }

    Console.ReadKey();
}
}
```

Switch Statements in C#

The switch is a keyword in the C# language, and by using this switch keyword we can create selection statements with multiple blocks. And the Multiple blocks can be constructed by using the case keyword.

Switch case statements in C# are a substitute for long if else statements that compare a variable or expression to several values.

When do we need to go for a switch statement?

When there are several options and we have to choose only one option from the available options depending on a single condition then we need to go for a switch statement. Depending on the selected option a particular task can be performed.

Syntax of Switch Statements in C# Language:

In C#, the Switch statement is a multiway branch statement. It provides an efficient way to transfer the execution to different parts of a code based on the value of the expression. The switch expression is of integer type such as int, byte, or short, or of an enumeration type, or of character type, or of string type. The expression is checked for different cases and the match case will be executed. The following is the syntax to use switch case statement in C# language.

```
switch(variable)
{
    case 1:
        //execute your code
        break;
    case n:
        //execute your code
        break;
    default:
        //execute your code
        break;
}
```

image_54.png

Example to understand Switch Statement in C# Language:

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 2;
            switch (x)
            {
                case 1:
                    Console.WriteLine("Choice is 1");
                    break;
                case 2:
                    Console.WriteLine("Choice is 2");
                    break;
                case 3:
                    Console.WriteLine("Choice is 3");
                    break;
                default:
                    Console.WriteLine("Choice other than 1, 2 and 3");
                    break;
            }
        }
    }
}
```

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

Output:

```
Choice is 2
```

After the end of each case block, it is necessary to insert a break statement. If we are not inserting the break statement, then we will get a compilation error. But you can combine multiple case blocks with a single break statement if and only if the previous case statement does not have any code block. For a better understanding, please have a look at the below example.

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            string str = "C#";
            switch (str)
            {
                case "C#":
                case "Java":
                case "C":
                    Console.WriteLine("It's a Programming Langauge");
                    break;

                case "MSSQL":
                case "MySQL":
                case "Oracle":
                    Console.WriteLine("It's a Database");
                    break;
            }
        }
    }
}
```

```

        case "MVC":
        case "WEB API":
            Console.WriteLine("It's a Framework");
            break;

        default:
            Console.WriteLine("Invalid Input");
            break;
    }
    Console.ReadKey();
}
}

```

Output:

It's a Programming Language

Is default block Mandatory in a Switch Statement?

No, the default block in the switch statement is not mandatory. If you are putting the default block and if any of the case statement is not fulfilled, then only the default block is going to be executed. For a better understanding, please have a look at the below example where we don't have the default block.

```

using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            string str = "C#2";
            Console.WriteLine("Switch Statement Started");
            switch (str)
            {

```

```

        case "C#":
        case "Java":
        case "C":
            Console.WriteLine("It's a Programming Language");
            break;

        case "MSSQL":
        case "MySQL":
        case "Oracle":
            Console.WriteLine("It's a Database");
            break;

        case "MVC":
        case "WEB API":
            Console.WriteLine("It's a Framework");
            break;
    }
    Console.WriteLine("Switch Statement Ended");
    Console.ReadKey();
}
}

```

Output:

```

Switch Statement Started
Switch Statement Ended

```

Nested Switch Statement in C#:

```

using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {

```

```

//Ask the user to enter a number between 1 and 3
Console.Write("Enter a Number Between 1 and 3:");
int number = Convert.ToInt32(Console.ReadLine());

//outer Switch Statement
switch (number)
{
    case 1:
        Console.WriteLine("You Entered One");
        //Ask the user to enter the character R, B, or G
        Console.Write("Enter Color Code (R/G/B): ");
        char color = Convert.ToChar(Console.ReadLine());

        //Inner Switch Statement
        switch (Char.ToUpper(color))
        {
            case 'R':
                Console.WriteLine("You have Selected Red
Color");
                break;
            case 'G':
                Console.WriteLine("You have Selected Green
Color");
                break;
            case 'B':
                Console.WriteLine("You have Selected Blue
Color");
                break;
            default:
                Console.WriteLine($"You Have Enter Invalid
Color Code: {Char.ToUpper(color)}");
                break;
        }
        break;

    case 2:
        Console.WriteLine("You Entered Two");
        break;
}

```

```

        case 3:
            Console.WriteLine("You Entered Three");
            break;
        default:
            Console.WriteLine("Invalid Number");
            break;
    }

    Console.ReadLine();
}
}
}

```

output:

```

Enter a Number Between 1 and 3:1
You Entered One
Enter Color Code (R/G/B): b
You have Selected Blue Color

```

Iteration Statements

Loops in C#

Loops are also called repeating statements or iterative statements. Loops play an important role in programming. The Looping Statements are also called Iteration Statements. So, we can use the word Looping and Iteration and the meanings are the same.

Example to print numbers from 1 to 10 without using the loop in C# Till now what we learned using those concepts If I write a program to print 1 to 10 it looks something like this.

```

using System;
namespace ControlFlowDemo
{
    class Program

```

```
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("1");  
        Console.WriteLine("2");  
        Console.WriteLine("3");  
        Console.WriteLine("4");  
        Console.WriteLine("5");  
        Console.WriteLine("6");  
        Console.WriteLine("7");  
        Console.WriteLine("8");  
        Console.WriteLine("9");  
        Console.WriteLine("10");  
  
        Console.ReadKey();  
    }  
}
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

⚠ Note: Even though we are able to print the numbers from 1 to 10, the code doesn't look good as the same instruction is written multiple times also what if we want to print from 1 to 1000? Or from 1 to 100000? So, without loops, the code not even looks understandable and efficient.

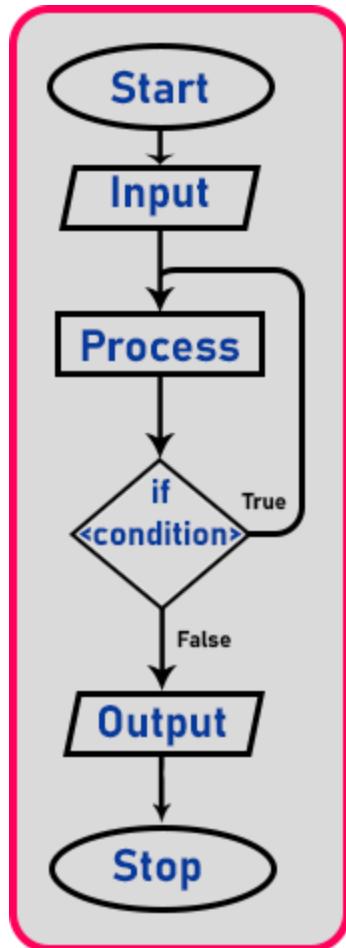
What is looping?

Looping in programming languages is a feature that facilitates the execution of a set of instructions repeatedly while some condition evaluates to true.

The process of repeatedly executing a statement or group of statements until the condition is satisfied is called looping. In this case, when the condition becomes false the execution of the loops terminates. The way it repeats the execution of the statements or instructions will form a circle that's why iteration statements are called loops.

Types of Loops in C#

1. For loop
2. For Each Loop
3. While loop
4. Do while loop



image_55.png

As the input is processed then it checks for the condition, if the condition is true then again it goes back and processing will do and then again check for the condition, if the condition is true then again goes back, and so on.

This will be repeated. So, this processing part will go on repeating as long as that condition is true and once the conditions become false it will come out from here and print the output.

While Loop in C#

A while loop is used for executing a statement repeatedly until a given condition returns **false**. Here, statements may be a single statement or a block of statements. The loop iterates while the condition is **true**. If you see the syntax and flow chart parallelly, then you will get more clarity of the while loop.

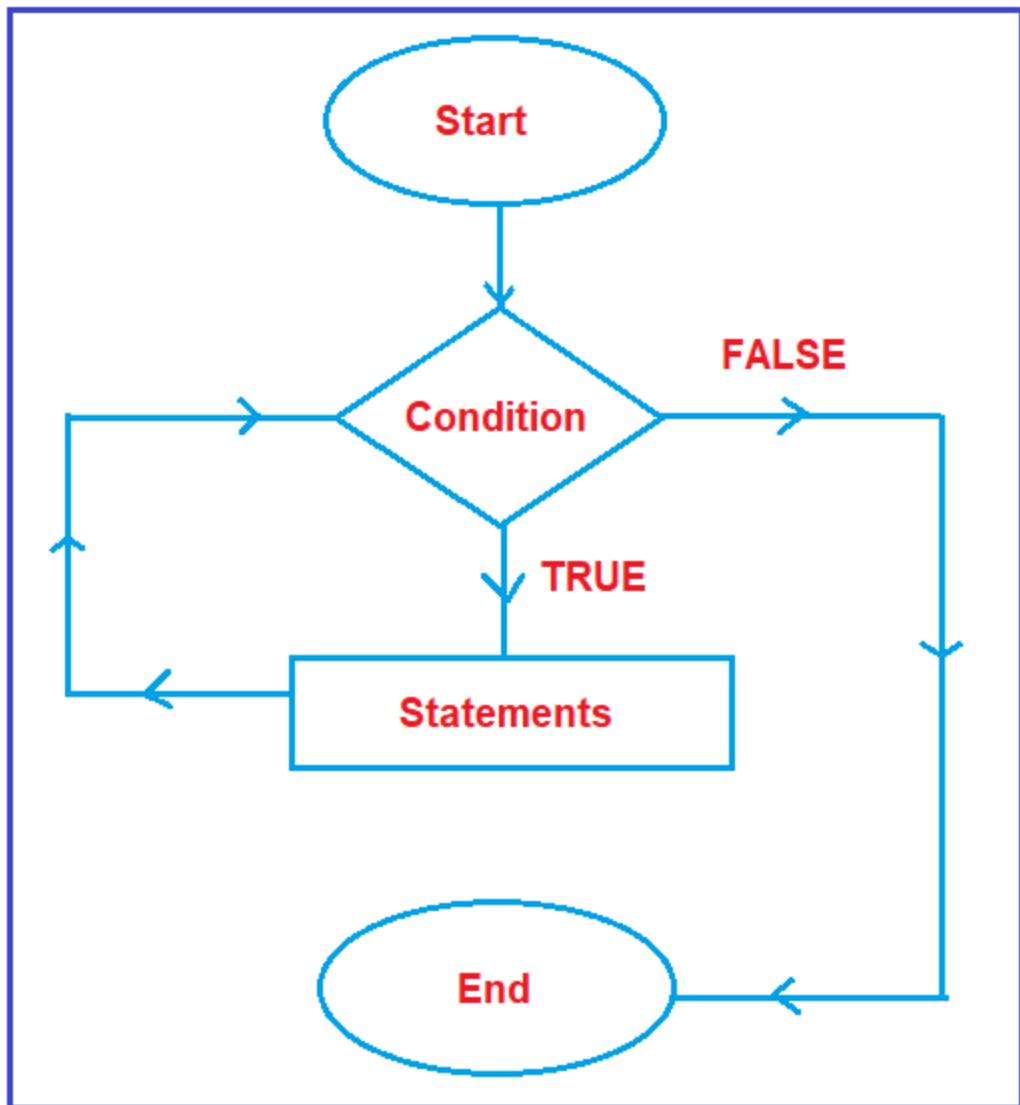
While Loop Syntax in C#

Syntax:

```
while (Condition)
{
    Statements;
}
```

image_56.png

Flow Chart of While Loop in C#



image_57.png

Example to understand While loop

In the below example,

- The variable x is initialized with value 1
- Then it has been tested for the condition.
- If the condition returns true then the statements inside the body of the while loop are executed else control comes out of the loop.
- The value of x is incremented using the ++ operator.
- Then it has been tested again for the loop condition.

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 1;
            while (x <= 5)
            {
                Console.WriteLine("Value of x:" + x);
                x++;
            }
            Console.ReadKey();
        }
    }
}
```

Output:

```
Value of x:1
Value of x:2
Value of x:3
Value of x:4
Value of x:5
```

Advanced Example to understand While loop Print the numbers in the following format up to a given number and that number is entered from the keyboard. 2 4 6 8 up to that given number

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int i, n;
            Console.Write("Enter a Number : ");
            n = Convert.ToInt32(Console.ReadLine());
            i = 2;
            while (i <= n)
            {
                Console.WriteLine($"{i} ");
                i = i + 2;
            }

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

Output:

```
Enter a Number : 10
2 4 6 8 10
```

Nested While Loop in C#

Writing a while loop inside another while loop is called nested while loop or you can say defining one while loop inside another while loop is called nested while loop. This is the reason why nested loops are also called “loops inside the loop”.

Nested While Loop Syntax

Syntax:

```
while (outer condition)
{
    Outer while Statements;

    while (inner condition)
    {
        Inner while Statements;
    }

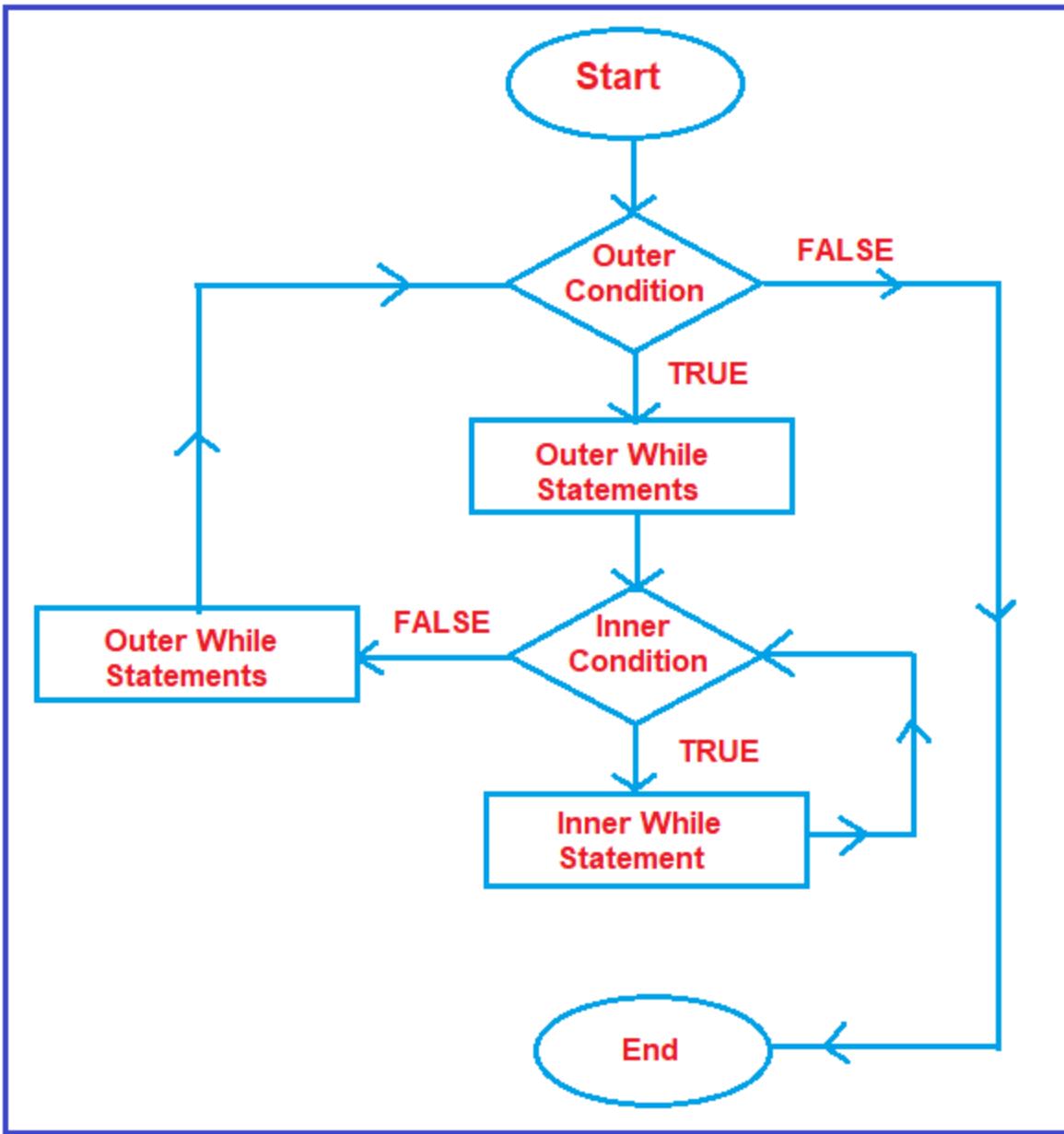
    Outer while Statements;
}
```

image_58.png



Note: In the nested while loop, the number of iterations will be equal to the number of iterations in the outer loop multiplied by the number of iterations in the inner loop. Nested while loops are mostly used for making various pattern programs in C# like number patterns or shape patterns.

Flow Chart of Nested While Loop



image_59.png

Example to Print the Following Format using Nested While Loop in C# Language

```
ENTER A NUMBER : 6
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```

image_60.png

```

using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("ENTER A NUMBER ");
            int n = Convert.ToInt32(Console.ReadLine());
            int i = 1;
            while (i <= n)
            {
                Console.WriteLine();
                int j = 1;
                while (j <= i)
                {
                    Console.Write(j + " ");
                    j++;
                }
                i++;
            }

            Console.ReadKey();
        }
    }
}

```

Do While Loop in C#

The do-while loop is a post-tested loop or exit-controlled loop i.e. first it will execute the loop body and then it will be going to test the condition. That means we need to use the do-while loop where we need to execute the loop body at least once. The do-while loop is mainly used in menu-driven programs where the termination condition depends upon the end-user.

Syntax to use Do While Loop in C#

```
do
{
    statements;
}
while(condition);
```

image_61.png

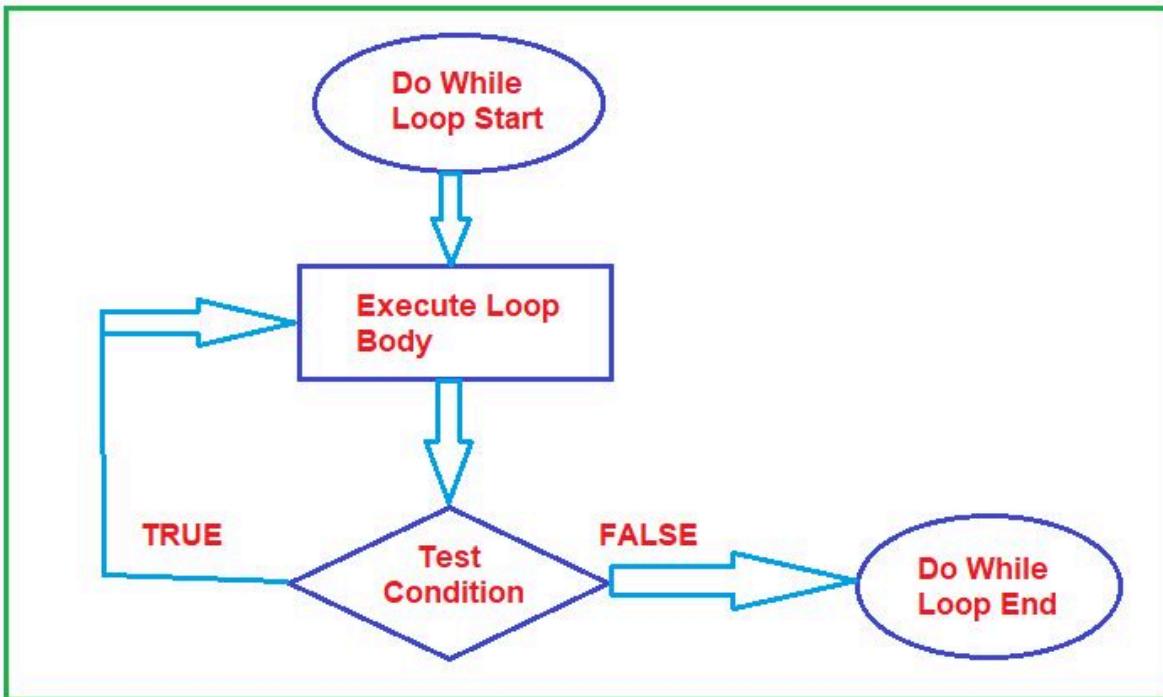
While and do-while are almost the same. So, what is the difference? Which one do we use?

The difference between the do-while loop and the while loop in C# is that the do-while evaluates its test condition at the bottom of the loop whereas the while loop evaluates its test condition at the top. Therefore, the statements written inside the do-while block are executed at least once whereas we cannot give a guarantee that the statements written inside the while loop are going to be executed at least once.



Note: When you want to execute the loop body at least once irrespective of the condition, then you need to use the do-while loop else you need to use the while loop.

Flow Chart of Do-While Loop



image_62.png

Example to understand do while loop in C#

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int number = 1;
            do
            {
                Console.Write($"{number} ");
                number++;
            } while (number <= 5);

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

```
    }  
}
```

Output: 1 2 3 4 5

Nested Do-While Loop in C#

Using a do-while loop inside another do-while loop is called nested do-while loop. The syntax to use the nested do-while loop in C# language is given below.

```
do  
{  
    statement n;  
    do  
    {  
        statement n;  
    }  
    while(test condition);  
}  
while(test expression);
```

image_63.png

Example to Understand Nested do-while Loop in C#

```
using System;  
namespace ControlFlowDemo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            do  
            {  
                Console.WriteLine("I am from outer do-while loop");  
                do  
                {  
                    Console.WriteLine("I am from inner do-while loop");  
                }  
                while (1 > 10);  
            }  
        }  
    }  
}
```

```
    while (2 > 10);

    Console.ReadKey();
}

}

}
```

Output:

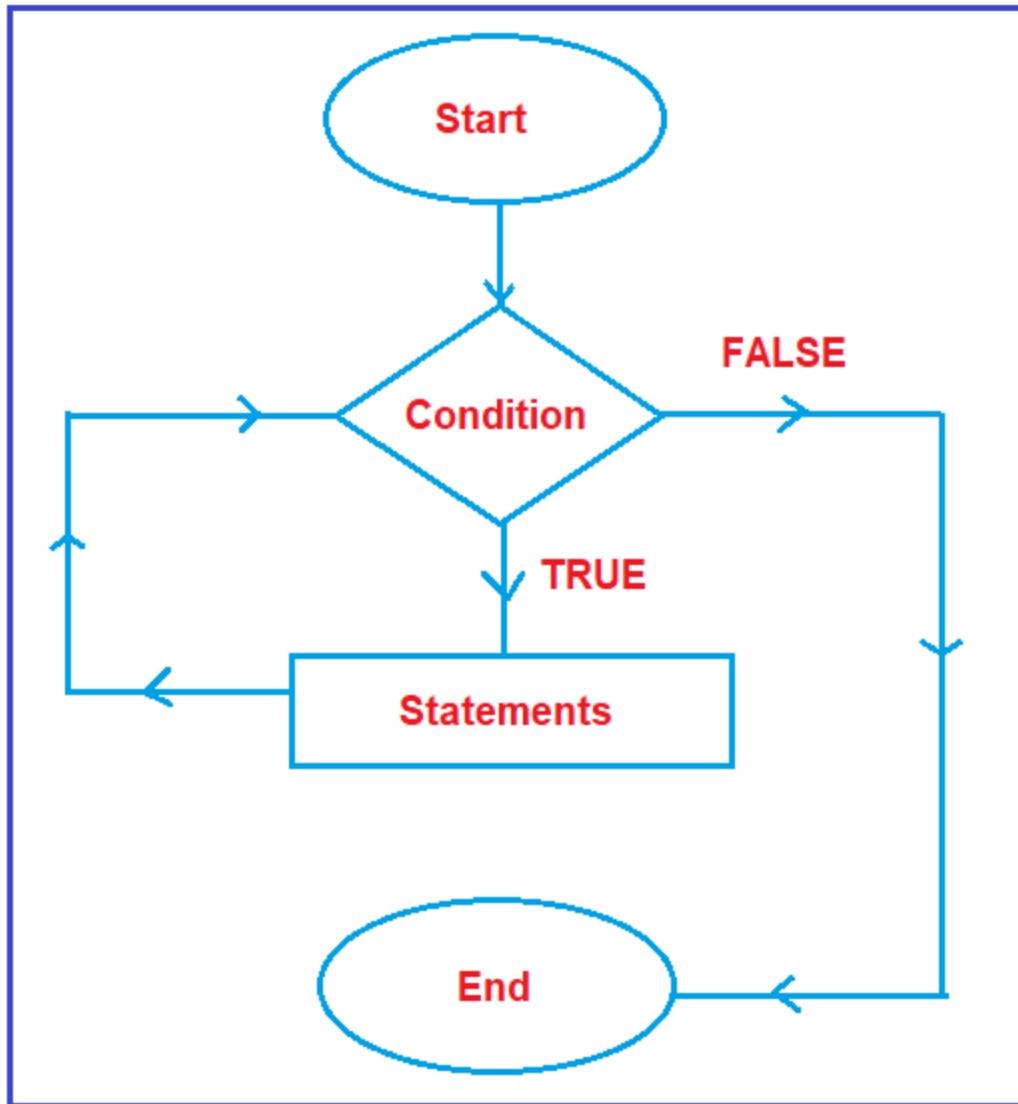
```
I am from outer do-while loop
I am from inner do-while loop
```

For Loop in C#

For loop is one of the most commonly used loops in the C# language. If we know the number of times, we want to execute some set of statements or instructions, then we should use for loop. For loop is known as a Counter loop. Whenever counting is involved for repetition, then we need to use for loop.

For Loop Flowchart

The following diagram shows the flowchart of the for loop.



image_64.png

Syntax to use For Loop in C#

The for loop allows the execution of instructions for a specific amount of time. It has four stages.

- Loop initialization
- Condition evaluation
- Execution of instruction
- Increment/Decrement

Now let's have a look at the for loop syntax:

```
for (initialization; condition; increment/decrement)
{
    // c# instructions
}
```

image_65.png

Explanation of the for-loop syntax:

- **Loop Initialization:** Loop initialization happens only once while executing the for loop, which means that the initialization part of for loop only executes once. Here, initialization means we need to initialize the counter variable.
- **Condition Evaluation:** Conditions in for loop are executed for each iteration and if the condition is true, it executes the C# instruction and if the condition is false then it comes out of the loop.
- **Execution of Instruction:** Once the condition is evaluated, and if the condition is true, then the control comes to the loop body i.e. the loop body is going to be executed.
- **Increment/Decrement:** After executing the loop body, the increment/decrement part of the for loop will be executed, and once it executes the increment decrement part i.e. once it increments and decrements the counter variable, again it will go to the condition evaluation stage.

Example to Print Numbers From 1 to n Using For Loop

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter one Integer Number:");
        }
    }
}
```

```
        int number = Convert.ToInt32(Console.ReadLine());
        for (int counter = 1; counter <= number; counter++)
        {
            Console.WriteLine(counter);
        }
        Console.ReadKey();
    }
}
```

Output:

```
1
2
3
4
5
```

You will get the same output as the previous example.

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter one Integer Number:");
            int number = Convert.ToInt32(Console.ReadLine());
            int counter = 1; //Initialization
            for (;counter <= number;)
            {
                Console.WriteLine(counter);
                counter++; //Updation
            }
            Console.ReadKey();
        }
}
```

```
    }  
}
```

- ⚠ Can we run for loop without condition in C#? Yes, we can run a for loop without condition. And, it will be an infinite loop. Because if we don't mention any termination condition in for loop, the for loop is not going to end.

Nested for Loop in C#:

When we created one for loop inside the body of another for loop, then it is said to be nested for loop in C# language. The syntax to use nested for loop is given below.

```
for (initialize counter; test condition; ++ or --)  
{  
    for (initialize counter; test condition; ++ or --)  
    {  
        . . . // inner for loop  
    }  
} // outer for loop
```

image_66.png

- ⚠ The point that you need to remember is when the inner for loop condition failed, then it will terminate the inner for loop only. And when the outer for loop condition failed, then it will terminate the outer for loop.

Example to Understand Nested For Loop in C#: In the below example, we have created a nested for loop. The outer for loop is going to be executed 5 times and for each iteration of the outer for loop, the inner for loop is going to execute 10 times.

```
using System;  
namespace ControlFlowDemo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            for (int i = 1; i <= 5; i++) //Outer For Loop
```

```

    {
        Console.WriteLine($"Outer For Loop : {i}");
        for (int j = 1; j <= 10; j++) //Inner For Loop
        {
            Console.Write($" {j}");
        }
        Console.WriteLine();
    }

    Console.ReadKey();
}
}
}

```

Output:

```

Outer For Loop : 1
1 2 3 4 5 6 7 8 9 10
Outer For Loop : 2
1 2 3 4 5 6 7 8 9 10
Outer For Loop : 3
1 2 3 4 5 6 7 8 9 10
Outer For Loop : 4
1 2 3 4 5 6 7 8 9 10
Outer For Loop : 5
1 2 3 4 5 6 7 8 9 10

```

Jumping Statements

The Jump Statements in C# are used to transfer control from one point or location or statement to another point or location or statement in the program due to some specified condition while executing the program.

The Jump Statements in C# Language are used to modify the behavior of conditional (if, else, switch) and iterative (for, while, and do-while) statements. The Jump Statements allow us to exit a loop, and start the next iteration, or explicitly transfer the program

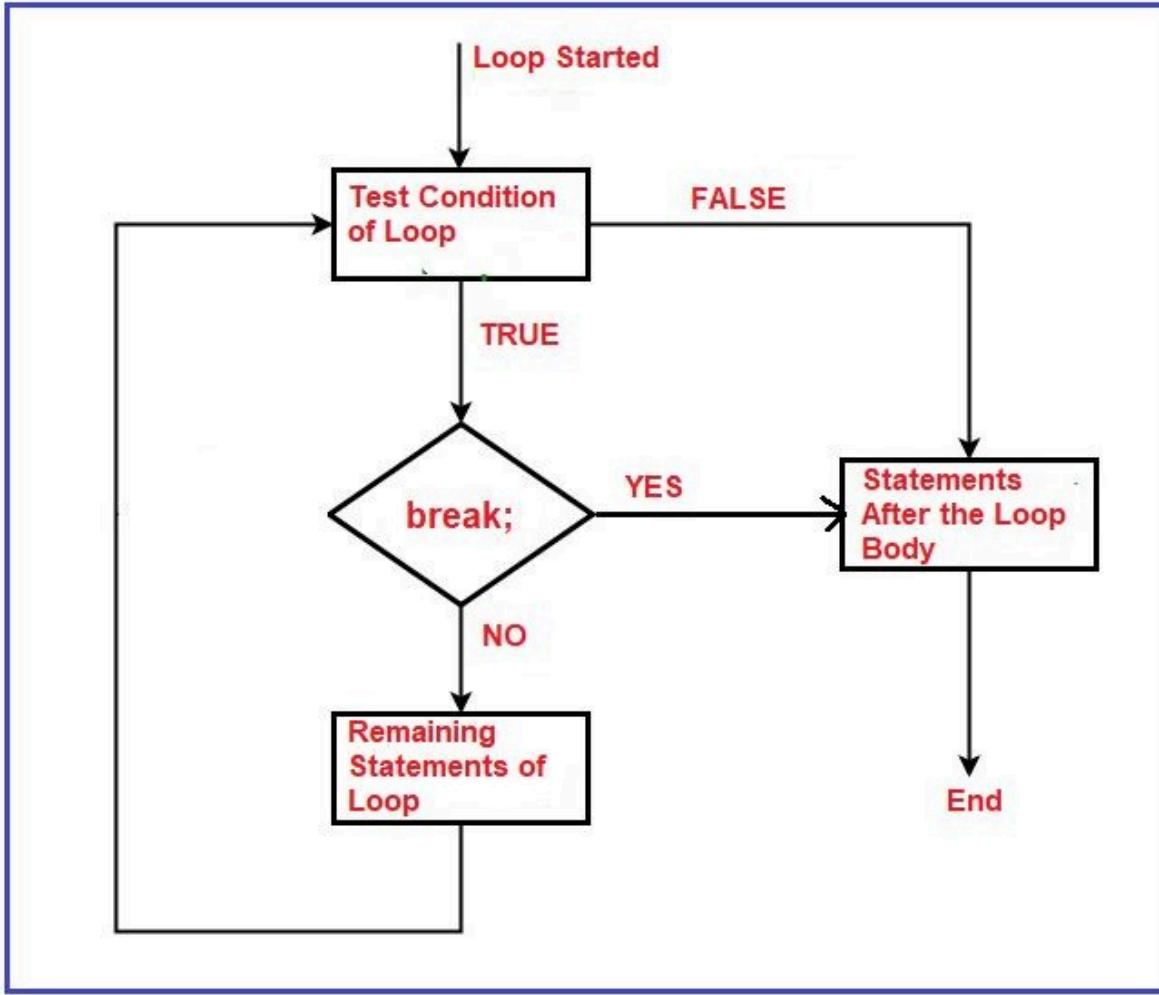
control to a specified location in your program. C# supports the following jump statements:

- **break**
- **continue**
- **goto**
- **return** (In the Function section we will discuss the return statement)
- **throw** (In the Exception Handling section we will discuss the throw statement)

Break Statement in C#

In C#, the break is a keyword. By using the break statement, we can terminate either the loop body or the switch body. The most important point that you need to keep in mind is the use of a break statement is optional but if you want to use then the break statement should be placed either within the loop body or switch body.

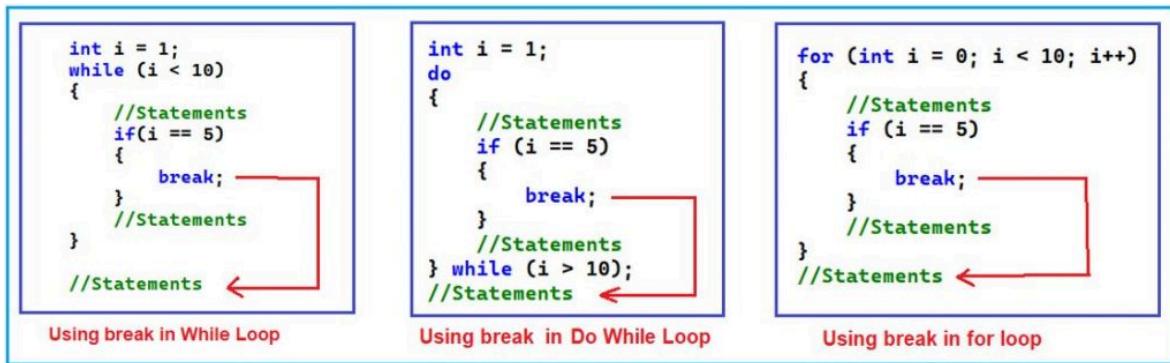
Flowchart of Break Statement:



image_67.png

When it encounters the break statement inside a loop body or switch body, then immediately it terminates the loop and switch execution and executes the statements which are present after the loop body or switch body. But if the break statement is not executed, then the statements which are present after the break statement will be executed and then it will continue its execution with the next iteration of the loop.

How does the break statement work in C#



image_68.png

If you notice the above code, we have written the if conditional statement inside the loop body, and within the if condition block, we have written the break statement. So, when the loop executes, in each iteration, the if condition will be checked and if the condition is false, then it will execute the statements which are present after the if block and continue with the next iteration. Now, what happens when the if condition is true? Once the if condition is evaluated to true, then the if block will be executed, and once the break statement within the if block is executed, it immediately terminates the loop, and the statements which are present after the loop block will be executed.

Example to Understand Break Statement in C#

In the below example, we have provided the condition for the loop to be executed 10 times i.e. starting from I value 1 to 10. But our requirement is when the I value becomes 5, we need to terminate the loop. In this case, we need to write if condition inside the loop body and check whether the current I value is equal to 5 or not. If it is not equal to 5, then continue the execution of the for loop and execute the next iteration. But if the I value is 5, then the if condition will return true, and, in that case, the break statement will be executed and once the break statement is executed, it will immediately terminate the loop body.

```
using System;
namespace JumpStatementDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 1; i <= 10; i++)
```

```

    {
        Console.WriteLine($"I : {i}");
        if (i == 5)
        {
            break;
        }
    }
    Console.WriteLine("Out of for-loop");

    Console.ReadKey();
}
}
}

```

Output:

```

I : 1
I : 2
I : 3
I : 4
I : 5
Out of for-loop

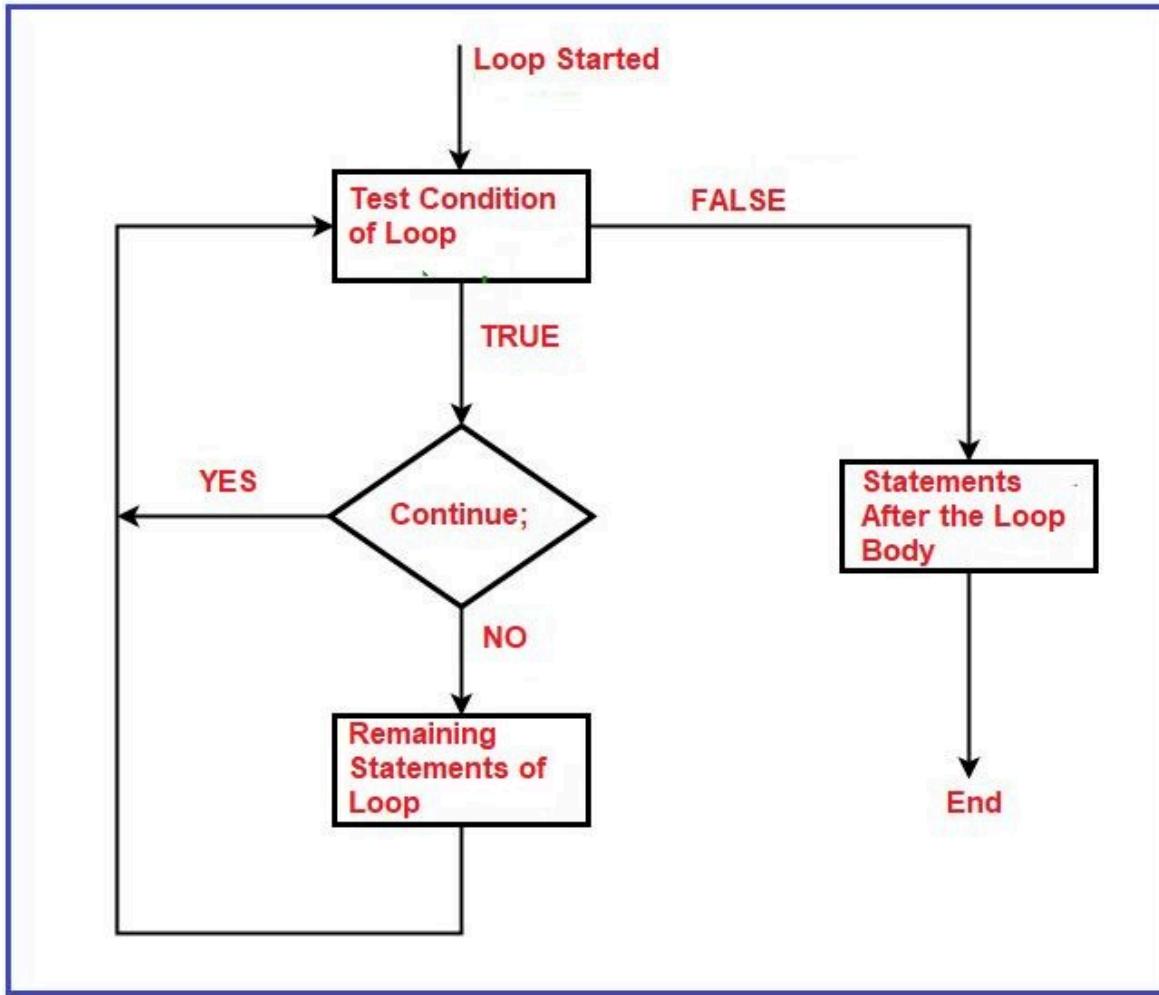
```

Continue Statement in C#

In C#, continue is a keyword. By using the continue keyword, we can skip the statement execution from the loop body. Like the break statement, the use of the continue statement is also optional but if you want to use then you can use it only within the loop body.



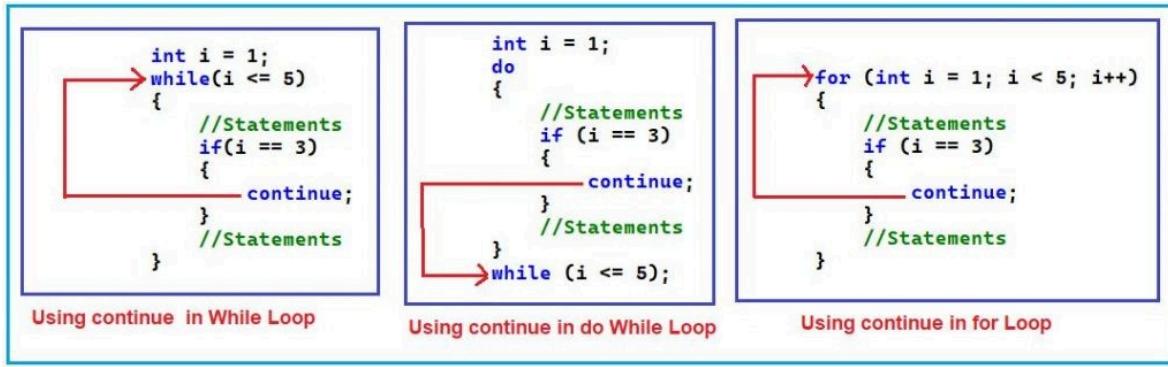
The BREAK statement terminates the loop, whereas the CONTINUE statement skips only the current loop iteration, and allows the next loop iteration to proceed. The continue statement is almost always used with the if...else statement.



image_69.png

The continue statement in C# is very similar to the break statement, except that instead of terminating the loop, it will skip the current iteration and continue with the next iteration. That means the continue statement skips the rest of the body of the loop and immediately checks the loop's condition.

How Does the Continue Statement Work in C#



image_70.png

If you notice the above code, we have written the if conditional statement inside the loop body, and within the if condition block, we have written the continue statement. So, when the loop executes, in each iteration, the if condition will be checked and if the condition is false, then it will execute the statements which are present after the if block and continue with the next iteration. Now, what happens when the if condition is true? Once the if condition is evaluated to true, then the if block will be executed, and once the continue statement within the if block is executed, it will skip the execution of the statements which are present after the continue statement and continue with the execution of the next iteration of the loop.

Example to Understand Continue Statement in C#

```
using System;
namespace JumpStatementDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 1; i <= 5; i++)
            {
                if (i == 3)
                {
                    continue;
                }
                Console.WriteLine($"I : {i}");
            }
        }
    }
}
```

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

Output:

```
I : 1
I : 2
I : 3
I : 4
I : 5
```

Functions in C#

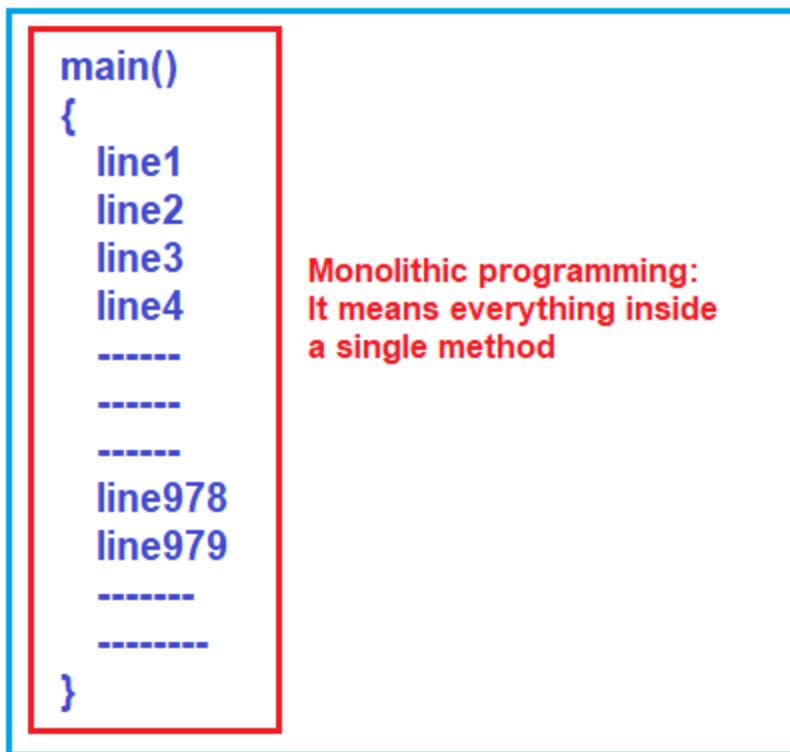
What are the Functions of C#

A function is a group of related instructions that performs a specific task. It can be a small or big task, but the function will perform that task completely. Functions take some input as parameters and return the result as a return value.

Why do we need functions?

Let us understand why we need functions with an example. Functions are also called modules or procedures. Instead of writing a single main program, i.e., everything inside the main function, we can break the main function into small manageable size pieces and separate the repeating tasks or smaller tasks as a function.

For Example, if we write one program and put everything inside the main function, then such a type of programming approach is called Monolithic Programming. If your main function contains thousands of lines of code, then it is becoming very difficult to manage. This is actually not a good programming approach.



image_80.png

Problems in Monolithic Programming:

1. **First problem:** if there is an error in a single line, it's an error in the entire program or main function.
2. **Second problem:** 10000 lines of code we cannot finish in one hour or one day; it might take a few days, and we should remember everything throughout that time. Then only we can make changes or write new lines in the program. So, we should memorize the whole program.
3. **Third problem:** How many people can write this one single main function? Only one person can write. We cannot make it a teamwork and more than one person can't work on the same main function. So, work cannot be distributed in a team.
4. **Fourth problem:** when this program becomes very big, it may fit in some computer memories, and it may not fit in some of the memories. It Depends on the size and the hardware contribution of the computer you are running.

So, these are the few problems due to *monolithic programming*. Monolithic means everything is a single unit.

We prefer to break the program into pieces, manageable and small pieces, and reusable pieces.

So, suppose we break the program into smaller tasks, i.e., into many smaller functions, and each function performs a specific task. In that case, such type of programming is called “modular programming” or “procedural programming,” and this approach is good for development.

```
function1()
{
    //function1 task
}
function2()
{
    //function2 task
}
function3()
{
    //function3 task
}

main()
{
    function1();
    function2();
    function3();
}
```

**Modular programming or
Procedural Programming**

image_81.png

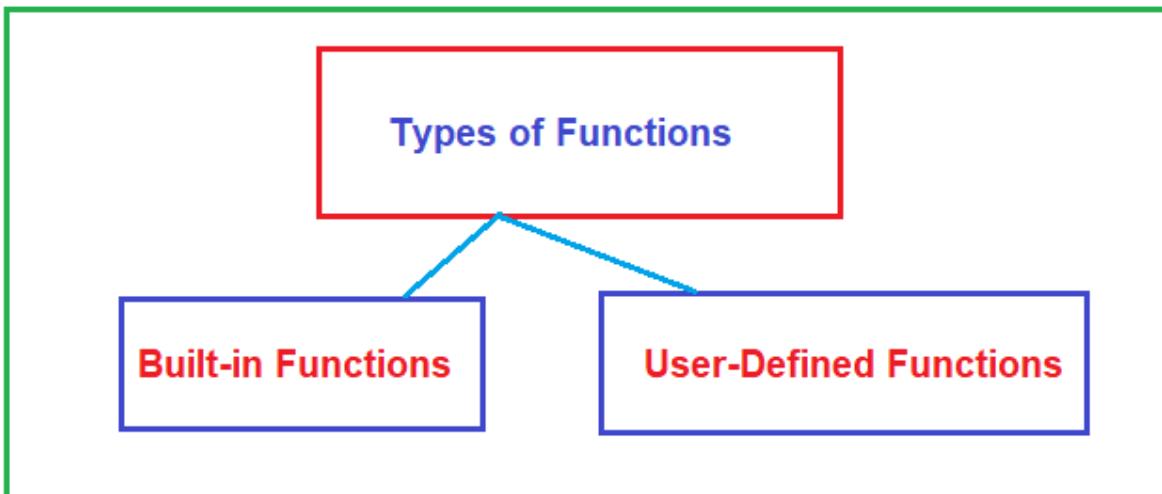
As shown in the above image, the first function, i.e., function1(), will perform some specific task, and another function, i.e., function2(), will perform some other task, and similarly, function3() may perform some task. So, in this way, we can break the larger task into smaller simple tasks, and then we can use them all together inside the main function.

Here, in the **Modular Programming Approach**, you can break the program into smaller tasks, focus on smaller tasks, finish them, and make them perfect. It is easy for one single individual to develop the application, even if you can break this software project into a team of programmers where each programmer will focus on one or many smaller tasks.

Types of Functions in C#:

Basically, there are two types of Functions in C#. They are as follows:

- Built-in Functions
- User-Defined Functions



image_82.png

- ⚠ The function which is already defined in the framework and available to be used by the developer or programmer is called a built-in function, whereas if the function is defined by the developer or programmer explicitly, then it is called a user-defined function.

Functions Vs Methods in C#

Functions:

- ⚠ A function is a generic term for a block of code that performs a specific task and can return a value.
- ⚠ In C#, technically all functions are methods, because C# doesn't allow free-floating functions — every function must be defined inside a class or struct.
- ⚠ However, when we say “function,” we’re usually talking in general programming terms, not C# specifically.

Example (conceptual “function”):

```
int Add(int a, int b)
{
    return a + b;
}
```

⚠ In C#, this would be inside a class — so it's actually a method (see below).

Methods:

- **⚠** A method is a function that belongs to a class or object.
- **⚠** It defines the behavior of that class or object.
- Methods can be:
 - **⚠** Instance methods (require an object instance)*
 - **⚠** Static methods (belong to the class itself, not any instance)

```
public class Calculator
{
    // Instance method
    public int Add(int a, int b)
    {
        return a + b;
    }

    // Static method
    public static int Multiply(int a, int b)
    {
        return a * b;
    }
}
```

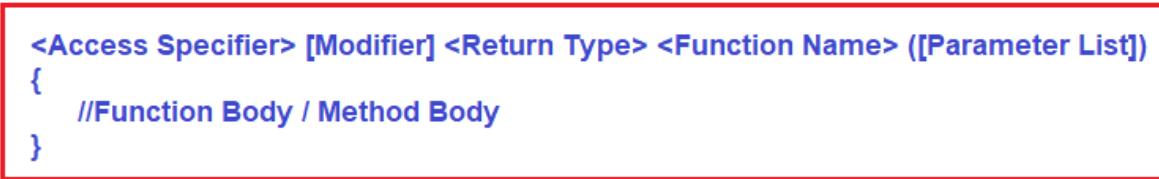
```
    }  
}
```

Usage:

```
Calculator calc = new Calculator();  
int sum = calc.Add(3, 5);           // Instance method  
int product = Calculator.Multiply(3, 5); // Static method
```

User-Defined Function in C#

First of all, the function should have a **name that is mandatory**. Then it should have a **parameter list** (the parameters it is taking) which is optional. Then the function should have a **return type which is mandatory**. A function can have an access specifier, which is optional, and a modifier which is also optional. For a better understanding, please have a look at the below image.



```
<Access Specifier> [Modifier] <Return Type> <Function Name> ([Parameter List])  
{  
    //Function Body / Method Body  
}
```

image_83.png

Example to Understand No Arguments Passed and No Return Value Function:

In the below example, the Sum() function does not take any parameters, or even it does not return a value. The return type of the function is void. Hence, no value is returned from the function.

```
using System;  
namespace FunctionDemo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Sum();
```

```

        Console.ReadKey();
    }
    static void Sum()
    {
        int x = 10;
        int y = 20;
        int sum = x + y;
        Console.WriteLine($"Sum of {x} and {y} is {sum}");
    }
}

```

Output: Sum of 10 and 20 is 30

No Arguments Passed but Return a Value Function

When a function has no arguments, it receives no data from the calling function but returns a value. The calling function receives the data from the called function. So, there is no data transfer between the calling function to called function but data transfer from the called function to the calling function. The called function is executed line by line in a normal fashion until the return statement is encountered.

Example to Understand No Arguments Passed but Return a Value Function :

In the below example, the empty parentheses in int Result = Sum(); statement indicates that no argument is passed to the function. And the value returned from the function is assigned to the Result variable.

```

using System;
namespace FunctionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int Result=Sum();
            Console.WriteLine($"Sum is {Result}");
            Console.ReadKey();
        }
    }
}

```

```

static int Sum()
{
    int x = 10;
    int y = 20;
    int sum = x + y;
    return sum;
}
}

```

Output: Sum is 30

Argument Passed but no Return Value Function :

When a function has arguments, it receives data from the calling function but does not return any value. So, there is data transfer between the calling function to called function, but there is no data transfer from the called function to the calling function. The nature of data communication between the calling function and the called function with arguments but no return value.

Example to Understand Argument Passed but no Return Value Function :

In the example below, we pass two values to the Sum function, but the Sum function does not return any value to the main function.

```

using System;
namespace FunctionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10, y = 20;
            Sum(x, y);
            Console.ReadKey();
        }
        static void Sum(int x, int y)
        {
            int sum = x + y;
        }
    }
}

```

```
        Console.WriteLine($"Sum is {sum}");
    }
}
}
```

Output: Sum is 30

Argument Passed and Return Value Function in C# Language:

A self-contained and independent function should behave like a “*black box*” that receives an input and outputs a value. Such functions will have two-way data communication.

```
using System;
namespace FunctionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10, y = 20;
            int Result = Sum(x, y);
            Console.WriteLine($"Sum is {Result}");
            Console.ReadKey();
        }

        static int Sum(int x, int y)
        {
            int sum = x + y;
            return sum;
        }
    }
}
```

Output: Sum is 30

Function Overloading

In C#, we can write more than one function with the **same name**, but with a different argument or parameter list, and when we do so, it is called function overloading. Let us understand this with an example.

```
static void main(){
    int a = 10, b = 2, c;
    c = add(a, b);
}
```

This is our main function. Inside this function, we have declared 3 variables. Next, we store the result of the '*add()*' function in the 'c' variable. The following is the add function.

```
static int add(int x, int y){
    return x + y;
}
```

Here we haven't declared any variable; return ' $x + y$ '. When we call the 'add' function inside the main function, then 'a' will be copied in 'x', and 'b' will be copied in 'y', and it will add these two numbers, and the result will store in 'c'. Now we want to write one more function here,

```
static int add(int x, int y, int z){
    return x + y + z;
}
```

We have changed the main function as follows.

```
static void main(){
int a = 10, b = 2, c, d;
    c = add (a, b);
    d = add (a, b, c);
}
```

Here we have created another function with the same name, 'add', but it takes 3 parameters. Inside the main function, we have called '**add(x,y,z)**' and stored the result in

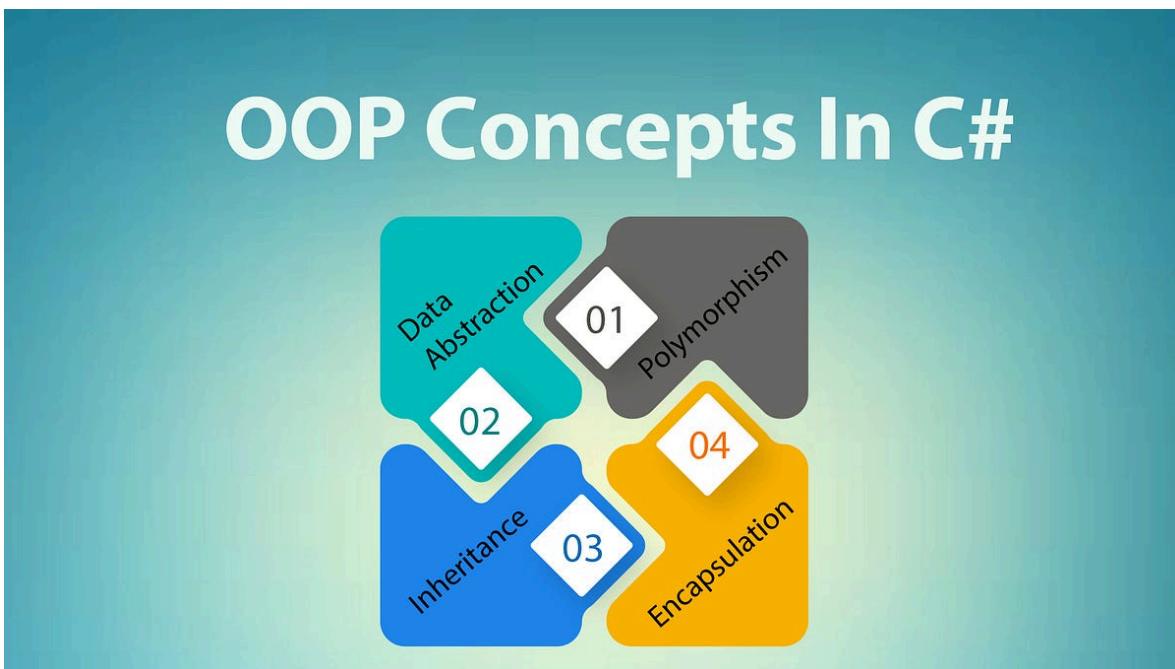
the ‘d’ variable. So, we can have two functions with the same name but with different parameters

So when we call “ **add(a, b)**” it will be calling **add(int x, int y)**, and when we call ‘ **add(a, b, c)**’ it will be “ **add(int x, int y, int z)**”. The C# compiler can differentiate between these two functions, and this is the concept of ***function overloading in C#.***

Example to Understand Function Overloading

```
using System;
namespace FunctionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 10, b = 2, c, d;
            c = add(a, b);
            Console.WriteLine($"Sum of {a} and {b} is {c}");
            d = add(a, b, c);
            Console.WriteLine($"Sum of {a} and {b} and {c} is {d}");
            Console.ReadKey();
        }
        static int add(int x, int y)
        {
            return x + y;
        }
        static int add(int x, int y, int z)
        {
            return x + y + z;
        }
    }
}
```

Object-Oriented Programming



image_286.png

In this article, I will give an overview of Object-Oriented Programming (OOPs) in C#, i.e., discuss the OOPs Principles in C#. Object-Oriented Programming, commonly known as OOPs, is a technique, not a technology. It means it doesn't provide any syntaxes or APIs; instead, it provides suggestions to design and develop objects in programming languages. As part of this article, we will cover the following OOP concepts in C#.

How do we Develop Applications?

Object-Oriented Programming is a strategy that provides some principles for developing applications or software. It is a methodology. Like OOPs, other methodologies exist, such as Structured Programming, Procedural Programming, or Modular Programming. But nowadays, one of the well-known and famous styles is Object Orientation, i.e., Object-Oriented Programming

Nowadays, almost all the latest programming languages support object orientation. This object orientation is more related to the designing of software, and this deals with the internal design of the software, not the external design. So, it is nowhere related to the users of the software. It is related to the programmers who are working on developing software.

Object-Oriented vs Modular Programming

Now, I will explain to you Object Orientation by comparing it with Modular Programming. The reason is that people who came to learn C# already know the C language. The C programming language supports Modular or Procedural Programming. Based on that, I can give you an idea of how object orientation differs from modular programming. Let us compare Object-Oriented vs Modular Programming through some examples.

Bank Example

So first, we are taking an example of a bank. If you're developing an application for a bank using modular programming, how do you see the system, how do you see the workings of a bank, and what will be your design? That depends on how you understand it and how you see the system. So, let us see how we look at the bank system using modular programming.

In a bank, you can *open an account*, you can *deposit an amount*, you can *withdraw an amount*, you can *check your account balance*, or you can also *apply for a loan*, and so on. So, these are the things that you can do at the bank.

So, Opening an Account, Depositing Money, Withdrawing Money, Checking Your Balance, and Applying For a Loan are functions. All these are nothing but functions. And you can do the specific operations by calling that specific function. So, if you're developing software for a bank, it is nothing but a collection of functions. So, the bank application will be based on these functions, and the user of the bank application will be utilizing these functions to perform his required task. So, you will develop software as a set of functions in Modular Programming

Why Object Orientation?

Let us talk about a manufacturing firm which manufactures cars or vehicles. If you look at that manufacturing farm, then it may be working in the form of departments like one is an inventory department that maintains the stock of raw materials and one is manufacturing, which is the production work that they do, and one department will be looking at sales and one department is looking at marketing. One is about payroll, and one is for accounts, and so on. So, there may be many departments.

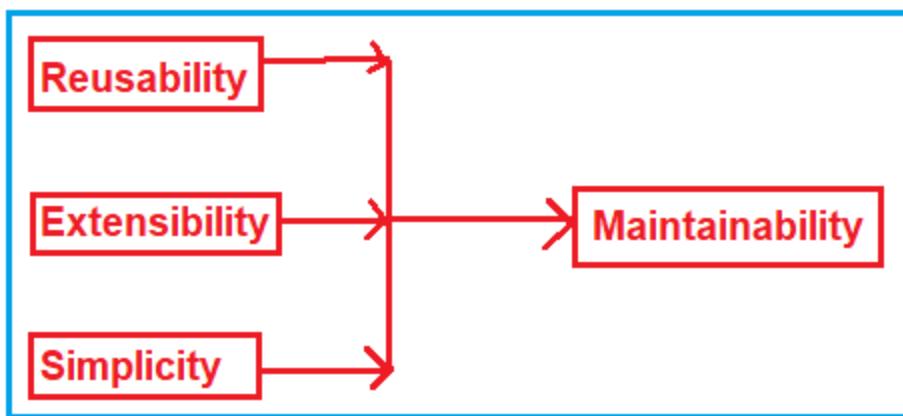
Suppose you are developing software only for *payroll* or *inventory purposes*. In that case, you may look at the system just like a *modular approach*, and in that, you can find functions like *placing an order* and *checking the item in stock*. These types of things can

have a set of functions so that you can develop the software only for the inventory system as a collection of functions. Still, **when developing software for the entire organization, you must see things in objects**.

⚠ So, the **inventory item is an object**, an **employee is an object**, an **account is an object**, and a **product manufacturer is an object**. The **machines used for production are an object**. So, all these things are objects.

What are the Problems of Modular Programming?

Modular programming has the following problems.



image_84.png

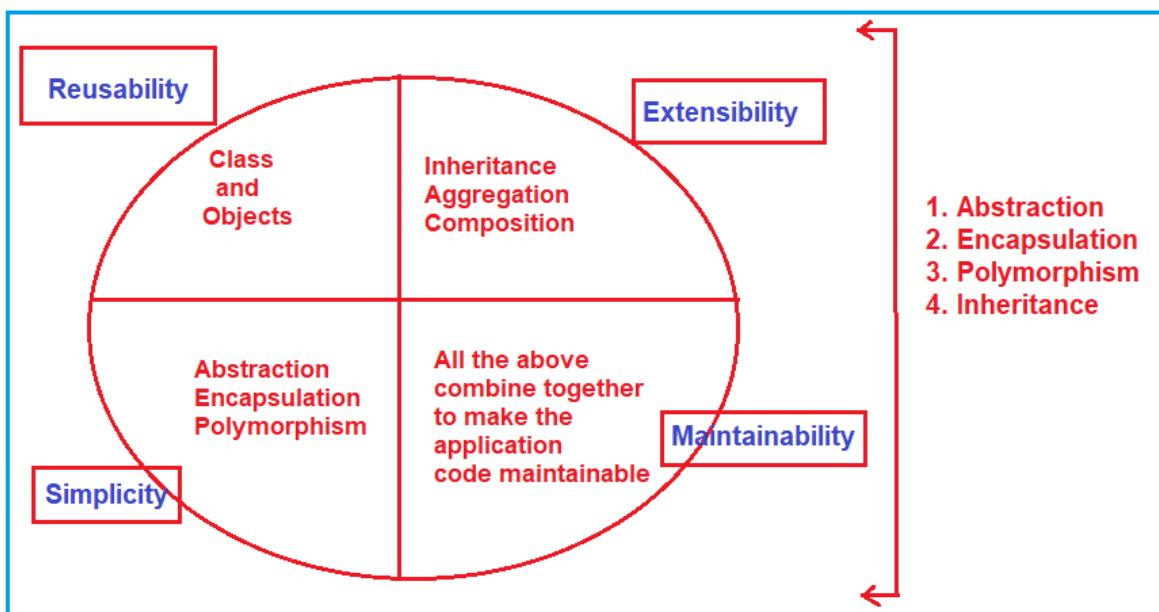
- **Reusability:** In Modular Programming, we must write the same code or logic at multiple places, increasing code duplication. Later, if we want to change the logic, we must change it everywhere.
- **Extensibility:** It is not possible in modular programming to extend the features of a function. Suppose you have a function and you want to extend it with some additional features; then it is not possible. You have to create an entirely new function and then change the function as per your requirement.
- **Simplicity:** As extensibility and reusability are impossible in Modular Programming, we usually end up with many functions and scattered code.
- **Maintainability:** As we don't have Reusability, Extensibility, and Simplicity in modular Programming, it is very difficult to manage and maintain the application code.

How Can We Overcome Modular Programming Problems?

We can overcome the modular programming problems (*Reusability, Extensibility, Simplicity, and Maintainability*) using Object-Oriented Programming. OOPs provide some principles, and using those principles, we can overcome Modular Programming Problems

What Is Object-Oriented Programming?

Let us understand Object-Oriented Programming, i.e., OOP concepts using C#. Object-oriented programming (OOPs) in C# is a design approach where we think in terms of real-world objects rather than functions or methods. Unlike procedural programming language, in OOPs, programs are organized around objects and data rather than action and logic. Please have a look at the following diagram to understand this better.



image_85.png

- **Reusability:** To address reusability, object-oriented programming provides something called Classes and Objects. So, rather than copy-pasting the same code repeatedly in different places, you can create a class and make an instance of the class, which is called an object, and reuse it whenever you want.
- **Extensibility:** Suppose you have a function and want to extend it with some new features that were impossible with functional programming. You have to create an entirely new function and then change the whole function to whatever you want.

OOPs, this problem is addressed using concepts called Inheritance, Aggregation, and Composition. In our upcoming article, we will discuss all these concepts in detail.

- Simplicity: Because we don't have extensibility and reusability in modular programming, we end up with lots of functions and scattered code, and from anywhere we can access the functions, security is less. In OOPs, this problem is addressed using Abstraction, Encapsulation, and Polymorphism concepts.
- Maintainability: As OOPs address Reusability, Extensibility, and Simplicity, we have good, maintainable, and clean code, increasing the application's maintainability.

What are the OOPs Principles or OOPs Concepts?

OOPs provide 4 principles. They are :

- Abstraction: The process of representing the essential features without including the background details. In simple words, we can say that it is a process of defining a class by providing necessary details to the external world, which are required by hiding or removing unnecessary things.
- Inheritance: The process by which the members of one class are transferred to another class. The class from which the members are transferred is called the Parent/Base/Superclass, and the class that inherits the Parent/Base/Superclass members is called the Derived/Child/Subclass. We can achieve code *extensibility* through inheritance.
- Polymorphism: is derived from the Greek word, where Poly means many and morph means faces/ behaviors. So, the word polymorphism means the ability to take more than one form. Technically, we can say that the same function/operator will show different behaviors by taking different types of values or with a different number of values.
- Encapsulation : The process of binding the data and functions together into a single unit (i.e., class). In simple words, we can say that it is a process of defining a class by hiding its internal data members from outside the class and accessing those internal data members only through publicly exposed methods or properties.



⚠️ 🎉 Note: Don't consider Class and Objects as OOPs principle. We use classes and objects to implement OOP Principles.

Class & Objects

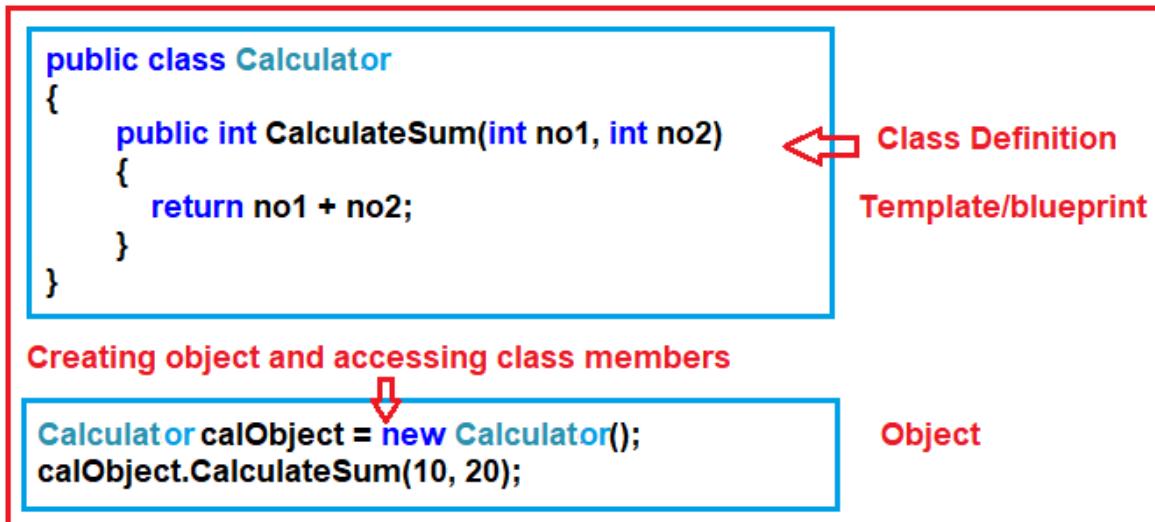
As we already discussed in our previous article, class, and objects addresses the reusability functionality. Again we discussed in Object-Oriented Programming, we need to think in terms of objects rather than functions. So, let us discuss what exactly classes and objects are from the Layman point of view as well as from the programming point of view.

Class: A class is simply a *user-defined data type* that represents both *state* and *behavior*. The *state represents the properties* and *behavior is the action* that objects can perform. In other words, we can say that a class is the blueprint/plan/template that describes the details of an object. A class is a blueprint from which the individual objects are created. In C#, a Class is composed of three things i.e. a name, attributes, and operations.

Objects: It is an instance of a class. A class is brought live by creating objects. An object can be considered as a thing that can perform activities. The set of activities that the object performs defines the object's behavior. All the members of a class can be accessed through the object. To access the class members, we need to use the dot (.) operator. The dot operator links the name of an object with the name of a member of a class.

How can we create a Class and Object in C#?

Let us understand how to create class and object in C#. In order to understand this, please have a look at the following image. As you can see in the below image, a class definition starts with the keyword `class` followed by the class name (here the class name is `Calculator`), and the class body is enclosed by a pair of curly braces. As part of the class body, you define class members (properties, methods, variables, etc.). Here as part of the body, we define one method called `CalculateSum`. The class `Calculator` is just a template. In order to use this class or template, you need an object. As you can see in the second part of the image, we create an object of the class `Calculator` using the `new` keyword. And then store the object reference on the variable `calObject` which is of type `Calculator`. Now, using this `calObject` object we can access the class members using a dot.



image_86.png

So, the point that you need to remember is, to create a class you need to use the class keyword while if you want to create an object of a class then you need to use the new keyword. Once you create the object then you can access the class members using the object.

```

using System;
namespace ClassObjectsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            //Creating object
            Calculator calObject = new Calculator();

            //Accessing Calculator class member using Calculator class
            object
            int result = calObject.CalculateSum(10, 20);

            Console.WriteLine(result);
            Console.ReadKey();
        }
    }
}

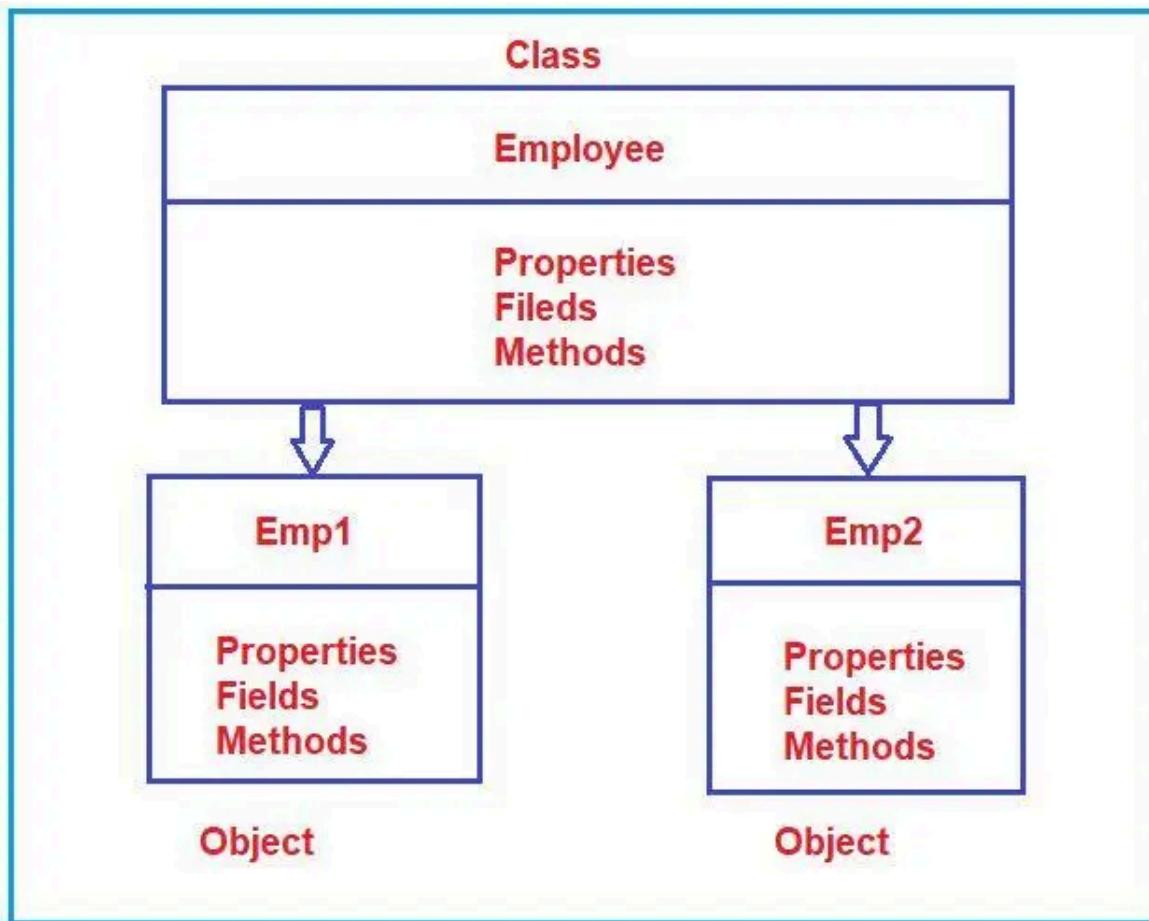
```

```
//Defining class or blueprint or template
public class Calculator
{
    public int CalculateSum(int no1, int no2)
    {
        return no1 + no2;
    }
}
```

Output: 30

Difference between Class and Objects in C#

Many programmers or developers still get confused by the difference between class and object. As we already discussed, in object-oriented programming, a Class is a template or blueprint for creating Objects, and every Object in C# must belong to a Class. Please have a look at the following diagram to understand the difference between them.



image_87.png

As you can see in the above image, here we have one class called “Employee”. All the Employees have some properties such as employee id, name, salary, gender, department, etc. These properties are nothing but the attributes (properties or fields) of the Employee class.

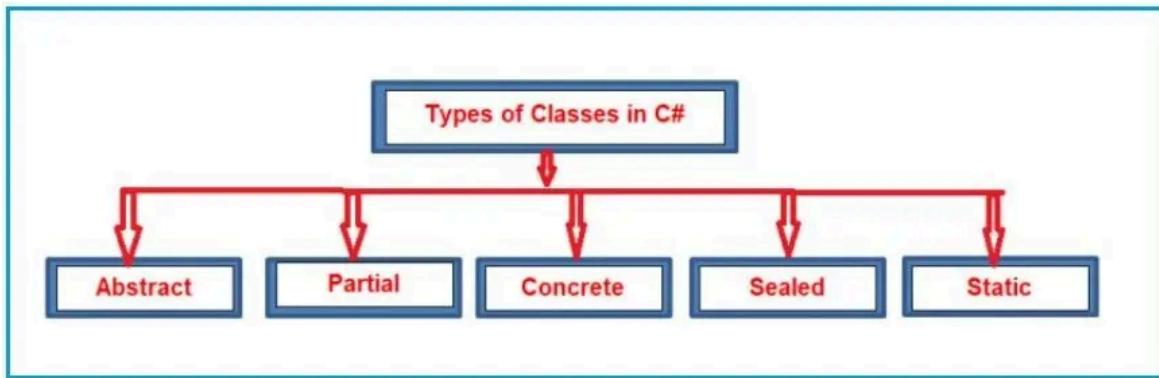
If required you can also add some methods (functions) that are common to all

Employees such as InsertData and DisplayData to insert and display the Employee Data.

So, the idea is that the template or blueprint of the Employee is not going to change.

Each and every Object is going to build from the same template (Class) and therefore contains the same set of methods and properties. Here, all Objects share the same template but maintain a separate copy of the member data (Properties or fields).

Types of Classes in C#:



image_88.png

Constructors in C#

It is a special method present inside a class responsible for initializing the variables of that class.

How to Define the Constructor Explicitly

We can also define the constructor explicitly in C#. The following is the explicit constructor syntax.

```
[<modifiers>] <class name> ([<Parameter List>])
{
    //Statements
}
```

image_89.png

Whenever we are creating an instance, there will be a call to the class constructor. For a better understanding, please have a look at the below example. Here, we defined one parameter less constructor explicitly, and then from the Main method, we create an instance. When we create the instance, it will make a call to the constructor, and the statements written inside the constructor will be executed. In this case, it will execute the print statement in the console.

```
using System;
namespace ConstructorDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            ExplicitConstructor obj = new ExplicitConstructor();

            Console.ReadKey();
        }
    }
    class ExplicitConstructor
```

```
{  
    public ExplicitConstructor()  
    {  
        Console.WriteLine("Explicit Constructor is Called!");  
    }  
}  
}
```

Output: Explicit Constructor is Called!

One more important point that you need to remember is, how many instances you created, and that many times the constructor is called for us. Let us prove this. Please modify the example code as follows. Here, I am creating the instance four times and it should and must call the constructor 4 times and we should see the print statement four times in the console window.

```
using System;  
namespace ConstructorDemo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            ExplicitConstructor obj1 = new ExplicitConstructor();  
            ExplicitConstructor obj2 = new ExplicitConstructor();  
            ExplicitConstructor obj3 = new ExplicitConstructor();  
            ExplicitConstructor obj4 = new ExplicitConstructor();  
  
            Console.ReadKey();  
        }  
    }  
    class ExplicitConstructor  
    {  
        public ExplicitConstructor()  
        {  
            Console.WriteLine("Explicit Constructor is Called!");  
        }  
    }  
}
```

```
    }  
}
```

output :

```
Explicit Constructor is Called!  
Explicit Constructor is Called!  
Explicit Constructor is Called!  
Explicit Constructor is Called!
```

We should not use the word **Implicitly** while calling the constructor in C#, why? See, if we are not defining any constructor explicitly, then the compiler will provide the constructor which is called **Implicitly Constructor**. See, the following example. If you move the mouse pointer over the Test class, then you will see the following. Here, Test is a class present under the **ConstructorDemo** namespace.

The screenshot shows a C# code editor with the following code:

```
using System;  
namespace ConstructorDemo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Test obj = new Test();  
        }  
    }  
    class Test  
    {  
        public int i;  
        public bool b;  
        public string s;  
    }  
}
```

A tooltip is displayed over the word "Test" in the line "Test obj = new Test();". The tooltip contains the text "class ConstructorDemo.Test" with a red arrow pointing to it. Below the tooltip, a red box contains the text "Here, Test is a class belongs to ConstructorDemo namespace".

image_90.png

Now, move the mouse pointer to Test() as shown in the below image. Here, the first Test is the class name and the second Test() is the constructor. That means we are calling the constructor explicitly.

The screenshot shows a C# code editor with the following code:

```
namespace ConstructorDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Test obj = new Test();
            Console.ReadKey();
        }
    }
    class Test
    {
        public int i;
        public bool b;
        public string s;
    }
}
```

A tooltip for the expression `Test.Test()` is displayed, showing the fully qualified name `Test.Test()`. Two red arrows point from the text annotations to this tooltip. The annotation "Making call to Constructor" is positioned above the tooltip, and the annotation "Test is the class name" is positioned below it.

image_91.png

Here, we are explicitly making a call to the constructor and when we call the constructor, the implicit constructor which is provided by the compiler is called and will initialize the variables.

What are the rules to follow while working with C# Constructor?

- The constructor's name should be the same as the class name.
- It should not contain a return type even void also.
- As part of the constructor body return statement with a value is not allowed.

User-Defined Default Constructor

The constructor which is defined by the user without any parameter is called the user-defined default constructor. This constructor does not accept any argument but as part of the constructor body, you can write your own logic.

Example to understand User-defined Default Constructor

In the below example, within the Employee class, we have created a public parameterless constructor which is used to initialize the variables with some default hard-coded values. And then from the Main method, we created an instance of the Employee class and invoke the Display method.

```
using System;
namespace ConstructorDemo
{
    class Employee
    {
        public int Id, Age;
        public string Address, Name;
        public bool IsPermanent;

        //User Defined Default Constructor
        public Employee()
        {
            Id = 100;
            Age = 30;
            Address = "Bhubaneswar";
            Name = "Anurag";
            IsPermanent = true;
        }

        public void Display()
        {
            Console.WriteLine("Employee Id is: " + Id);
            Console.WriteLine("Employee Age is: " + Age);
            Console.WriteLine("Employee Address is: " + Address);
            Console.WriteLine("Employee Name is: " + Name);
            Console.WriteLine("Is Employee Permanent: " + IsPermanent);
        }
    }

    class Program
    {
```

```

    static void Main(string[] args)
    {
        Employee e1 = new Employee();
        e1.Display();

        Console.ReadKey();
    }
}

```

When should we define a parameterized constructor in a class?

If we want to initialize the object dynamically with the user-given values or if we want to initialize each instance of a class with a different set of values then we need to use the Parameterized Constructor in C#. The advantage is that we can initialize each instance with different values.

What is Parameterized Constructor?

If a constructor method is defined with parameters, we call it a Parameterized Constructor in C#, and these constructors are defined by the programmers only but never can be defined implicitly. So, in simple words, we can say that the developer-given constructor with parameters is called Parameterized Constructor in C#.

```

static void Main(string[] args)
{
    ParameterizedConstructor obj = new ParameterizedConstructor(10);
    Console.ReadKey();
}

public class ParameterizedConstructor
{
    public ParameterizedConstructor(int i)
    {
        Console.WriteLine($"Parameterized Constructor is Called : {i}");
    }
}

```

The diagram illustrates a call from a Main() method to a ParameterizedConstructor() method. A red box highlights the Main() method, and another red box highlights the ParameterizedConstructor() method. A red arrow points from the 'i' parameter in the ParameterizedConstructor() method's definition to the corresponding argument in the Main() method's call to the constructor.

image_92.png

```
using System;
namespace ConstructorDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            ParameterizedConstructor obj = new
ParameterizedConstructor(10);
            Console.ReadKey();
        }
    }

    public class ParameterizedConstructor
    {
        public ParameterizedConstructor(int i)
        {
            Console.WriteLine($"Parameterized Constructor is Called:
{i}");
        }
    }
}
```

Output: Parameterized Constructor is Called: 10

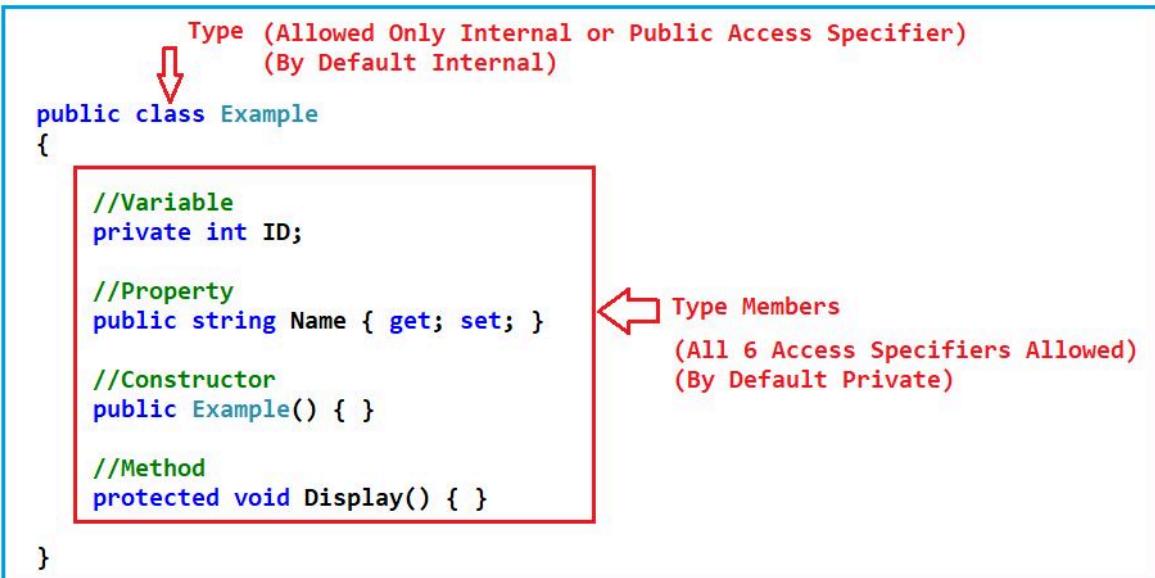
Access Specifiers

Every keyword that we use such as private, public, protected, virtual, sealed, partial, abstract, static, base, etc. is called Modifiers. *Access Specifiers are special kinds of modifiers using which we can define the scope of a type and its members.*

So, in simple words, we can say that the Access Specifiers are used to define the **scope of the type** (**Class, Interface, Structs, Delegate, Enum, etc.**) as well as the **scope of their members** (**Variables, Properties, Constructors, and Methods**). Scope means accessibility or visibility that is who can access them and who cannot access them are defined by the Access Specifiers. See, I have a class with a set of members, who can consume these members, and who cannot consume these members are defined by the access specifiers.

C# supports **6 types of access specifiers**. They are as follows

- Private
- Public
- Protected
- Internal
- Protected Internal
- Private Protected (C# Version 7.2 onwards)



image_93.png

Before understanding Access Specifier, let us first understand what are Types and Type Members in C#. Please have a look at the below diagram. Here, Example (which is created by using the class keyword) is a Type, and Variable ID, Property Name, Constructor Example, and Method Display are type members.

So, in general classes, structs, enums, interfaces, and delegates are called types, and variables, properties, constructors, methods, etc. that normally reside within a type are called type members.

Private Access Specifier/Modifier

When we declare a type member (variable, property, method, constructor, etc) as private, then we can access that member with the class only

Let us understand Private Members with an example. Now, go to the class library project and modify class1.cs class file as follows. As you can see, here we have created three classes and in the AssemblyOneClass1 we have created one private variable and then tried to access the private variable within the same class (AssemblyOneClass1), from the derived class (AssemblyOneClass2), and from the non-derived class (AssemblyOneClass3). And all these classes are within the same assembly only.

```

using System;
namespace AssemblyOne
{

```

```

public class AssemblyOneClass1
{
    private int Id;
    public void Display1()
    {
        //Private Member Accessible with the Containing Type only
        //Where they are created, they are available only within
that type
        Console.WriteLine(Id);
    }
}
public class AssemblyOneClass2 : AssemblyOneClass1
{
    public void Display2()
    {
        //You cannot access the Private Member from the Derived
Class
        //Within the Same Assembly
        Console.WriteLine(Id); //Compile Time Error
    }
}

public class AssemblyOneClass3
{
    public void Dispplay3()
    {
        //You cannot access the Private Member from the Non-Derived
Classes
        //Within the Same Assembly
        AssemblyOneClass1 obj = new AssemblyOneClass1();
        Console.WriteLine(obj.Id); //Compile Time Error
    }
}

```

When you try to build the above code, you will get some compilation errors as shown in the below image. Here, you can see, it is clearly saying that you cannot access

'AssemblyOneClass1.Id' due to its protection level

Code	Description
CS0122	'AssemblyOneClass1.Id' is inaccessible due to its protection level
CS0122	'AssemblyOneClass1.Id' is inaccessible due to its protection level

image_94.png

These errors make sense that you cannot access the private members from derived and non-derived classes from different assemblies also.

⚠ So, the scope of the **private member in C#.NET is as follows:**

- With the Class: YES
- Derived Class in Same Assembly: NO
- Non-Derived Class in Same Assembly: NO
- Derived Class in Other Assemblies: NO
- Non-Derived Class in Other Assemblies: NO

Public Access Specifier/Modifier

When we declare a type member (variable, property, method, constructor, etc) as public, then we can access that member from anywhere. That means there is no restriction for public members.

```
using System;
namespace AssemblyOne
{
    public class AssemblyOneClass1
    {
        public int Id;
        public void Display1()
        {
            //Public Members Accessible with the Containing Type
        }
}
```

```

        //Where they are created
        Console.WriteLine(Id);
    }
}

public class AssemblyOneClass2 : AssemblyOneClass1
{
    public void Display2()
    {
        //We Can access public Members from Derived Class
        //Within the Same Assembly
        Console.WriteLine(Id); //No-Compile Time Error
    }
}

public class AssemblyOneClass3
{
    public void Dispplay3()
    {
        //We Can access public Members from Non-Derived Classes
        //Within the Same Assembly
        AssemblyOneClass1 obj = new AssemblyOneClass1();
        Console.WriteLine(obj.Id); //No-Compile Time Error
    }
}

```

⚠ So, the scope of the **public member** in C#.NET is as follows:

- With the Class: **YES**
- Derived Class in Same Assembly: **YES**
- Non-Derived Class in Same Assembly: **YES**
- Derived Class in Other Assemblies: **YES**
- Non-Derived Class in Other Assemblies: **YES**

Protected Access Specifier/Modifier

Protected Members in C# are available within the containing type as well as to the types that are derived from the containing type. That means protected members are available within the parent class (i.e. the containing type) as well as to the child/derived classes (classes derived from the containing type).

```
using System;
namespace AssemblyOne
{
    public class AssemblyOneClass1
    {
        protected int Id;
        public void Display1()
        {
            //protected Members Accessible with the Containing Type
            //Where they are created
            Console.WriteLine(Id);
        }
    }
    public class AssemblyOneClass2 : AssemblyOneClass1
    {
        public void Display2()
        {
            //We Can access protected Member from Derived Classes
            //Within the Same Assembly
            Console.WriteLine(Id); //No-Compile Time Error
        }
    }
}

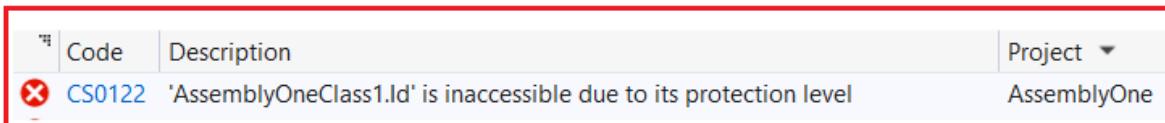
public class AssemblyOneClass3
{
    public void Dispplay3()
    {
        //We Cannot access protected Member from Non-Derived Classes
        //Within the Same Assembly
        AssemblyOneClass1 obj = new AssemblyOneClass1();
```

```

        Console.WriteLine(obj.Id); //Compile Time Error
    }
}
}

```

Output:



image_95.png

⚠ So, the scope of the `protected` members in C#.NET is as follows:

- With the Class: YES
- Derived Class in Same Assembly: YES
- Non-Derived Class in Same Assembly: NO
- Derived Class in Other Assemblies: YES
- Non-Derived Class in Other Assemblies: NO

Internal Access Specifier/Modifier

Whenever a member is declared with Internal Access Specifier in C#, then it is available anywhere within the containing assembly. It's a compile-time error to access an internal member from outside the containing assembly.

```

using System;
namespace AssemblyOne
{
    public class AssemblyOneClass1
    {
        internal int Id;
        public void Display1()
    }
}

```

```

{
    //internal Members Accessible with the Containing Type
    //Where they are created
    Console.WriteLine(Id);
}
}

public class AssemblyOneClass2 : AssemblyOneClass1
{
    public void Display2()
    {
        //We can access internal Members from Derived Classes
        //Within the Same Assembly
        Console.WriteLine(Id); //No-Compile Time Error
    }
}

public class AssemblyOneClass3
{
    public void Dispplay3()
    {
        //We can access internal Members from Non-Derived Classes
        //Within the Same Assembly
        AssemblyOneClass1 obj = new AssemblyOneClass1();
        Console.WriteLine(obj.Id); //No-Compile Time Error
    }
}
}

```

A So, the scope of the internal members in C#.NET is as follows:

- With the Class: **YES**
- Derived Class in Same Assembly: **YES**
- Non-Derived Class in Same Assembly: **YES**
- Derived Class in Other Assemblies: **NO**

- Non-Derived Class in Other Assemblies: NO

Protected Internal Access Specifier/Modifier

Protected Internal Members in C# can be accessed anywhere within the same assembly i.e. in which it is declared or from within a derived class from another assembly. So, we can think, it is a combination of Protected and Internal access specifiers. If you understood the Protected and Internal access specifiers, then this should be very easy to follow. Protected means, members can be accessed within the derived classes, and Internal means within the same assembly.

Let us understand this Protected Internal Access Specifier in C# with an example. Now, modify class1.cs class file as follows: Here, we are modifying the variable from internal to protected internal. Here, you can observe while accessing the protected internal member from the containing type, from the derived classes, and from the non-derived class within the same assembly, we are not getting any compilation error.

```
using System;
namespace AssemblyOne
{
    public class AssemblyOneClass1
    {
        protected internal int Id;
        public void Display1()
        {
            //protected internal Members Accessible with the Containing
            Type
            //Where they are created
            Console.WriteLine(Id);
        }
    }
    public class AssemblyOneClass2 : AssemblyOneClass1
    {
        public void Display2()
        {
            //We can access protected internal Member from Derived
            Classes
        }
    }
}
```

```

        //Within the Same Assembly
        Console.WriteLine(Id); //No-Compile Time Error
    }

}

public class AssemblyOneClass3
{
    public void Dispplay3()
    {
        //We can access protected internal Member from Non-Derived
        Classes
        //Within the Same Assembly
        AssemblyOneClass1 obj = new AssemblyOneClass1();
        Console.WriteLine(obj.Id); //No-Compile Time Error
    }
}

```

Now, let us try to access the protected internal members from a different assembly. Modify the Program.cs class file as follows. From other assemblies, you can access the protected internal member from the derived classes, but you cannot access from the non-derived classes.

```

using AssemblyOne;
using System;
namespace AccessSpecifierDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
        }
    }

    public class AnotherAssemblyClass1 : AssemblyOneClass1
    {
        public void Display4()
    }
}

```

```

    {
        //We can access the protected internal Members from Derived
        Classes
        //from Other Assemblies
        Console.WriteLine(Id); //No-Compile Time Error
    }
}

public class AnotherAssemblyClass2
{
    public void Dispplay3()
    {
        //We cannot access protected internal Members from Non-
        Derived Classes
        //from Other Assemblies
        AssemblyOneClass1 obj = new AssemblyOneClass1();
        Console.WriteLine(obj.Id); //Compile Time Error
    }
}

```

Output:

	Code	Description	Project
CS0122	'AssemblyOneClass1.Id' is inaccessible due to its protection level		AccessSpecifierDemo

image_96.png

⚠ So, the scope of the **protected internal** members in C#.NET is as follows:

- With the Class: **YES**
- Derived Class in Same Assembly: **YES**
- Non-Derived Class in Same Assembly: **YES**
- Derived Class in Other Assemblies: **YES**

- Non-Derived Class in Other Assemblies: NO

⚠ Note: Here, I have shown the example by using a variable, but the same is applicable to other members of a class like properties, methods, and constructors. The following table shows the summary of all the access specifiers with the type members.

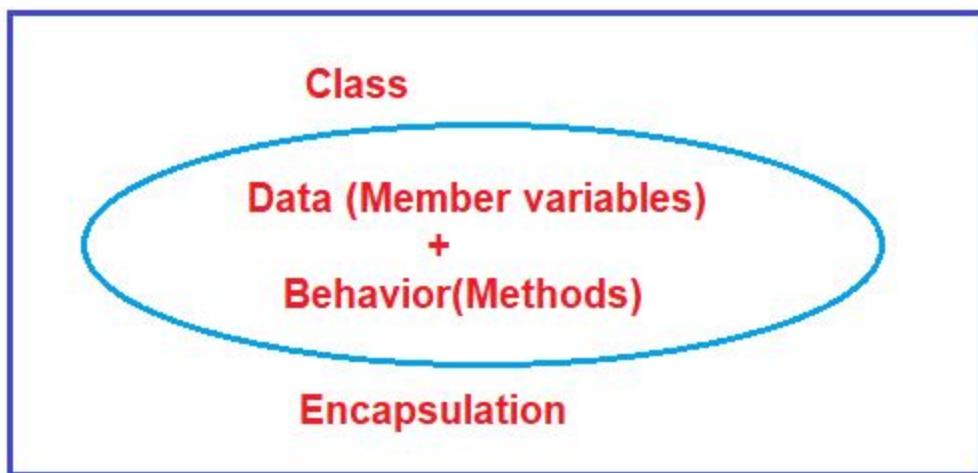
Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗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

image_97.png

OOP Principles: Encapsulation

Encapsulation Hides the internal state and functionality of an object and only allows access through a public set of functions. Let us simplify the above definition as follows:

The process of binding or grouping the State (i.e., Data Members) and Behaviour (i.e., Member Functions) together into a single unit (i.e., class, interface, struct, etc.) is called Encapsulation in C#. The Encapsulation Principle ensures that the state and behavior of a unit (i.e., class, interface, struct, etc.) cannot be accessed directly from other units (i.e., class, interface, struct, etc.).



image_98.png

Example to Understand Encapsulation in C#:

Every class, interface, struct, enum, etc. that we created is an example of encapsulation, so let's create a class called Bank as follows to understand the encapsulation:

```
namespace BankExample
{
    public class Bank
    {
        public long AccountNumber;
        public string Name;
        public int Balance;

        public Bank(long _accountNumber, string _name, int _balance)
    }
}
```

```

{
    AccountNumber = _accountNumber;
    Name = _name;
    Balance = _balance;
}

public void Getbalance()
{
    Console.WriteLine($"Account Number: {AccountNumber}, Name: {Name}, Balance: {Balance}");
}

public int WithdrawAmount(int deduction)
{
    return Balance -= deduction;
}

public int Deposit(int MoneyDeposited)
{
    return Balance += MoneyDeposited;
}

```

If other classes want to access these details, they need to create the object of the Bank class to access its data and behavior, as shown in the code below.

```

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

            Bank equity = new Bank(1234567890, "John Doe", 5000);
            equity.Getbalance();
            equity.Deposit(2000);
            equity.Getbalance();
        }
    }
}
```

```
        equity.WithdrawAmount(1500);
        equity.Getbalance();
        Console.ReadKey();
    }
}
}
```

What is Data Hiding?

Data hiding or Information Hiding is a Process in which we hide internal data from outside the world. The purpose of data hiding is to protect the data from misuse by the outside world.



Data Encapsulation is also called Data Hiding because by using this principle, we can hide the internal data from outside the class.

How can we Implement Data Hiding

- By declaring the variables as private (to restrict their direct access from outside the class)
- By defining one pair of public setter and getter methods or properties to access private variables from outside the class

RECAP

- **public:** The public members can be accessed by any other code in the same assembly or another assembly that references it.
- **private:** The private members can be accessed only by code in the same class.
- **protected:** The protected Members in C# are available within the same class as well as to the classes that are derived from that class.
- **internal:** The internal members can be accessed by any code in the same assembly but not from another assembly.

- **protected internal:** The protected internal members can be accessed by any code in the assembly in which it's declared or from within a derived class in another assembly.
- **private protected:** The private protected members can be accessed by types derived from the class that is declared within its containing assembly.

Implementing Data Encapsulation/Hiding in C# using Setter and Getter Methods:

```

using System;
namespace EncapsulationDemo
{
    public class Bank
    {
        //Hiding class data by declaring the variable as private
        private double balance;

        //Creating public Setter and Getter methods

        //Public Getter Method
        //This method is used to return the data stored in the balance
variable
        public double GetBalance()
        {
            //add validation logic if needed
            return balance;
        }

        //Public Setter Method
        //This method is used to stored the data in the balance
variable
        public void SetBalance(double balance)
        {
            // add validation logic to check whether data is correct or
not
            this.balance = balance;
        }
    }
}

```

```

class Program
{
    public static void Main()
    {
        Bank bank = new Bank();
        //You cannot access the Private Variable
        //bank.balance; //Compile Time Error

        //You can access the private variable via public setter and
        //getter methods
        bank.SetBalance(500);
        Console.WriteLine(bank.GetBalance());
        Console.ReadKey();
    }
}

```

Implementing Data Encapsulation/Hiding in C# using Properties:

The Properties are a new language feature introduced in C#. Properties in C# help in protecting a field or variable of a class by reading and writing the values to it. The first approach, i.e., **setter** and **getter** itself, is good, but Data Encapsulation in C# can be accomplished much smoother with properties.

Let us understand how to implement Data Encapsulation or Data Hiding in C# using properties with an example

```

using System;
namespace EncapsulationDemo
{
    public class Bank
    {
        private double _Amount;
        public double Amount
        {
            get
            {
                return _Amount;
            }
        }
    }
}

```

```

    }
    set
    {
        // Validate the value before storing it in the _Amount
variable
        if (value < 0)
        {
            throw new Exception("Please Pass a Positive Value");
        }
        else
        {
            _Amount = value;
        }
    }
}
class Program
{
    public static void Main()
    {
        try
        {
            Bank bank = new Bank();
            //We cannot access the _Amount Variable directly
            //bank._Amount = 50; //Compile Time Error
            //Console.WriteLine(bank._Amount); //Compile Time Error

            //Setting Positive Value using public Amount Property
            bank.Amount= 10;

            //Setting the Value using public Amount Property
            Console.WriteLine(bank.Amount);

            //Setting Negative Value
            bank.Amount = -150;
            Console.WriteLine(bank.Amount);
        }
        catch (Exception ex)

```

```
        {
            Console.WriteLine(ex.Message);
        }

        Console.ReadKey();
    }
}
```

Output :

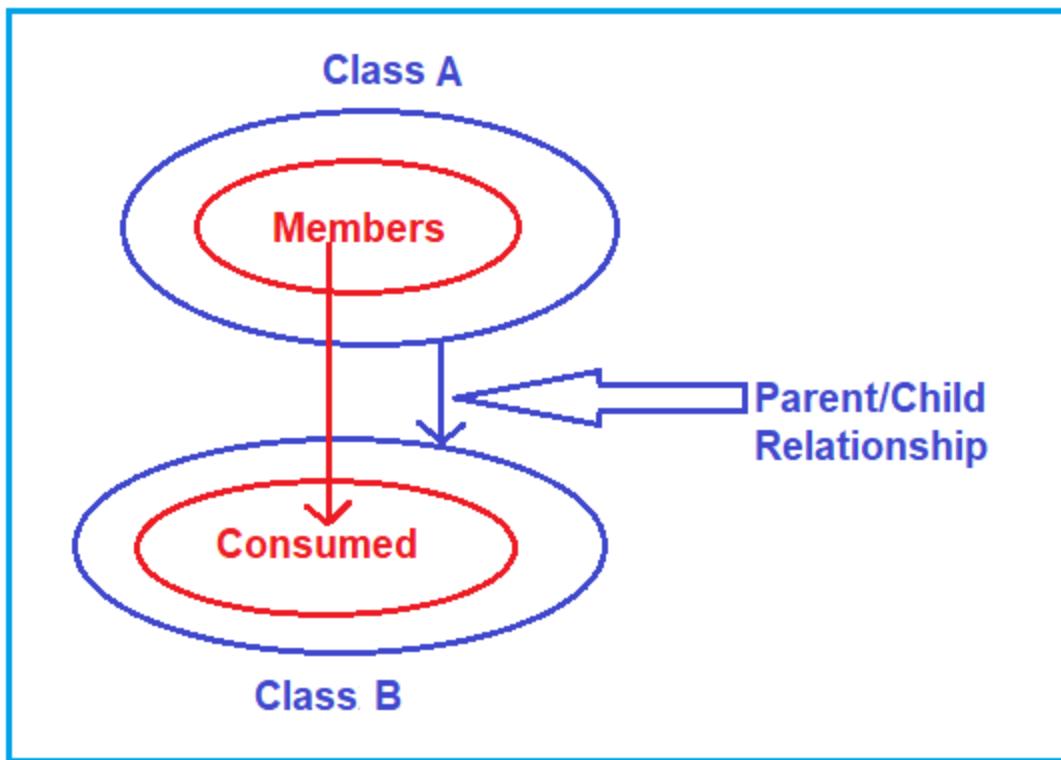
```
10
Please Pass a Positive Value
```

Advantages of Encapsulation in C#:

- Data protection: You can validate the data before storing it in the variable.
- Achieving Data Hiding: The user will have no idea about the inner implementation of the class.
- Security: The encapsulation Principle helps to secure our code since it ensures that other units(classes, interfaces, etc) can not access the data directly.
- Flexibility: The encapsulation Principle in C# makes our code more flexible, allowing the programmer to easily change or update the code.
- Control: The encapsulation Principle gives more control over the data stored in the variables. For example, we can control the data by validating whether the data is good enough to store in the variable.

OOP Principles: Inheritance

Inheritance in C# is a mechanism of consuming the members that are defined in one class from another class. See, we are aware that a class is a collection of members. The members defined in one class can be consumed by another class by establishing a parent/child relationship between the classes.



image_99.png

So, Inheritance in C# is a mechanism of consuming the members of one class in another class by establishing a parent/child relationship between the classes, which provides reusability.

How to Implement Inheritance

To Implement Inheritance in C#, we need to establish a Parent/Child relationship between classes. Let us understand how to establish a Parent/Child relationship in C#. Suppose we have a class called A with a set of members. And we have another class B, and we want this class B to be inherited from class A. The following code shows how to establish the Parent-Child relationship between Class A and Class B.

```

class A
{
    //Members
}
class B : A ← Establishing Parent/Child Relationship
{
    //Consuming the Members of A from here
}

```

image_100.png

So, this is the basic process for establishing a Parent/Child relationship in C#. Now, let us see the basic syntax to establish a Parent/Child relationship between classes. The syntax is given below.

[<modifiers>] class <child class> : <parent class>

- ⚠** In Inheritance, the Child class can consume members of its Parent class as if it is the owner of those members (except private members of the parent).

Example to Understand Inheritance

Let us see a simple example to understand Inheritance in C#. Let us create a class with two methods, as shown below.

```

class A
{
    public void Method1()
    {
        Console.WriteLine("Method 1");
    }
    public void Method2()
    {
        Console.WriteLine("Method 2");
    }
}

```

```
    }  
}
```

Here, we have created class A with two public methods, i.e., Method1 and Method2.

Now, I want the same two methods in another class, i.e., class B. One way to do this is to copy the above two methods and paste them into class B as follows.

```
class B  
{  
    public void Method1()  
    {  
        Console.WriteLine("Method 1");  
    }  
    public void Method2()  
    {  
        Console.WriteLine("Method 2");  
    }  
}
```

If we do this, then it is not code re-usability. It is code rewriting that affects the size of the application. So, without rewriting, what we need to do is, we need to perform inheritance here as follows. Here, class B is inherited from class A, and hence, inside the Main method, we create an instance of class B and invoke the methods which are defined in Class A.

```
class B : A  
{  
    static void Main()  
    {  
        B obj = new B();  
        obj.Method1();  
        obj.Method2();  
    }  
}
```

Once you perform the Inheritance, class B can take the two members defined in class A.

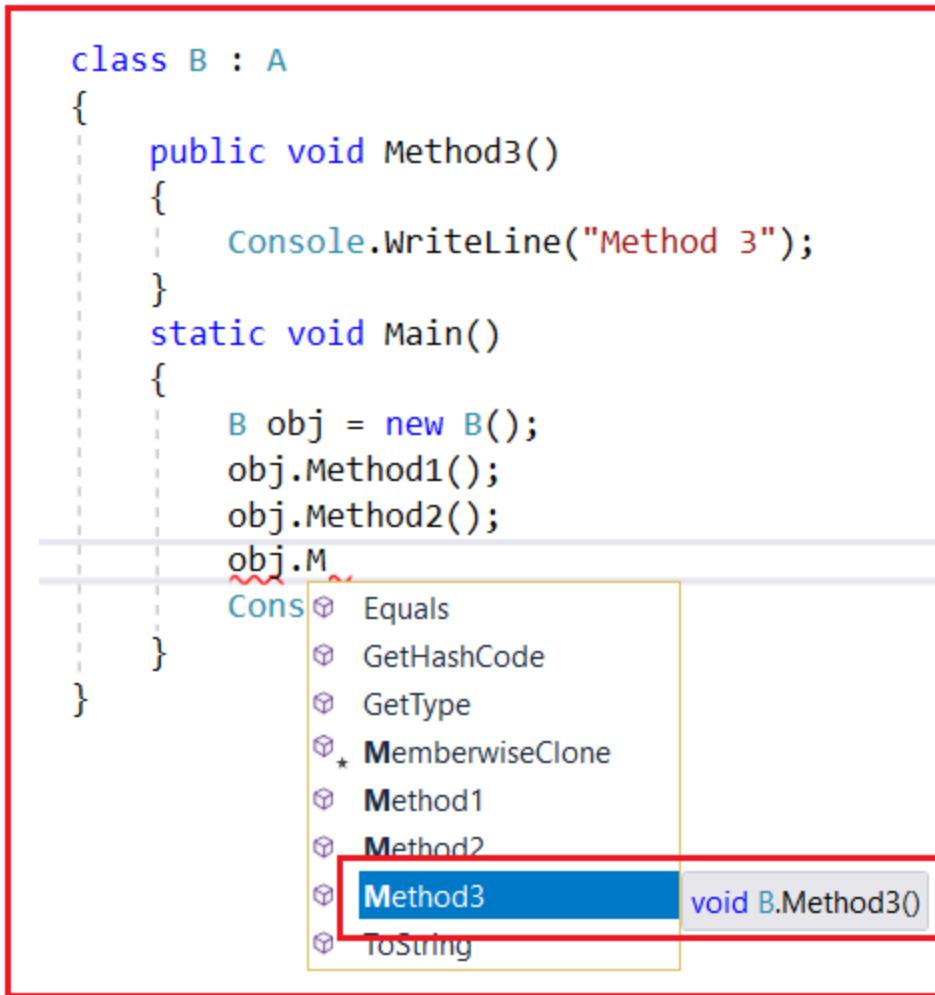
Why? Because all the properties of a Parent belong to Children. Here, class A is the Parent/Super/Base class, and Class B is the Child/Sub/Derived class.

```
using System;
namespace InheritanceDemo
{
    class A
    {
        public void Method1()
        {
            Console.WriteLine("Method 1");
        }
        public void Method2()
        {
            Console.WriteLine("Method 2");
        }
    }
    class B : A
    {
        static void Main()
        {
            B obj = new B();
            obj.Method1();
            obj.Method2();
            Console.ReadKey();
        }
    }
}
```

Output :

```
Method 1
Method 2
```

Now, let us add a new method, i.e., Method3 in Class B, as follows. Inside the Main method, if you see the method description, it shows that the method belongs to class B.



image_101.png

The complete example is given below.

```
using System;
namespace InheritanceDemo
{
    class A
    {
        public void Method1()
        {
            Console.WriteLine("Method 1");
        }
        public void Method2()
        {
            Console.WriteLine("Method 2");
        }
    }
}
```

```

    }
}

class B : A
{
    public void Method3()
    {
        Console.WriteLine("Method 3");
    }

    static void Main()
    {
        B obj = new B();
        obj.Method1();
        obj.Method2();
        obj.Method3();
        Console.ReadKey();
    }
}
}

```

Implicit & Explicit Constructor

Right now, in our example, both class A and class B have implicit constructors. Yes, every class in C# contains an implicit constructor if as a developer we did not define any constructor explicitly. We already learned it in our constructor section.

If a constructor is defined implicitly, then it is a public constructor. In our example, class B can access the class A implicit constructor as it is public. Now, let us define one explicit constructor in Class A as follows.

```

class A
{
    public A()
    {
        Console.WriteLine("Class A Constructor is Called");
    }

    public void Method1()
    {
        Console.WriteLine("Method 1");
    }
}

```

```
    }
    public void Method2()
    {
        Console.WriteLine("Method 2");
    }
}
```

Output :

```
Class A Constructor is Called
Method 1
Method 2
```

- ⚠** When you execute the code, the class A constructor is first called, and that is what you can see in the output. Why? This is because whenever the child class instance is created, the child class constructor will implicitly call its parent class constructors. This is a rule.

But remember, if you are defining an explicit constructor, if you make that constructor private, and if you don't provide an access specifier, then by default, the class member's access specifier is private in C#. For example, modify class A as follows. As you can see, we have removed the access specifier from the constructor which makes it private.

```
class A
{
    A()
    {
        Console.WriteLine("Class A Constructor is Called");
    }
    public void Method1()
    {
        Console.WriteLine("Method 1");
    }
    public void Method2()
    {
        Console.WriteLine("Method 2");
    }
}
```

```
    }  
}
```

As you can see in the code, the Class A constructor is private, so it is not accessible to Class B. Now, if you try to run the code, you will get the following compile-time error as shown in the below image which tells Class A Constructor is inaccessible due to its protection level.

Code	Description
CS0122	'A.A()' is inaccessible due to its protection level

image_102.png

We are getting the above error because when we create an instance of the child class, the child class constructor will implicitly call its parent class constructors. Right now, the Class B constructor trying to call the Class A constructor, which is not accessible because that constructor is private.

How to pass dynamic value to Parent class constructor

Let us see this with an example. In the below example, the child class, i.e., class B constructor, takes one parameter and passes that parameter value to the parent class, i.e., Class A constructor. And when we are creating the instance of Class B, we need to pass the parameter value.

```
using System;  
namespace InheritanceDemo  
{  
    class A  
    {  
        public A(int number)  
        {  
            Console.WriteLine($"Class A Constructor is Called :  
{number}");  
        }  
        public void Method1()  
        {  
            Console.WriteLine("Method 1");  
        }  
    }  
}
```

```

    }
    public void Method2()
    {
        Console.WriteLine("Method 2");
    }
}

class B : A
{
    public B(int num) : base(num)
    {
        Console.WriteLine("Class B Constructor is Called");
    }
    public void Method3()
    {
        Console.WriteLine("Method 3");
    }
    static void Main()
    {
        B obj1 = new B(10);
        B obj2 = new B(20);
        B obj3 = new B(30);
        Console.ReadKey();
    }
}
}

```

Output:

```

Class A Constructor is Called : 10
Class B Constructor is Called
Class A Constructor is Called : 20
Class B Constructor is Called
Class A Constructor is Called : 30
Class B Constructor is Called

```

image_103.png

So, in the above example, when we are creating the instance, we are passing the value. The value first reaches the child class constructor, and the child class constructor passes

the same value to the parent class constructor. If you want, then you can also use the same value in the child class.

Types of Inheritance

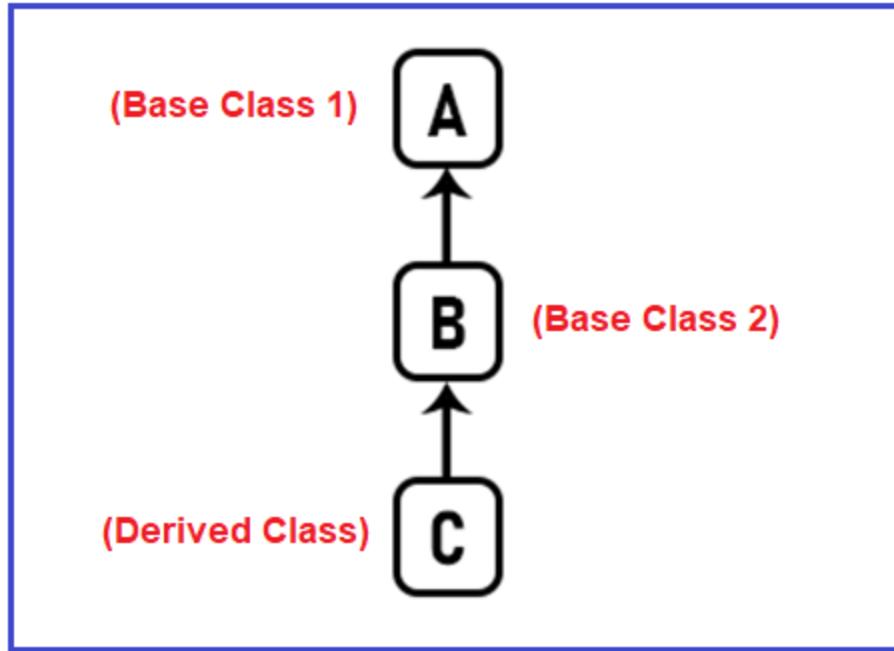
What these types of Inheritance will tell us is the number of parent classes a child class has or the number of child classes a parent class has. According to C++, why I am telling about C++ is because Object-Oriented Programming came into the picture from C++ only, there are five different types of Inheritances.

- Single Inheritance
- Multi-Level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance
- Multiple Inheritance

If you look at Single, Multilevel, and Hierarchical inheritances, they look very similar. At any point in time, they are having a single immediate parent class. But, if you look at Multiple and Hybrid, they are having more than one immediate parent class for a child class. So, we can broadly classify the above five categories of inheritances into two types based on immediate parent class as follows:

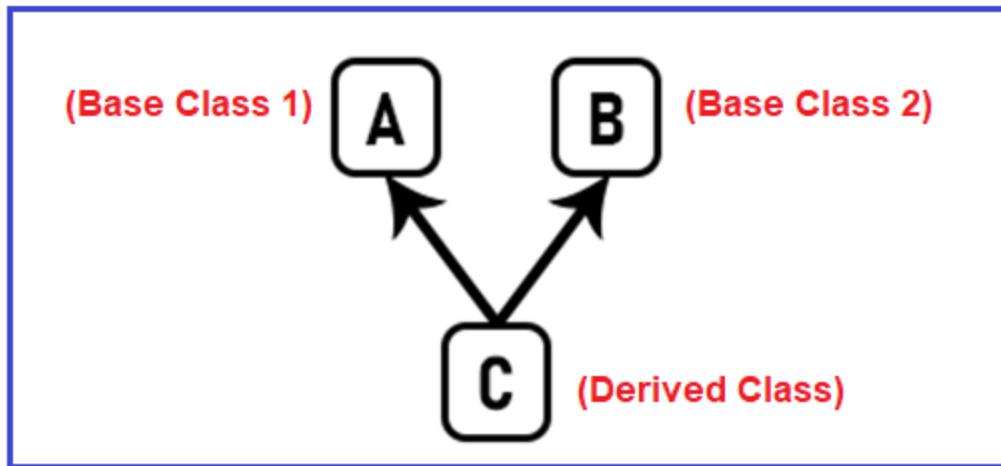
- Single Inheritance (Single, Multilevel, and Hierarchical)
- Multiple Inheritance (Multiple and Hybrid)

Single Inheritance: If at all a class has 1 immediate parent class to it, we call it Single Inheritance in C#. For a better understanding, please have a look at the below diagram. See, how many immediate parent class C has? 1 i.e. B, and how many immediate parent class B has? 1 i.e. A. So, for class C, the immediate Parent is class B and for class B, the immediate Parent is class A.



image_104.png

Multiple Inheritance in C#: If a class has more than 1 immediate parent class to it, then we call it Multiple Inheritance in C#. For a better understanding, please have a look at the below diagram. See, class C has more than one immediate Parent class i.e. A and B and hence it is Multiple Inheritance.



image_105.png

So, the point that you need to remember is how many immediate Parent classes a child class has. If one immediate Parent class, call it Single Inheritance, and if more than one immediate Parent class, call it is multiple inheritance. So, there should not be any

confusion between 5 different types of inheritances, simply we have two types of inheritances.

Example to Understand Single Inheritance:

```
using System;
namespace InheritanceDemo
{
    public class Program
    {
        static void Main()
        {
            // Creating object of Child class and
            // invoke the methods of Parent and Child classes
            Cuboid obj = new Cuboid(2, 4, 6);
            Console.WriteLine($"Volume is : {obj.Volume()}");
            Console.WriteLine($"Area is : {obj.Area()}");
            Console.WriteLine($"Perimeter is : {obj.Perimeter()}");
            Console.ReadKey();
        }
    }
    //Parent class
    public class Rectangle
    {
        public int length;
        public int breadth;
        public int Area()
        {
            return length * breadth;
        }
        public int Perimeter()
        {
            return 2 * (length + breadth);
        }
    }

    //Child Class
    class Cuboid : Rectangle
```

```

{
    public int height;
    public Cuboid(int l, int b, int h)
    {
        length = l;
        breadth = b;
        height = h;
    }
    public int Volume()
    {
        return length * breadth * height;
    }
}

```

Output:

```

Volume is : 48
Area is : 8
Perimeter is : 12

```

image_106.png

Example to Understand Multiple Inheritance:

```

using System;
namespace InheritanceDemo
{
    public class Program
    {
        static void Main()
        {
            // Creating object of Child class and
            // invoke the methods of Parent classes and Child class
            SmartPhone obj = new SmartPhone(); ;
            obj.GetPhoneModel();
            obj.GetCameraDetails();
            obj.GetDetails();
        }
    }
}

```

```

        Console.ReadKey();
    }
}

//Parent Class 1
class Phone
{
    public void GetPhoneModel()
    {
        Console.WriteLine("Redmi Note 5 Pro");
    }
}

//Parent class2
class Camera
{
    public void GetCameraDetails()
    {
        Console.WriteLine("24 Mega Pixel Camera");
    }
}

//Child Class derived from more than one Parent class
class SmartPhone : Phone, Camera
{
    public void GetDetails()
    {
        Console.WriteLine("Its a RedMi Smart Phone");
    }
}

```

Output: Compile Time Error

- ⚠** Handling the complexity caused due to multiple inheritances is very complex. Hence it was not supported in dot net with class and it can be done with interfaces. So, in our Multiple Inheritance articles, we will discuss this concept in detail.

How to use Inheritance in Application Development

Inheritance is something that comes into the picture, not in the middle of a project or middle of application development. This can also come in the middle of the project development but generally when we start application development, in the initial stages only we plan inheritance and implement it in our project.

What is an Entity?

In DBMS terminology **what is an Entity?** An Entity is something that is associated with a set of attributes. An Entity can be a living or non-living object. But anything that is associated with a set of attributes is called Entity.

Remember, when we are going to develop an application, our application mainly deals with these Entities. Suppose, you are developing an application for a Bank. So, the Entity associated with the bank is a customer. A customer is an Entity. You are developing an application for a school; the Student will be the Entity. Suppose, you are developing an application for a business, then Employee is an entity. So, every application that we develop is associated with a set of entities.

OOP Principles: Abstraction

The process of representing the essential features without including the background details is called Abstraction. In simple words, we can say that it is a process of defining a class by providing the necessary details to call the object operations (i.e., methods) by hiding its implementation details. It is called abstraction in C#.

Real-Time Example of Abstraction

We all use ATM machines for cash withdrawals, money transfers, retrieving min-statements, etc. in our daily lives. But we don't know internally what things are happening inside an ATM machine when we insert an ATM card for performing different kinds of operations. Information like where the server is, where the database server is, what programming language they use to write the logic, how they are validating the data, how they are implementing logic for various kinds of operations, and what SQL statements get executed on the database when we perform any operations, all these things are hidden from us. What they provide as part of the ATM machine is services (cash withdrawal, money transfer, retrieving min-statement, etc), but how these services are implemented is abstracted to us.

Example to Understand Abstraction Principle

Now, we are going to develop one application to implement the Banking functionality. First, we will develop the application without following the Abstraction Principle, and then we will understand the problems. Then, we will see what are the different mechanisms to implement the abstraction principle in C#. So, what we will do is we will create two classes. One class is for SBI Bank, and another class is for AXIX Bank. As part of each class, we are going to provide 5 services, which are as follows:

1. ValidateCard
2. WithdrawMoney
3. CheckBalance
4. BankTransfer
5. MiniStatement

Then, from the Main method, we will create the instances of each class and will invoke the respective services, i.e., respective methods. Here, you can consider the Main method is the user who will use the services provided by the Bank classes.

```
using System;
namespace GarbageCollectionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Transaction doing SBI Bank");
            SBI sbi = new SBI();
            sbi.ValidateCard();
            sbi.WithdrawMoney();
            sbi.CheckBalanace();
            sbi.BankTransfer();
            sbi.MiniStatement();

            Console.WriteLine("\nTransaction doing AXIX Bank");
            AXIX AXIX = new AXIX();
            AXIX.ValidateCard();
            AXIX.WithdrawMoney();
            AXIX.CheckBalanace();
            AXIX.BankTransfer();
            AXIX.MiniStatement();

            Console.Read();
        }
    }

    public class SBI
    {
        public void BankTransfer()
        {
            Console.WriteLine("SBI Bank Bank Transfer");
        }
    }
}
```

```
public void CheckBalanace()
{
    Console.WriteLine("SBI Bank Check Balanace");
}

public void MiniStatement()
{
    Console.WriteLine("SBI Bank Mini Statement");
}

public void ValidateCard()
{
    Console.WriteLine("SBI Bank Validate Card");
}

public void WithdrawMoney()
{
    Console.WriteLine("SBI Bank Withdraw Money");
}

}

public class AXIX
{
    public void BankTransfer()
    {
        Console.WriteLine("AXIX Bank Bank Transfer");
    }

    public void CheckBalanace()
    {
        Console.WriteLine("AXIX Bank Check Balanace");
    }

    public void MiniStatement()
    {
        Console.WriteLine("AXIX Bank Mini Statement");
    }
}
```

```

public void ValidateCard()
{
    Console.WriteLine("AXIX Bank Validate Card");
}

public void WithdrawMoney()
{
    Console.WriteLine("AXIX Bank Withdraw Money");
}
}

```

That is fine. We are getting output as expected. Then what is the problem with the above implementation?

- ⚠** The problem is the user of our application accesses the SBI and AXIX classes directly. Directly means they can go to the class definition and see the implementation details of the methods. That is, the user will come to know how the services or methods are implemented. This might cause security issues. We should not expose our implementation details to the outside.

How to Implement Abstraction Principle

In C#, we can implement the abstraction OOPs principle in two ways. They are as follows:

- Using **Interface**
- Using **Abstract Classes and Abstract Methods**

What are Interfaces, and what are Abstract Methods and Abstract Classes that we will discuss in detail in our upcoming article? But for now, you just need to understand one thing: both interface and abstract classes and abstract methods provide some mechanism to hide the implementation details by only exposing the services. The user only knows what are services or methods available, but the user will not know how these services or methods are implemented. Let us see this with examples.

Example to Implement Abstraction Principle using Interface

In the below example, I am using an interface to achieve the abstraction principle in C#. Using the interface, we can achieve 100% abstraction. Now, the user will only know the services that are defined in the interface, but how the services are implemented, the user will never know. This is how we can implement abstraction in C# by hiding the implementation details from the user. Here, the user will only know about IBank, but the user will not know about the SBI and AXIX Classes.

```
using System;
namespace GarbageCollectionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Transaction doing SBI Bank");
            IBank sbi = BankFactory.GetBankObject("SBI");
            sbi.ValidateCard();
            sbi.WithdrawMoney();
            sbi.CheckBalanace();
            sbi.BankTransfer();
            sbi.MiniStatement();

            Console.WriteLine("\nTransaction doing AXIX Bank");
            IBank AXIX = BankFactory.GetBankObject("AXIX");
            AXIX.ValidateCard();
            AXIX.WithdrawMoney();
            AXIX.CheckBalanace();
            AXIX.BankTransfer();
            AXIX.MiniStatement();

            Console.Read();
        }
    }

    public interface IBank
    {
```

```

        void ValidateCard();
        void WithdrawMoney();
        void CheckBalanace();
        void BankTransfer();
        void MiniStatement();
    }

public class BankFactory
{
    public static IBank GetBankObject(string bankType)
    {
        IBank BankObject = null;
        if (bankType == "SBI")
        {
            BankObject = new SBI();
        }
        else if (bankType == "AXIX")
        {
            BankObject = new AXIX();
        }
        return BankObject;
    }
}

public class SBI : IBank
{
    public void BankTransfer()
    {
        Console.WriteLine("SBI Bank Bank Transfer");
    }

    public void CheckBalanace()
    {
        Console.WriteLine("SBI Bank Check Balanace");
    }

    public void MiniStatement()
    {

```

```
        Console.WriteLine("SBI Bank Mini Statement");
    }

    public void ValidateCard()
    {
        Console.WriteLine("SBI Bank Validate Card");
    }

    public void WithdrawMoney()
    {
        Console.WriteLine("SBI Bank Withdraw Money");
    }
}

public class AXIX : IBank
{
    public void BankTransfer()
    {
        Console.WriteLine("AXIX Bank Bank Transfer");
    }

    public void CheckBalanace()
    {
        Console.WriteLine("AXIX Bank Check Balanace");
    }

    public void MiniStatement()
    {
        Console.WriteLine("AXIX Bank Mini Statement");
    }

    public void ValidateCard()
    {
        Console.WriteLine("AXIX Bank Validate Card");
    }

    public void WithdrawMoney()
    {
```

```

        Console.WriteLine("AXIX Bank Withdraw Money");
    }
}
}

```

Example to Implement Abstraction Principle using Abstract Class and Abstract Methods:

In the below example, we are using abstract class and abstract methods to achieve the abstraction principle in C#. We can achieve 0 to 100% abstraction using the abstract class and abstract methods. In the below example, the user will only know the services that are defined in the abstract class, but how these services are implemented, the user will never know. This is how we can implement abstraction in C# by hiding the implementation details from the user.

```

using System;
namespace Bank
{
    public abstract class IBank
    {
        public abstract void ValidateCard();
        public abstract void WithdrawMoney();
        public abstract void CheckBalanace();
        public abstract void BankTransfer();
        public abstract void MiniStatement();
    }

    public class BankFactory
    {
        public static IBank GetBankObject(string bankType)
        {
            IBank BankObject = null;
            if (bankType == "SBI")
            {
                BankObject = new SBI();
            }
            else if (bankType == "AXIX")

```

```

        {
            BankObject = new AXIX();
        }
        return BankObject;
    }
}

public class SBI : IBank
{
    public override void BankTransfer()
    {
        Console.WriteLine("SBI Bank Bank Transfer");
    }

    public override void CheckBalanace()
    {
        Console.WriteLine("SBI Bank Check Balanace");
    }

    public override void MiniStatement()
    {
        Console.WriteLine("SBI Bank Mini Statement");
    }

    public override void ValidateCard()
    {
        Console.WriteLine("SBI Bank Validate Card");
    }

    public override void WithdrawMoney()
    {
        Console.WriteLine("SBI Bank Withdraw Money");
    }
}

public class AXIX : IBank
{
    public override void BankTransfer()

```

```

{
    Console.WriteLine("AXIX Bank Bank Transfer");
}

public override void CheckBalanace()
{
    Console.WriteLine("AXIX Bank Check Balanace");
}

public override void MiniStatement()
{
    Console.WriteLine("AXIX Bank Mini Statement");
}

public override void ValidateCard()
{
    Console.WriteLine("AXIX Bank Validate Card");
}

public override void WithdrawMoney()
{
    Console.WriteLine("AXIX Bank Withdraw Money");
}

}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Transaction doing SBI Bank");
        IBank sbi = BankFactory.GetBankObject("SBI");
        sbi.ValidateCard();
        sbi.WithdrawMoney();
        sbi.CheckBalanace();
        sbi.BankTransfer();
        sbi.MiniStatement();

        Console.WriteLine("\nTransaction doing AXIX Bank");
    }
}

```

```

    IBank AXIX = BankFactory.GetBankObject("AXIX");
    AXIX.ValidateCard();
    AXIX.WithdrawMoney();
    AXIX.CheckBalanace();
    AXIX.BankTransfer();
    AXIX.MiniStatement();

    Console.Read();
}
}
}

```

- A** Using abstract class, we can achieve 0 to 100% abstraction. The reason is that you can also provide implementation to the methods inside the abstract class. It does not matter whether you implement all methods or none of the methods inside the abstract class. This is allowed, which is not possible with an interface.

Encapsulation vs Abstraction

- The Encapsulation Principle is all about data hiding (or information hiding). On the other hand, the Abstraction Principle is all about detailed hiding (implementation hiding).
- Encapsulation principle, we are exposing the data through publicly exposed methods and properties. The advantage is that we can validate the data before storing and returning it. On the other hand, using the Abstraction Principle, we are exposing only the services so that the user can consume the services, but how the services/methods are implemented is hidden from the user. The user will never know how the method is implemented.

Application Development Process

So, generally, when we are developing an application, the process will be as follows.

- **Step1: Identify the Entities that are associated with the application :** Suppose, we are developing an application for a School. Then for this Student Application, **who are the entities**. The Student is an Entity. TeachingStaff is an Entity. NonTeachingStaff is another Entity. Like this, we can identify the entities. So, in our application, we have identified three entities.



Entities: Student, TeachingStaff, NonTeachingStaff

- **Step2: Identify the attributes of each and every entity.**
 - **Entity: Student:**
 - Attributes: Id, Name, Address, Phone, Class, Marks, Grade, Fees
 - **Entity: TeachingStaff**
 - TeachingStaff Attributes: Id, Name, Address, Phone, Designation, Salary, Qualification, Subject
 - **Entity: NonTeachingStaff**
 - NonTeachingStaff Attributes: Id, Name, Address, Phone, Designation, Salary, DeptName, ManagerId

For a better understanding, please have a look at the below diagram.

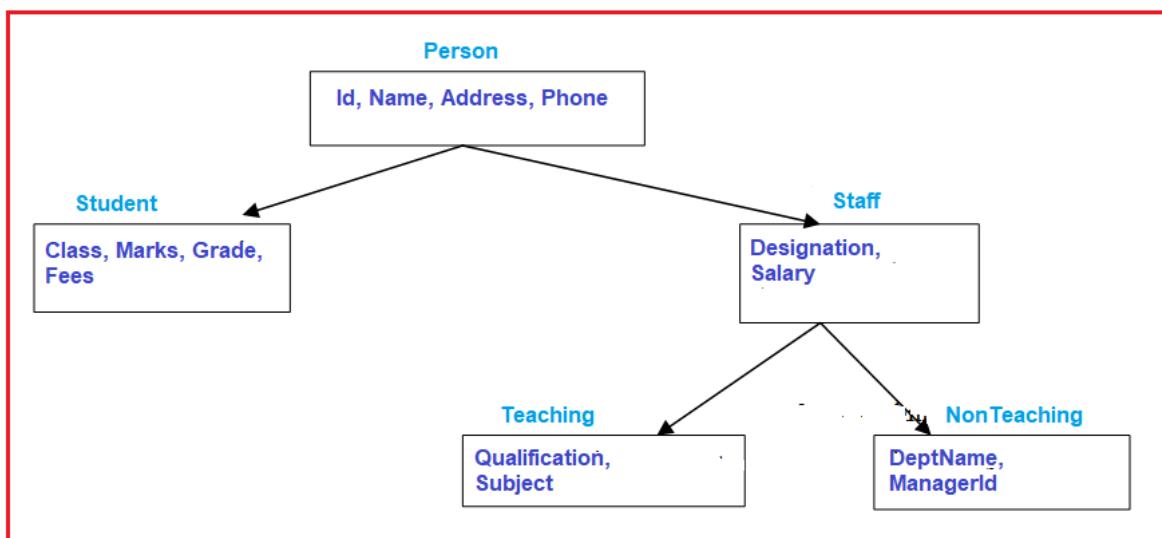
Student	TeachingStaff	NonTeachingStaff
Id Name Address Phone Class Marks Grade Fees	Id Name Address Phone Designation Salary Qualification Subject	Id Name Address Phone Designation Salary DeptName ManagerId

School Application

image_107.png

- Step3: Identify the common attributes and put them in a hierarchical order

The Person contains four attributes Id, Name, Address, and Phone. Under the Person, we have Student and Staff. The Student inherits from the Person, so by default Student will get all those four attributes, and the rest of all other attributes are defined inside the Student Entity. Then we Staff Entity which is also inherited from Person and hence all those four common attributes are also available and plus we have defined the common attributes for Teaching and NonTeaching entities inside the Staff entity. So, Staff will contain six attributes. Finally, both Teaching and NonTeaching are inherited from the Staff Entity.



image_108.png

So, tomorrow if temporary staff comes into the picture, then also these properties are applicable to Temporary Staff. What you need to do is, create a new Entity with the specific properties and inherit it from the Staff entity.

- **Step4: Defining the classes that are representing the entities in Hierarchical order**
- After identifying of attributes of each entity.
- Define classes representing each and every entity. That is one class representing students, one class representing teaching staff, and another class representing the non-teaching staff.

But, if we are defining three classes representing one entity, then there is a problem. **The problem is there are some common attributes in each entity.** So, if we start defining three classes individually, then there is code duplication. Why code duplication? See, we need to define Id three times, Name three times, Address three times, and Phone number three times. Like this, we have duplication in the code.

For all the three entities which are the common attributes? Id, Name, Address, and Phone are the common attributes. Let us put these common attributes in a class called Person. Once we define this class and once, we make this class a Parent class, then no need to define these attributes three times. One time we need to declare in the parent class and then we are consuming these properties under all the child classes. That means reusability comes into the picture.

```
public class Person
{
    public int Id;
    public string Name;
    public string Address;
    public string Phone;
}
```

Now we can define a class called Student inheriting from the Person class. And in the student class, we only need to define the Class, Marks, Grade, and Fees attributes as Id, Name, Address, and Phone are coming from the Person parent class.

```
public class Student : Person
{
    public int Class;
    public float Fees;
    public float Marks;
    public char Grade;
}
```

Next, you can create TeachingStaff and NonTeachingStaff classes inheriting from the Person class. But if you look at the TeachingStaff and NonTeachingStaff entities, apart from the four common attributes i.e. Id, Name, Address, Phone, these two entities also have another two common attributes i.e. Designation and Salary. Again, if we put these two properties in TeachingStaff and NonTeachingStaff classes, duplication comes. So, we need to create a separate class, let us call that class Staff and this Staff class inheriting from the Person class and in this class, we will put the two common properties i.e. Designation and Salary. So, now the Staff class has 6 attributes, four are coming from the Person class and two are defined in this class itself.

```
public class Staff : Person
{
    string Designation;
    double Salary;
}
```

Now, if we make the Staff class a parent class for TeachingStaff and NonTeachingStaff, then by default six attributes are coming. So, in the TeachingStaff we only need to define properties that are only for TeachingStaff such as Qualification and Subject. On the other hand, in the NonTeachingStaff, we only need to define the properties which are only for NonTeachingStaff such as DeptName and ManagerId. And both the TeachingStaff and NonTeachingStaff classes will inherit from the Staff class. Now, we are not going to call them TeachingStaff and NonTeachingStaff, rather we call them Teaching and NonTeaching as they are inheriting from the Staff.

```
public class Teaching : Staff
{
    string Qualification;
```

```
        string Subject;  
    }  
    public class NonTeaching : Staff  
    {  
        string Deptname;  
        string ManagerId;  
    }
```

So, this should be the process of how-to apply Inheritance in Application development.

How to Make use of Inheritance in Realtime Application Development?

Generally, when we develop an application, we will be following a process as follows.

1. Identify the entity associated with the application
2. Identify the attribute that is associated with the application.
3. Now separate the attribute of each entity in a hierarchical order without having any duplicates.
4. Convert those entities into classes.

Interface in C#

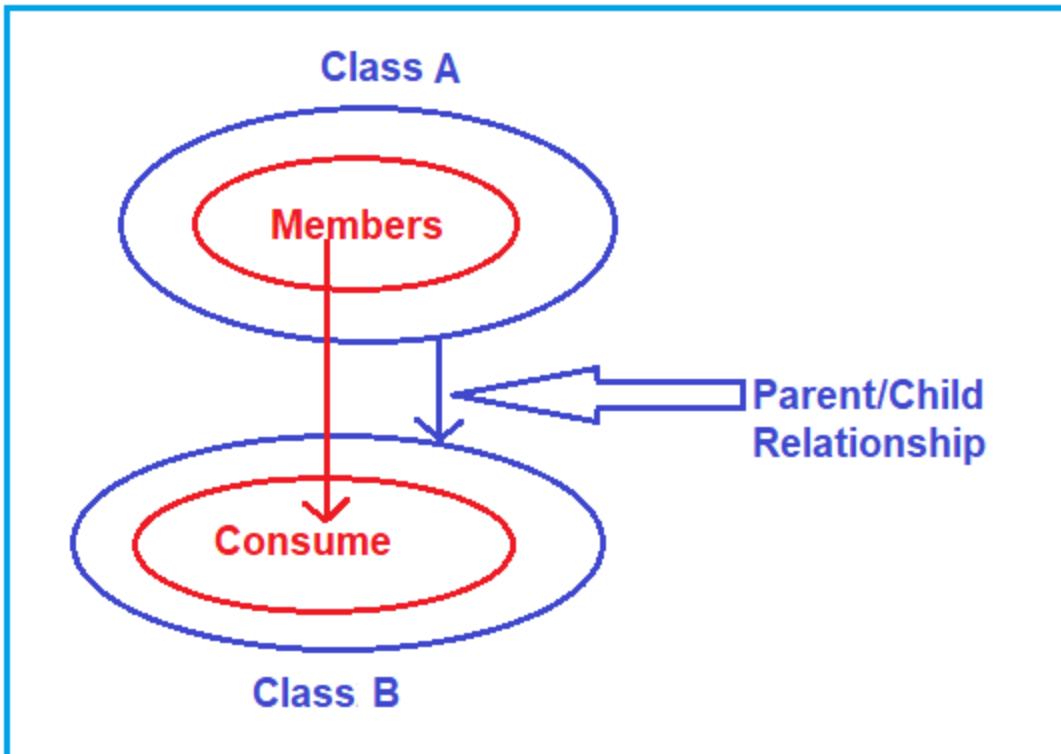
The Interface in C# is a Fully Unimplemented Class used for declaring a set of operations/methods of an object. So, we can define an interface as a pure abstract class, which allows us to define only abstract methods. The abstract method means a method without a body or implementation. It is used to achieve multiple inheritances, which the class can't achieve. It is used to achieve full abstraction because it cannot have a method body.

Differences Between Concrete Class, Abstract Class, and Interface

1. **Class:** Contains only the Non-Abstract Methods (Methods with Method Body).
2. **Abstract Class:** Contains both Non-Abstract Methods (Methods with Method Body) and Abstract Methods (Methods without Method Body).
3. **Interface:** Contain only Abstract Methods (Methods without Method Body).

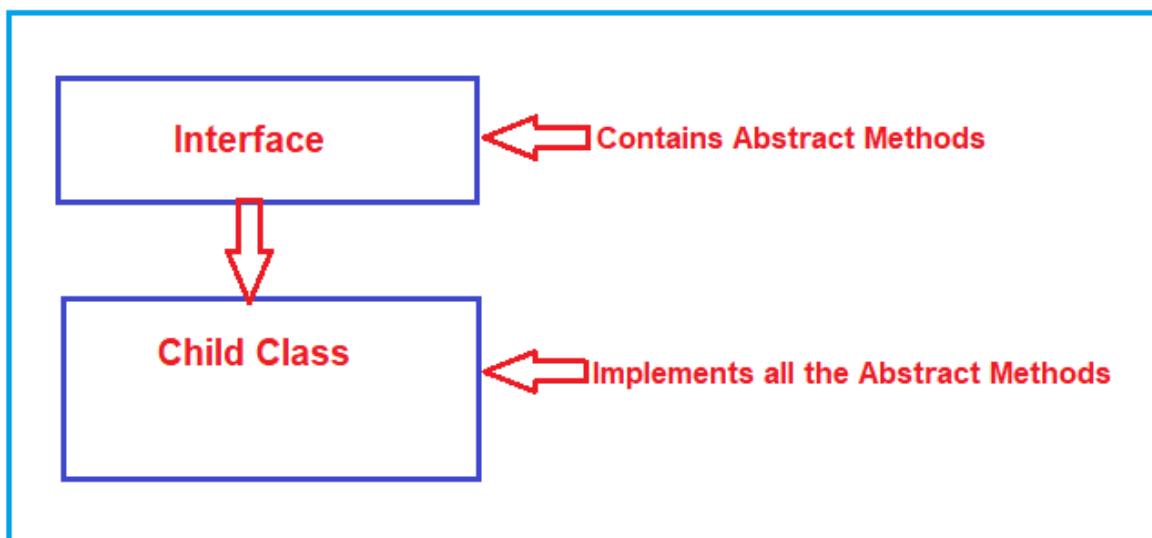


In Inheritance, we already learned that a class inherits from another class, and the child class consumes the Parent class members. Please observe the following diagram. Here, we have class A with a set of members, and class B inherits from class A. And there is a relationship called Parent/Child relation between them. Once the Parent/Child relationship is established, then the members of class A can be consumed under class B. So, this is what we learned in Inheritance.



image_109.png

Now, just like a class has another class as a Parent, a class can also have an Interface as a Parent. If a class has an interface as a Parent, the class is responsible for implementing all the abstract methods of the interface. For a better understanding, please have a look at the below diagram.



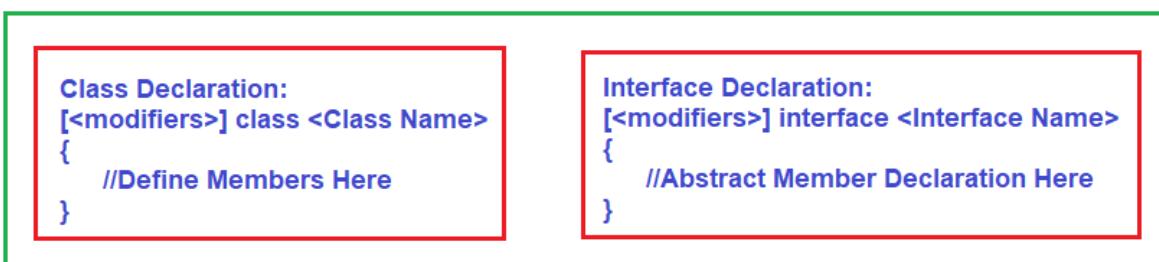
image_110.png

A Simply speaking, the Parent Interface imposes restrictions on the Child Classes. What restrictions? The restrictions are to implement each and every method of the interface under the child class.

A Generally, a class inherits from another class to consume members of its Parent. On the other hand, if a class inherits from an interface, it is to implement the members of its Parent (Interface), not for consumption.

How to Define an Interface in C#

The way we define a class is the same way we need to define an interface in C#. In a class declaration, we need to use the class keyword, whereas in an interface declaration, we need to use the interface keyword. Moreover, in an interface, we can only declare abstract members. For a better understanding, please have a look at the below diagram.



image_111.png

For a better understanding, please have a look at the below example. Here, we have created one interface with the name ITestInterface by using the interface keyword.

```
interface ITestInterface
{
    //Abstract Member Declarations
}
```

How to Define Abstract Methods in an Interface in C#?

In a class (i.e., Abstract Class), we generally use the abstract keyword to define abstract methods as follows.

```
public abstract void Add(int num1, int num2);
```

If you want to write the above abstract method in an interface, then you don't require public and abstract keywords in the method signature as follows:

```
void Add(int num1, int num2);
```

While working with Interface, we need to remember some Rules. Let us discuss those rules one by one with Examples.

- Point 1: The first point that you need to remember is that the default scope for an interface's members is public, whereas it is private in the case of a class.
- Point 2: The second point that you need to remember is by default, every member of an interface is abstract, so we aren't required to use the abstract modifier on it again, just like we do in the case of an abstract class.

Example to Understand Interface in C#:

```
using System;
namespace AbstractClassMethods
{
    interface ITestInterface1
    {
        void Add(int num1, int num2);
    }
    interface ITestInterface2 : ITestInterface1
    {
        void Sub(int num1, int num2);
    }

    public class ImplementationClass1 : ITestInterface1
    {
        //Implement only the Add method
        public void Add(int num1, int num2)
        {
            Console.WriteLine($"Sum of {num1} and {num2} is {num1 +
num2}");
        }
    }
}
```

```

public class ImplementationClass2 : ITestInterface2
{
    //Implement Both Add and Sub method
    public void Add(int num1, int num2)
    {
        Console.WriteLine($"Sum of {num1} and {num2} is {num1 +
num2}");
    }

    public void Sub(int num1, int num2)
    {
        Console.WriteLine($"Divison of {num1} and {num2} is {num1 -
num2}");
    }
}

class Program
{
    static void Main()
    {
        ImplementationClass1 obj1 = new ImplementationClass1();
        //Using obj1 we can only call Add method
        obj1.Add(10, 20);
        //We cannot call Sub method
        //obj1.Sub(100, 20);

        ImplementationClass2 obj2 = new ImplementationClass2();
        //Using obj2 we can call both Add and Sub method
        obj2.Add(10, 20);
        obj2.Sub(100, 20);

        Console.ReadKey();
    }
}

```

Output:

```
Sum of 10 and 20 is 30
Sum of 10 and 20 is 30
Divison of 100 and 20 is 80
```

image_112.png

OOP Principles: Polymorphism

Polymorphism is one of the fundamental OOP concepts and is a term used to describe situations where something takes various roles or forms. In the programming world, these things can be operators or functions.

Example1:

Suppose you are in a classroom, then at that time, you will behave like a student. But when you are in the shopping mall, at that time you will behave like a customer. Again, when you are traveling on a bus, then you will behave like a passenger. And when you are at your home at that time, you will behave like a son or daughter. Here, you are one person but performing different behaviors. This is nothing but polymorphism. The behaviors change based on the place.

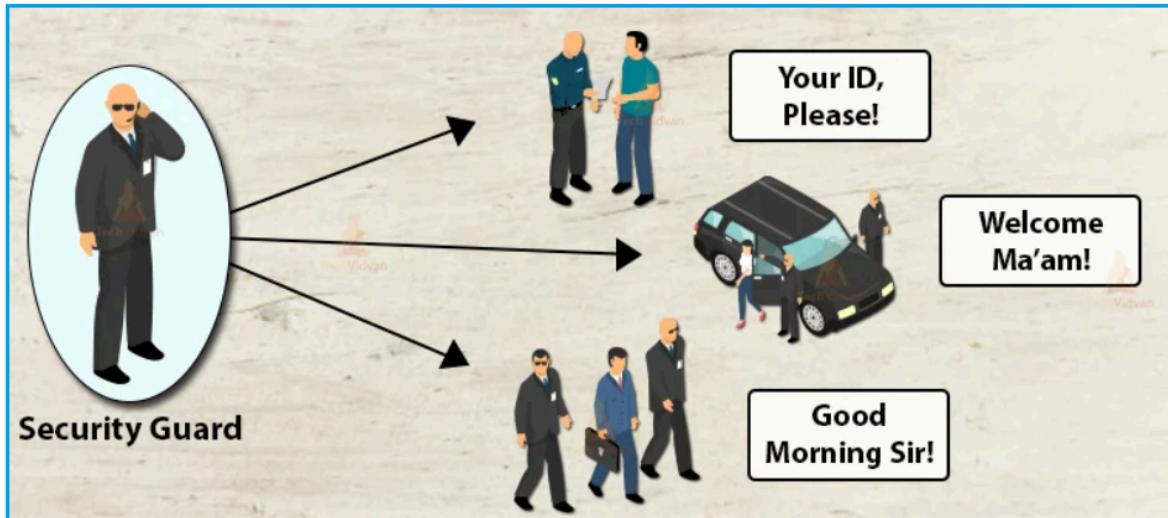


image_113.png

Example2:

A security guard in an organization behaves differently with different people entering the organization. The security behaves in a different way when the Boss comes and, in another way, when the employees come. And when the customers enter, the same security guard will respond differently. So here, the behavior of the security guard

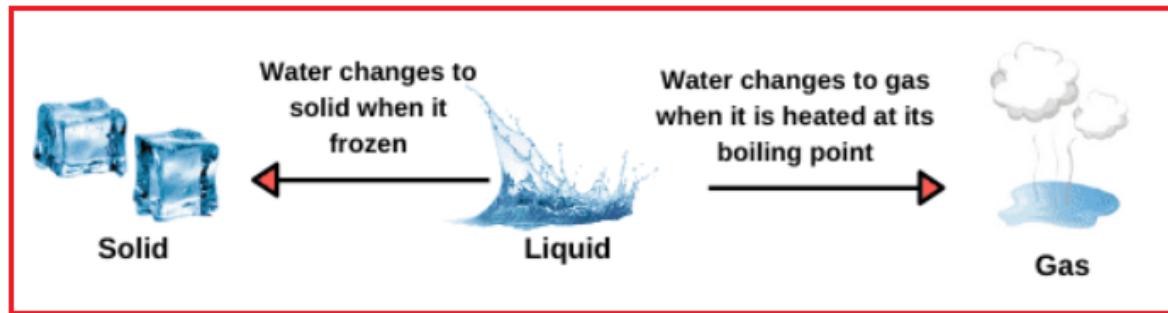
changes from person to person. It depends on the member who is entering the organization.



image_114.png

Example3:

Another good real-time example of polymorphism is water. We all know that water is a liquid at normal temperature, but it changes to a solid when it is frozen, and the same water changes to a gas when it is heated at its boiling point. This is also polymorphism



image_115.png

Example4:

One of the best real-time examples of polymorphism is Women in society. The same woman performs a different role in society. The woman can be the wife of someone, the mother of her child, can be doing a job in an organization, and many more at the same time. But the Woman is only one. So, the same woman performing different roles is nothing but performing polymorphism.



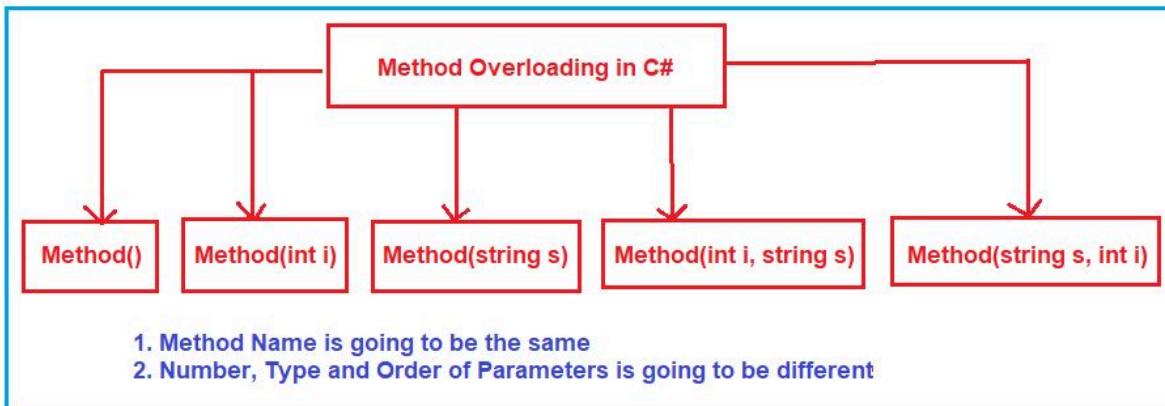
image_116.png

The polymorphism can be implemented using the following three ways.

- Method Overloading
- Operator Overloading
- Method Overriding
- Method Hiding

Method Overloading

Method Overloading means it is an approach to defining multiple methods under the class with a single name. So, we can define more than one method with the same name in a class.



image_117.png

Example to Understand Method Overloading:

```
using System;
namespace MethodOverloading
{
    class Program
    {

        public void Method()
        {
            Console.WriteLine("1st Method");
        }
        public void Method(int i)
        {
            Console.WriteLine("2nd Method");
        }
        public void Method(string s)
        {
            Console.WriteLine("3rd Method");
        }
    }
}
```

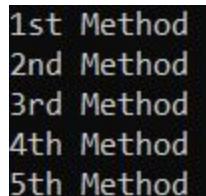
```

    }
    public void Method(int i, string s)
    {
        Console.WriteLine("4th Method");
    }
    public void Method(string s, int i)
    {
        Console.WriteLine("5th Method");
    }
    static void Main(string[] args)
    {
        Program obj = new Program();
        obj.Method(); //Invoke the 1st Method
        obj.Method(10); //Invoke the 2nd Method
        obj.Method("Hello"); //Invoke the 3rd Method
        obj.Method(10, "Hello"); //Invoke the 4th Method
        obj.Method("Hello", 10); //Invoke the 5th Method

        Console.ReadKey();
    }
}

```

Output:



1st Method
2nd Method
3rd Method
4th Method
5th Method

image_118.png

When should we Overload Methods

We have understood what is Method Overloading and how to implement the Method Overloading in C#. But, the important question is when we need to implement or when we need to go for Method Overloading in C#. Let us understand this with an example.

Here, we have created three methods with the same name to perform the addition of two integers, two floats, and two strings. So, when we give two integer numbers we will get

one output and when we provide two string values, then we will get a different output, and similarly, when we give two float numbers we will get another output. That means when the input changes the output or behavior also automatically changes. This is called polymorphism in C#.

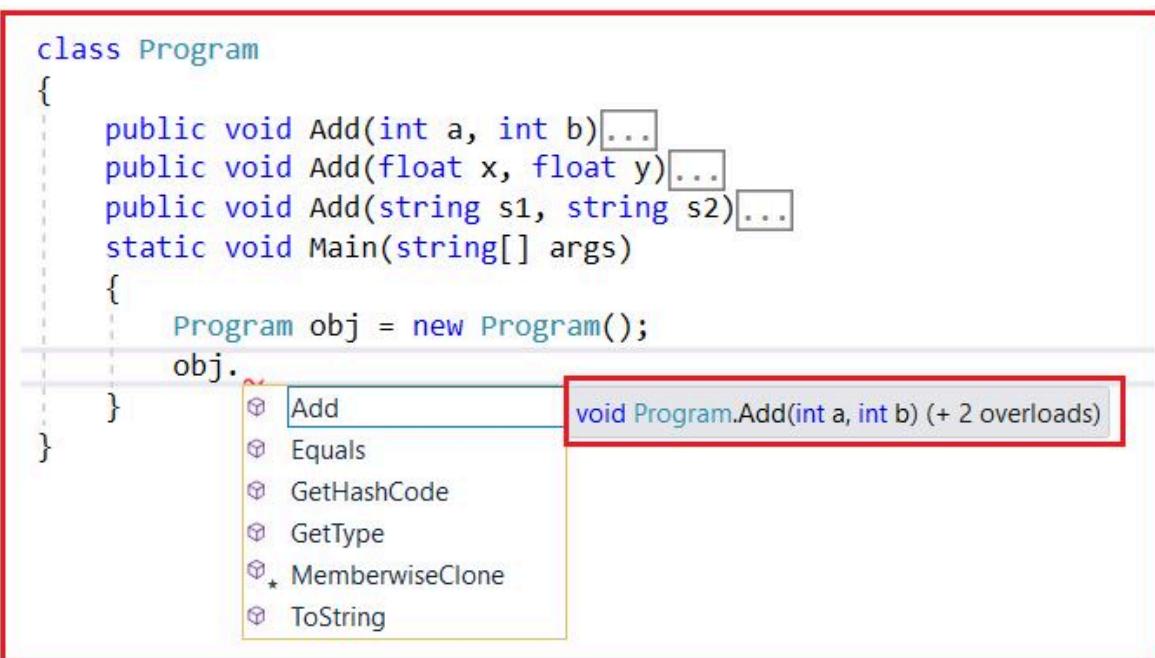
```
using System;
namespace MethodOverloading
{
    class Program
    {
        public void Add(int a, int b)
        {
            Console.WriteLine(a + b);
        }
        public void Add(float x, float y)
        {
            Console.WriteLine(x + y);
        }
        public void Add(string s1, string s2)
        {
            Console.WriteLine(s1 +" "+ s2);
        }
        static void Main(string[] args)
        {
            Program obj = new Program();
            obj.Add(10, 20);
            obj.Add(10.5f, 20.5f);
            obj.Add("Pranaya", "Rout");
            Console.ReadKey();
        }
    }
}
```

Output:

```
30  
31  
Pranaya Rout
```

image_119.png

Suppose you are the user of the Program class and when you create the Program class instance and when you type the object name and dot operator, then you will not see three different Add methods, rather you will see only one Add method with two overloaded versions of the Add method as shown in the below image.



image_120.png

- ⚠** So, the advantage of using Method overloading is that if we overload the methods, then the user of our application gets a comfortable feeling in using the method with an impression that he/she calling one method by passing different types of values. The best example for us is the system-defined "WriteLine()" method. It is an overloaded method, not a single method taking different types of values. If you go to the definition of the Console class, then you will see the following overloaded versions of the WriteLine method defined inside the Console class.

```
...public static void WriteLine(float value);
...public static void WriteLine(int value);
...public static void WriteLine(uint value);
...public static void WriteLine(long value);
...public static void WriteLine(ulong value);
...public static void WriteLine(object value);
...public static void WriteLine(string value);
...public static void WriteLine(string format, object arg0);
...public static void WriteLine(string format, object arg0, object arg1, object arg2);
...public static void WriteLine(string format, object arg0, object arg1, object arg2, object arg3);
...public static void WriteLine(string format, params object[] arg);
...public static void WriteLine(char[] buffer, int index, int count);
...public static void WriteLine(decimal value);
...public static void WriteLine(char[] buffer);
...public static void WriteLine(char value);
...public static void WriteLine(bool value);
...public static void WriteLine(string format, object arg0, object arg1);
...public static void WriteLine(double value);
```

image_121.png

What is Inheritance-Based Method Overloading

A method that is defined in the parent class can also be overloaded under its child class. It is called Inheritance Based Method Overloading in C#. See the following example for a better understanding. As you can see in the below code, we have defined the Add method twice in the class Class1 and also defined the Add method in the child class Class1. Here, notice every Add method takes different types of parameters.

```
using System;
namespace MethodOverloading
{
    class Class1
    {
        public void Add(int a, int b)
        {
            Console.WriteLine(a + b);
        }
        public void Add(float x, float y)
        {
            Console.WriteLine(x + y);
        }
    }
    class Class2 : Class1
    {
```

```

public void Add(string s1, string s2)
{
    Console.WriteLine(s1 +" "+ s2);
}
class Program
{
    static void Main(string[] args)
    {
        Class2 obj = new Class2();
        obj.Add(10, 20);
        obj.Add(10.5f, 20.7f);
        obj.Add("Pranaya", "Rout");
        Console.ReadKey();
    }
}

```

- ⚠** To overload a parent class method under its child class the child class does not require any permission from its parent class.

Example to Understand Constructor Method Overloading

Please have a look at the following example. Here, we are creating three different versions of the Constructor, and each constructor takes a different number of parameters, and this is called Constructor Overloading in C#.

```

using System;
namespace ConstructorOverloading
{
    class ConstructorOverloading
    {
        int x, y, z;
        public ConstructorOverloading(int x)
        {
            Console.WriteLine("Constructor1 Called");
            this.x = 10;
        }
    }
}

```

```
}

public ConstructorOverloading(int x, int y)
{
    Console.WriteLine("Constructor2 Called");
    this.x = x;
    this.y = y;
}
public ConstructorOverloading(int x, int y, int z)
{
    Console.WriteLine("Constructor3 Called");
    this.x = x;
    this.y = y;
    this.z = z;
}
public void Display()
{
    Console.WriteLine($"X={x}, Y={y}, Z={z}");
}
}

class Test
{
    static void Main(string[] args)
    {
        ConstructorOverloading obj1 = new
ConstructorOverloading(10);
        obj1.Display();
        ConstructorOverloading obj2 = new ConstructorOverloading(10,
20);
        obj2.Display();
        ConstructorOverloading obj3 = new ConstructorOverloading(10,
20, 30);
        obj3.Display();
        Console.ReadKey();
    }
}
```

Output:

```
Constructor1 Called  
X=10, Y=0, Z=0  
Constructor2 Called  
X=10, Y=20, Z=0  
Constructor3 Called  
X=10, Y=20, Z=30
```

image_122.png

Method Overriding

The process of re-implementing the superclass non-static, non-private, and non-sealed method in the subclass with the same signature is called Method Overriding in C#. The same signature means the name and the parameters (type, number, and order of the parameters) should be the same.

When do we need to override a method in C#?

- ⚠** If the Super Class or Parent Class method logic is not fulfilling the Sub Class or Child Class business requirements, then the Sub Class or Child Class needs to override the superclass method with the required business logic. Usually, in most real-time applications, the Parent Class methods are implemented with generic logic which is common for all the next-level sub-classes.

How can we Override a Parent Class Method under Child Class

If you want to override the Parent class method in its Child classes, first the method in the parent class must be declared as virtual by using the `virtual` keyword, then only the child classes get permission for overriding that method. **Declaring the method as virtual is marking the method as overridable**. If the child class wants to override the parent class virtual method, **then the child class can override it with the help of the `override` modifier**. But overriding the parent class virtual methods under the child classes is not mandatory. The syntax is shown below to implement Method Overriding in C#.

```

class Class1
{
    //Virtual Function (Overridable Method)
    public virtual void Show()
    {
        //Parent Class Logic Same for All Child Classes
    }
}

class Class2 : Class1
{
    //Overriding Virtual Method
    public override void Show()
    {
        //Child Class Reimplementing the Logic
    }
}

class Class3 : Class1
{
    //Overriding Virtual Method is Optional
}

```

image_123.png

Example to Understand Method Overriding

```

using System;
namespace PolymorphismDemo
{
    class Class1
    {
        //Virtual Function (Overridable Method)
        public virtual void Show()
        {
            //Parent Class Logic Same for All Child Classes
            Console.WriteLine("Parent Class Show Method");
        }
    }
}

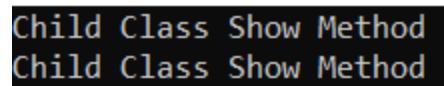
```

```
class Class2 : Class1
{
    //Overriding Method
    public override void Show()
    {
        //Child Class Reimplementing the Logic
        Console.WriteLine("Child Class Show Method");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Class1 obj1 = new Class2();
        obj1.Show();

        Class2 obj2 = new Class2();
        obj2.Show();
        Console.ReadKey();
    }
}
```

Output:



```
Child Class Show Method
Child Class Show Method
```

image_124.png

Method Hiding

Method Overriding is an approach of re-implementing the parent class methods under the child class exactly with the same signature (same name and same parameters).

Method Hiding/Shadowing is also an approach of re-implementing the parent class methods under the child class exactly with the same signature (same name and same parameters).

How we can Re-Implement a Parent Method in the Child Class in C#

We can re-implement the parent class methods under the child classes in two different approaches. They are as follows

- Method Overriding
- Method Hiding Then what are the differences between them, let us understand.

How to Implement Method Hiding/Shadowing

Please have a look at the following image to understand the syntax of Method Hiding/Shadowing in C#. It does not matter whether the parent class method is virtual or not. We can hide both virtual and non-virtual methods under the child class. Again, we can hide the method in the child class in two ways i.e. **by using the new keyword** and also, **without using the new keyword**.

```

public class Parent
{
    public virtual void Show() { }
    public void Display() { }
}

public class Child : Parent
{
    //Method Hiding using new keyword
    public new void Show() { }

    //Method Hiding without using new keyword
    //Getting one warning
    public void Display() { }
}

```

image_125.png

! When we use the new keyword to hide a Parent Class Methods under the child class, then it is called Method Hiding/Shadowing

! Using the new keyword for re-implementing the Parent Class Methods under the child class is optional.

If you look at the Show method, then it is declared as virtual inside the Parent class, so we can override this virtual method inside the Child class by using the override modifier. But we cannot override the Display method inside the Child class as it is not declared as virtual inside the Parent class. But we want to re-implement the method. In that case, we need to re-implement the Parent Class Display Method using the new keyword inside the Child class which is nothing but Method Hiding/Shadowing in C#. The complete example code is given below.

```

using System;
namespace MethodHiding
{

```

```
public class Parent
{
    public virtual void Show()
    {
        Console.WriteLine("Parent Class Show Method");
    }
    public void Display()
    {
        Console.WriteLine("Parent Class Display Method");
    }
}
public class Child : Parent
{
    //Method Overriding
    public override void Show()
    {
        Console.WriteLine("Child Class Show Method");
    }

    //Method Hiding/Shadowing
    public new void Display()
    {
        Console.WriteLine("Child Class Display Method");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Child obj = new Child();
        obj.Show();
        obj.Display();
        Console.ReadKey();
    }
}
```

Output:

```
Child Class Show Method  
Child Class Display Method
```

image_126.png

- ⚠** So, here, you can observe both Method Overriding and Method Hiding doing the same thing. That is re-implementing the Parent class methods under the Child class. Then what are the differences between them? With method overriding, you can re-implement only virtual methods. On the other hand, with Method Hiding, you can re-implement any methods.

OOP Real-Time Examples

Encapsulation Principle

Real-time Example of Encapsulation Principle: Bank Account Management System

One real-time example of encapsulation is a bank account management system. Let's consider the scenario where a bank customer can deposit or withdraw money from their account. Still, certain rules and validations are applied, such as ensuring that the balance does not go below a minimum limit or the customer can't withdraw more than what they have in the account. Let us see how we can implement this example using the Encapsulation Principle in C#:

```
using System;
namespace EncapsulationPrincipleCSharp
{
    public class BankAccount
    {
        // This private field is encapsulated and can't be directly
        // accessed from outside the class.
        private decimal balance;

        public decimal Balance
        {
            // Only provides a way to read the balance but not modify it
            // directly.
            get { return balance; }
        }

        public BankAccount(decimal initialBalance)
        {
            if (initialBalance < 0)
            {
                throw new ArgumentException("Initial balance cannot be
negative.");
            }
        }
    }
}
```

```
    balance = initialBalance;
}

public void Deposit(decimal amount)
{
    if (amount <= 0)
    {
        throw new ArgumentException("Deposit amount should be
positive.");
    }

    balance += amount;
}

public void Withdraw(decimal amount)
{
    if (amount <= 0)
    {
        throw new ArgumentException("Withdrawal amount should be
positive.");
    }

    if (balance - amount < 0)
    {
        throw new InvalidOperationException("Insufficient
funds.");
    }

    balance -= amount;
}

//Testing Encapsulation Principle
public class Program
{
    static void Main(string[] args)
{
```

```

    // Starts with a balance of 500
    BankAccount myAccount = new BankAccount(500);

    // Balance becomes 700
    myAccount.Deposit(200);
    Console.WriteLine(myAccount.Balance); // Outputs: 700

    // Balance becomes 600
    myAccount.Withdraw(100);
    Console.WriteLine(myAccount.Balance); // Outputs: 600

    // myAccount.balance = -1000; // This would be an error, as
    // the balance field is private and inaccessible directly.

    Console.Read();
}
}
}

```

In this Example:

- The BankAccount class encapsulates the balance field, meaning it can't be directly accessed or modified from outside the class. Instead, operations like depositing or withdrawing are controlled by methods (Deposit and Withdraw).
- The Balance property only provides a getter, ensuring the balance can be viewed but not changed directly.
- Methods Deposit and Withdraw encapsulate the business rules of our banking system, such as ensuring you can't have a negative initial balance, can't deposit negative amounts, and can't withdraw more than you have.

Encapsulation Real-time Example: Coffee Machine

Let's consider another Real-time Example of the coffee machine. A coffee machine has internal mechanisms and operations (like grinding coffee beans, heating water, etc.) that the user shouldn't be concerned about. The user selects the type of coffee they want,

and the machine delivers the final product. Let us see how we can implement this example using the Encapsulation Principle in C#:

```
using System;
namespace EncapsulationPrincipleCSharp
{
    public class CoffeeMachine
    {
        private int waterAmount; // in milliliters
        private int beansAmount; // in grams
        private bool isHeated;

        public CoffeeMachine(int water, int beans)
        {
            waterAmount = water;
            beansAmount = beans;
            isHeated = false;
        }

        private void HeatWater()
        {
            if (!isHeated)
            {
                Console.WriteLine("Heating water...");
                isHeated = true;
            }
        }

        private void GrindBeans(int amount)
        {
            if (beansAmount < amount)
            {
                throw new InvalidOperationException("Not enough coffee
beans!");
            }
            Console.WriteLine("Grinding coffee beans...");
            beansAmount -= amount;
        }
    }
}
```

```

}

// This is the method exposed to the user.
public void MakeEspresso()
{
    HeatWater();
    GrindBeans(20); // let's say we need 20 grams of beans for
an espresso
    Console.WriteLine("Making Espresso...");
}

// Another method exposed to the user.
public void MakeLatte()
{
    HeatWater();
    GrindBeans(25); // we need 25 grams of beans for a latte
    Console.WriteLine("Making Latte...");
}

public int BeansLeft()
{
    return beansAmount;
}

public int WaterLeft()
{
    return waterAmount;
}
}

//Testing Encapsulation Principle
public class Program
{
    static void Main(string[] args)
    {
        CoffeeMachine myMachine = new CoffeeMachine(1000, 100); // Initialize with 1000 ml of water and 100 grams of beans
    }
}

```

```

        myMachine.MakeEspresso(); // Outputs: Heating water...
Grinding coffee beans... Making Espresso...

        Console.WriteLine($"Beans left: {myMachine.BeansLeft()}"
grams"); // Outputs: Beans left: 80 grams

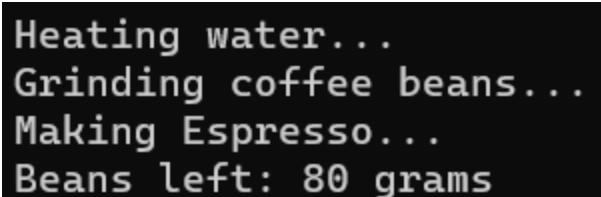
        Console.Read();
    }
}
}

```

In the above example:

- The CoffeeMachine class encapsulates the internal workings of the machine. Methods like HeatWater and GrindBeans are private and cannot be accessed outside the class.
- Only methods meant for public use, like MakeEspresso and MakeLatte, are exposed. This ensures the user interacts with the machine in a controlled manner.
- The internal state of the coffee machine (water and bean amounts, whether water is heated or not) is protected from external manipulation, ensuring its integrity.

This is a simplified representation of a coffee machine, but it illustrates the encapsulation concept: hiding the internal complexity and exposing only what's necessary for the user. When you run the above code, you will get the following output:



```

Heating water...
Grinding coffee beans...
Making Espresso...
Beans left: 80 grams

```

image_127.png

Abstraction Principle

Real-Time Example of Abstraction: Vehicle System

Let's consider a Real-time Example Vehicle System. Suppose you want to model a basic vehicle system. All vehicles can be started and stopped, but the underlying details of

how this happens can differ for each vehicle type. Let us see how we can implement this example using the Abstraction Principle in C#:

```
using System;
namespace AbstractionPrincipleCSharp
{
    //Abstract Base Class (Abstraction)
    public abstract class Vehicle
    {
        // These are abstract methods; the derived classes will provide
        the implementation.
        public abstract void Start();
        public abstract void Stop();
    }

    //Concrete Implementations
    public class Car : Vehicle
    {
        public override void Start()
        {
            Console.WriteLine("Car is starting with a key turn.");
        }

        public override void Stop()
        {
            Console.WriteLine("Car is stopping using its brakes.");
        }
    }

    public class ElectricTrain : Vehicle
    {
        public override void Start()
        {
            Console.WriteLine("Electric train is starting by powering
up.");
        }
    }
}
```

```

        public override void Stop()
        {
            Console.WriteLine("Electric train is stopping by cutting off
the power.");
        }
    }

//Testing Abstraction Principle
public class Program
{
    static void Main(string[] args)
    {
        //Using the Abstraction
        Vehicle myCar = new Car();
        Vehicle myTrain = new ElectricTrain();

        StartVehicle(myCar); // Output: Car is starting with a key
turn.
        StartVehicle(myTrain); // Output: Electric train is starting
by powering up.

        Console.Read();
    }

    static void StartVehicle(Vehicle vehicle)
    {
        vehicle.Start();
    }
}

```

In the example:

- The Vehicle class provides an abstraction for any vehicle. It doesn't detail how a vehicle starts or stops. It just declares that any vehicle should be able to start and stop.
- The Car and ElectricTrain classes are concrete implementations of the Vehicle

abstraction. They provide specific ways to start and stop.

- In the Main method, we use the Vehicle abstraction to start different types of vehicles. The actual method that gets called depends on the type of object passed in, which showcases polymorphism, another OOP principle.

In real-world applications, abstraction allows you to define a contract (in this case, “vehicles can start and stop”) without committing to specific details. This makes it easier to add new types of vehicles in the future without changing existing code.

Real-Time Example of Abstraction Principle: Messaging System

Imagine a system that has to communicate with multiple **messaging platforms: Email, SMS, and Push Notification**. Without abstraction, you might end up with methods like `SendEmail()`, `SendSms()`, `SendPushNotification()`, etc. scattered throughout your application. If a new messaging method is added, you must add another method and call it differently. Let us see how we can implement this example using the Abstraction Principle in C#:

```
using System;
namespace AbstractionPrincipleCSharp
{
    //Abstraction Layer
    //Define a common interface for all messaging platforms
    public interface IMessagingService
    {
        void SendMessage(string recipient, string message);
    }

    //Concrete Implementations
    //Now, we'll create concrete classes that implement this interface.
    public class EmailService : IMessagingService
    {
        public void SendMessage(string recipient, string message)
        {
            Console.WriteLine($"Sending Email to {recipient}:
{message}");
            // Here, you'd have the actual logic to send an email.
        }
    }
}
```

```

        }

    }

public class SmsService : IMessagingService
{
    public void SendMessage(string recipient, string message)
    {
        Console.WriteLine($"Sending SMS to {recipient}: {message}");
        // Here, you'd have the actual logic to send an SMS.
    }
}

public class PushNotificationService : IMessagingService
{
    public void SendMessage(string recipient, string message)
    {
        Console.WriteLine($"Sending Push Notification to
{recipient}: {message}");
        // Here, you'd have the actual logic to send a push
notification.
    }
}

//Testing Abstraction Principle
public class Program
{
    static void Main(string[] args)
    {
        // Using the Abstraction
        // With the Abstraction Principle applied, if you need to
send a message,
        // you don't need to know if it's an email, SMS, or push
notification.
        // You just call SendMessage() on any service implementing
IMessagingService.
        IMessagingService emailService = new EmailService();
        IMessagingService smsService = new SmsService();
        IMessagingService pushService = new

```

```

PushNotificationService();

    SendAlert(emailService, "example@example.com", "Hello via
Email!");
    SendAlert(smsService, "1234567890", "Hello via SMS!");
    SendAlert(pushService, "User123", "Hello via Push
Notification!");

    Console.Read();
}

static void SendAlert(IMessagingService service, string
recipient, string message)
{
    service.SendMessage(recipient, message);
}
}
}

```

By following the Abstraction Principle, we've ensured:

- A consistent method to send messages across different platforms.
- Ease of extension. If we add a new messaging platform, we implement IMessagingService without changing how we send messages in our main application.
- Decoupling. The high-level logic (sending alerts) is decoupled from the low-level messaging details (email, SMS, push).

When you run the above code, you will get the following output:

```

Sending Email to example@example.com: Hello via Email!
Sending SMS to 1234567890: Hello via SMS!
Sending Push Notification to User123: Hello via Push Notification!

```

image_128.png

Inheritance Principle

Real-Time Example of Inheritance Principle: Vehicle Management System

Consider different types of vehicles, such as a basic vehicle, a car, and a motorcycle. All vehicles can be started and stopped and have a speed. However, cars and motorcycles have specific properties and behaviors. Let us see how we can implement this example using the Inheritance Principle in C#:

```
using System;
namespace InheritancePrincipleCSharp
{
    //Base Class (Parent Class) - Vehicle
    public class Vehicle
    {
        public int Speed { get; protected set; }

        public void Start()
        {
            Console.WriteLine("Vehicle started.");
        }

        public void Stop()
        {
            Console.WriteLine("Vehicle stopped.");
        }

        public virtual void Accelerate()
        {
            Speed += 5;
            Console.WriteLine($"Vehicle accelerates. Current speed:
{Speed} km/h.");
        }
    }

    //Derived Class (Child Class) - Car
    public class Car : Vehicle
    {
        public int Doors { get; set; }
```

```

        public override void Accelerate()
        {
            Speed += 10;
            Console.WriteLine($"Car accelerates. Current speed: {Speed} km/h.");
        }

        public void OpenSunroof()
        {
            Console.WriteLine("Sunroof opened.");
        }
    }

    //Derived Class (Child Class) - Motorcycle
    public class Motorcycle : Vehicle
    {
        public bool HasSideCar { get; set; }

        public override void Accelerate()
        {
            Speed += 7;
            Console.WriteLine($"Motorcycle accelerates. Current speed: {Speed} km/h.");
        }

        public void UseKickstand()
        {
            Console.WriteLine("Kickstand placed.");
        }
    }

    //Testing Inheritance Principle
    public class Program
    {
        static void Main(string[] args)
        {
            //Using the Inheritance
            Car myCar = new Car { Doors = 4 };

```

```

        myCar.Start();
        myCar.Accelerate();
        myCar.OpenSunroof();
        myCar.Stop();

        Console.WriteLine();

        Motorcycle myBike = new Motorcycle { HasSideCar = false };
        myBike.Start();
        myBike.Accelerate();
        myBike.UseKickstand();
        myBike.Stop();

        Console.Read();
    }
}

```

In this Example:

- The Vehicle class represents the general properties and behaviors of vehicles.
- The Car and Motorcycle classes inherit from Vehicle, which means they automatically get the Start, Stop, and Accelerate methods. Additionally, they can have their own specific properties (like Doors for Cars or HasSideCar for Motorcycles) and behaviors (like OpenSunroof for Cars or UseKickstand for Motorcycles).
- The Accelerate method is overridden in both Car and Motorcycle to provide specific acceleration behaviors for each.

This example demonstrates the power of inheritance to promote code reuse, establish hierarchies, and allow specialized behavior in derived classes. When you run the above code, you will get the following output:

```
Vehicle started.  
Car accelerates. Current speed: 10 km/h.  
Sunroof opened.  
Vehicle stopped.  
  
Vehicle started.  
Motorcycle accelerates. Current speed: 7 km/h.  
Kickstand placed.  
Vehicle stopped.
```

image_129.png

Real-Time Example of Inheritance Principle: Education System

Let's consider another real-time example to understand the inheritance principle in C#, i.e., the education system scenario with students and teachers. In an educational institution, students and teachers are people with common attributes like name, age, and address. However, they also have distinct properties and behaviors. For instance, a student may enroll in a course while a teacher might teach one. Let us see how we can implement this example using the Inheritance Principle in C#:

```
using System;  
namespace InheritancePrincipleCSharp  
{  
    //Base Class (Parent Class) - Person  
    public class Person  
    {  
        public string Name { get; set; }  
        public int Age { get; set; }  
        public string Address { get; set; }  
  
        public Person(string name, int age, string address)  
        {  
            Name = name;  
            Age = age;  
            Address = address;  
        }  
    }
```

```

        public void DisplayDetails()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}, Address:
{Address}");
    }
}

//Derived Class (Child Class) - Student
public class Student : Person
{
    public string StudentId { get; set; }

    public Student(string name, int age, string address, string
studentId)
        : base(name, age, address) // Calling base class constructor
    {
        StudentId = studentId;
    }

    public void Enroll(string courseName)
    {
        Console.WriteLine($"{Name} has enrolled in {courseName}
course.");
    }
}

//Derived Class (Child Class) - Teacher
public class Teacher : Person
{
    public string EmployeeId { get; set; }

    public Teacher(string name, int age, string address, string
employeeId)
        : base(name, age, address) // Calling base class constructor
    {
        EmployeeId = employeeId;
    }
}

```

```

        public void Teach(string courseName)
    {
        Console.WriteLine($"'{Name}' is teaching {courseName}
course.");
    }
}

//Testing Inheritance Principle
public class Program
{
    static void Main(string[] args)
    {
        //Using the Inheritance
        Student john = new Student("John Doe", 20, "123 Main St",
"S12345");
        john.DisplayDetails();
        john.Enroll("Mathematics");

        Console.WriteLine();

        Teacher mrsSmith = new Teacher("Mrs. Smith", 40, "456 Elm
St", "T98765");
        mrsSmith.DisplayDetails();
        mrsSmith.Teach("Physics");

        Console.Read();
    }
}

```

In this example:

- The Person class captures general attributes and behaviors common to students and teachers.
- The Student and Teacher classes inherit from Person. This inheritance means they automatically have properties like Name, Age, and Address and the DisplayDetails

method. Additionally, they introduce specific properties and methods like Enroll for Student and Teach for Teacher.

- The derived classes' constructors utilize the base keyword to invoke the base class's constructor, ensuring that the common properties are set.

This example demonstrates how inheritance can model real-world relationships, facilitate code reuse, and provide a structured way to represent hierarchies in your system

Polymorphism Principle

Real-Time Example of Polymorphism Principle: Animals Sounds

Here's a simple real-time example: consider animals making sounds. While every animal makes a sound, each animal's sound is distinct. You can leverage polymorphism to model this scenario. Let's walk through it:

- Base Class: Animal
- Derived Classes: Dog, Cat Let us see how we can implement this example using the Polymorphism Principle in C#:

```
using System;

namespace PolymorphismExample
{
    // Base class
    public abstract class Animal
    {
        public abstract void MakeSound();
    }

    // Derived class
    public class Dog : Animal
    {
        public override void MakeSound()
        {
```

```

        Console.WriteLine("The dog barks.");
    }
}

// Another derived class
public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("The cat meows.");
    }
}

//Testing Polymorphism Principle
public class Program
{
    static void Main(string[] args)
    {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        MakeAnimalSound(myDog); // Outputs: The dog barks.
        MakeAnimalSound(myCat); // Outputs: The cat meows.

        Console.Read();
    }

    // This function showcases polymorphism in action.
    // Even though it accepts a parameter of type 'Animal',
    // it's able to handle any derived type.
    static void MakeAnimalSound(Animal animal)
    {
        animal.MakeSound();
    }
}

```

In this Example:

- We have an abstract Animal class with an abstract method, MakeSound().
- Derived classes (Dog and Cat) provide their own implementation of the MakeSound() method.
- In the Program class, even though we use the Animal type to hold references to the derived classes, we can still call the appropriate derived class's MakeSound() method. This is the essence of polymorphism.

This allows for flexibility and makes adding more animal types in the future easier without making major changes to existing code. If you were to add a new animal, say Bird, you'd need to create a Bird class derived from Animal and provide its own implementation for the MakeSound() method.

Real-Time Example of Polymorphism Principle: Payment Processing System

Let's understand polymorphism using another real-world example: Payment Processing System. Different payment methods, such as credit cards, bank transfers, and digital wallets, could have different processes to execute a payment. Here's how this can be represented with polymorphism:

- Base Class: PaymentMethod
- Derived Classes: CreditCard, BankTransfer, DigitalWallet Let us see how we can implement this example using the Polymorphism Principle in C#:

```
using System;

namespace PaymentPolymorphismExample
{
    // Base class
    public abstract class PaymentMethod
    {
        public abstract void ExecutePayment(decimal amount);
    }
}
```

```

// Derived class for credit card payment
public class CreditCard : PaymentMethod
{
    public override void ExecutePayment(decimal amount)
    {
        Console.WriteLine($"Processing a credit card payment for
${amount}.");
        // Here you'd have logic specific to credit card processing
    }
}

// Derived class for bank transfer
public class BankTransfer : PaymentMethod
{
    public override void ExecutePayment(decimal amount)
    {
        Console.WriteLine($"Processing a bank transfer for
${amount}.");
        // Logic specific to bank transfers would be here
    }
}

// Derived class for digital wallet payment
public class DigitalWallet : PaymentMethod
{
    public override void ExecutePayment(decimal amount)
    {
        Console.WriteLine($"Processing a digital wallet payment for
${amount}.");
        // Logic specific to digital wallet payment would go here
    }
}

class Program
{
    static void Main(string[] args)
    {
        PaymentMethod creditCardPayment = new CreditCard();

```

```

        PaymentMethod bankTransferPayment = new BankTransfer();
        PaymentMethod digitalWalletPayment = new DigitalWallet();

        ProcessPayment(creditCardPayment, 100.00M);           // Outputs:
Processing a credit card payment for $100.00.
        ProcessPayment(bankTransferPayment, 250.50M);       // Outputs:
Processing a bank transfer for $250.50.
        ProcessPayment(digitalWalletPayment, 75.25M);      // Outputs:
Processing a digital wallet payment for $75.25.

        Console.ReadKey();
    }

    // Demonstrating polymorphism.
    // This function can accept any payment method derived from
PaymentMethod.

    static void ProcessPayment(PaymentMethod paymentMethod, decimal
amount)
    {
        paymentMethod.ExecutePayment(amount);
    }
}

```

In this Example:

- The PaymentMethod base class provides a contract with an abstract ExecutePayment method.
- Each derived class (e.g., CreditCard, BankTransfer, DigitalWallet) provides its own specific implementation for processing the payment.
- The ProcessPayment function in the Program class accepts any payment method (an object of a type derived from PaymentMethod) and processes the payment using polymorphism. This provides flexibility to introduce new payment methods in the future without changing the core payment processing logic.

When you run the above code, you will get the following output:

```
Processing a credit card payment for $100.00.  
Processing a bank transfer for $250.50.  
Processing a digital wallet payment for $75.25.
```

image_130.png

Interface

Real-Time Example of Interface: Vehicles

Imagine you have different types of vehicles like cars, boats, and airplanes. Each of these vehicles can move, but how they move differs. Create a simple program where each vehicle type displays its mode of movement. Let us implement this example using Interface in C#:

```
using System;  
using System.Collections.Generic;  
  
namespace InterfaceCSharp  
{  
    //Step 1: Define the IMovable interface.  
    public interface IMovable  
    {  
        void Move();  
    }  
  
    //Step 2: Implement the interface for different types of vehicles.  
    // Car.cs  
    public class Car : IMovable  
    {  
        public void Move()  
        {  
            Console.WriteLine("The car drives on the road.");  
        }  
    }  
  
    // Boat.cs  
    public class Boat : IMovable
```

```
{  
    public void Move()  
    {  
        Console.WriteLine("The boat sails on the water.");  
    }  
}  
  
// Airplane.cs  
public class Airplane : IMovable  
{  
    public void Move()  
    {  
        Console.WriteLine("The airplane flies in the sky.");  
    }  
}  
  
//Step 4: Testing the application.  
class Program  
{  
    static void Main(string[] args)  
    {  
        List<IMovable> vehicles = new List<IMovable>  
        {  
            new Car(),  
            new Boat(),  
            new Airplane()  
        };  
  
        foreach (var vehicle in vehicles)  
        {  
            vehicle.Move();  
        }  
        Console.ReadKey();  
    }  
}
```

In this real-time example, the interface `IMovable` acts as a contract, ensuring that every vehicle type that implements it will have a `Move` method. This abstraction allows us to handle different types of vehicles uniformly, demonstrated in the foreach loop in the main program.

With this pattern, if we want to introduce more vehicle types (e.g., a bicycle, a skateboard, or a rocket), we create a new class for each and implement the `IMovable` interface. Our main program remains unchanged and functions correctly for any new vehicle type. When you run the above code, you will get the following output:

```
The car drives on the road.  
The boat sails on the water.  
The airplane flies in the sky.
```

image_131.png

Real-Time Example of Interface: Payment Gateway Integration

This is a typical scenario in many applications, where you might want to support multiple payment providers (like PayPal, Stripe, and others). In this context, interfaces become invaluable because they allow us to define a consistent way of processing payments, regardless of the underlying provider. Let us implement this example using Interface in C#:

```
using System;  
namespace InterfaceCSharp  
{  
    //Step 1: Define the IPaymentGateway interface.  
    public interface IPaymentGateway  
    {  
        bool ProcessPayment(decimal amount);  
    }  
  
    //Step 2: Implement the interface for different payment providers.  
    // PayPalPaymentGateway.cs  
    public class PayPalPaymentGateway : IPaymentGateway  
    {  
        public bool ProcessPayment(decimal amount)  
        {
```

```

        // Call PayPal's API to process the payment
        Console.WriteLine($"Processing ${amount} payment using
PayPal...");

        return true; // assume success for the sake of this example
    }
}

// StripePaymentGateway.cs
public class StripePaymentGateway : IPaymentGateway
{
    public bool ProcessPayment(decimal amount)
    {
        // Call Stripe's API to process the payment
        Console.WriteLine($"Processing ${amount} payment using
Stripe...");

        return true; // assume success for this example
    }
}

//Step 3: Use the implementations in a shopping cart scenario.
public class ShoppingCart
{
    private IPaymentGateway _paymentGateway;

    public ShoppingCart(IPaymentGateway paymentGateway)
    {
        _paymentGateway = paymentGateway;
    }

    public void Checkout(decimal amount)
    {
        if (_paymentGateway.ProcessPayment(amount))
        {
            Console.WriteLine("Payment was successful!");
        }
        else
        {
            Console.WriteLine("Payment failed. Please try again.");
        }
    }
}

```

```

        }
    }

//Step 4: Testing the application.
class Program
{
    static void Main(string[] args)
    {
        // Customer selects PayPal as their preferred payment method
        ShoppingCart cart = new ShoppingCart(new
PayPalPaymentGateway());
        cart.Checkout(100.00M);

        // Another customer selects Stripe
        cart = new ShoppingCart(new StripePaymentGateway());
        cart.Checkout(200.00M);

        Console.ReadKey();
    }
}

```

In this real-time example, the `IPaymentGateway` interface abstracts the specifics of processing payments. The `ShoppingCart` class doesn't need to know the details of each payment provider. It is called `ProcessPayment`, and the correct payment provider takes care of the rest.

This architecture makes it easy to add more payment providers in the future. When you want to add a new one, implement the `IPaymentGateway` interface, and the rest of the system can remain unchanged. When you run the above code, you will get the following output:

```

Processing $100.00 payment using PayPal...
Payment was successful!
Processing $200.00 payment using Stripe...
Payment was successful!

```

image_132.png

Real-Time Example of Interface in C#: User Authentication Methods

Imagine a scenario where an application allows users to authenticate using different methods, such as a password, fingerprint, or face recognition. Develop a mechanism that can support multiple authentication methods without changing the core authentication process. Let us implement this example using Interface in C#:

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

namespace InterfaceCSharp
{
    //Step 1: Define the IAuthenticator interface.
    public interface IAuthenticator
    {
        bool Authenticate();
    }

    //Step 2: Implement the interface for different authentication
    methods.

    // PasswordAuthenticator.cs
    public class PasswordAuthenticator : IAuthenticator
    {
        public bool Authenticate()
        {
            // Logic for password authentication
            Console.WriteLine("Authenticating using password...");
            return true; // For simplicity, assume authentication
            always succeeds
        }
    }

    // FingerprintAuthenticator.cs
    public class FingerprintAuthenticator : IAuthenticator
    {
        public bool Authenticate()
        {
            // Logic for fingerprint authentication
        }
    }
}
```

```

        Console.WriteLine("Authenticating using fingerprint...");
        return true;
    }
}

// FaceRecognitionAuthenticator.cs
public class FaceRecognitionAuthenticator : IAuthenticator
{
    public bool Authenticate()
    {
        // Logic for face recognition authentication
        Console.WriteLine("Authenticating using face
recognition...");
        return true;
    }
}

//Step 3: Implement the authentication process using the interfaces.
public class AuthService
{
    private IAuthenticator _authenticator;

    public AuthService(IAuthenticator authenticator)
    {
        _authenticator = authenticator;
    }

    public void AuthenticateUser()
    {
        if (_authenticator.Authenticate())
        {
            Console.WriteLine("Authentication successful!");
        }
        else
        {
            Console.WriteLine("Authentication failed.");
        }
    }
}

```

```

    }

    //Step 4: Testing the application.
    class Program
    {
        static void Main(string[] args)
        {
            AuthService passwordService = new AuthService(new
PasswordAuthenticator());
            passwordService.AuthenticateUser();

            AuthService fingerprintService = new AuthService(new
FingerprintAuthenticator());
            fingerprintService.AuthenticateUser();

            AuthService faceService = new AuthService(new
FaceRecognitionAuthenticator());
            faceService.AuthenticateUser();

            Console.ReadKey();
        }
    }
}

```

The IAuthenticator interface offers a consistent way to implement different authentication methods. The main AuthService class remains decoupled from the specifics of each method. This architecture makes it easy to add or change authentication methods in the future without affecting the core logic of the AuthService.

Abstract Class

Real-Time Example of Abstract Class in C#: Banking System

Let's see a real-time example of an abstract class in C# by considering a scenario related to the Banking System.

- Abstract Class – BankAccount

- Concrete Classes – SavingsAccount, CurrentAccount

Let us implement this example using Abstract Class in C#:

```

using System;
namespace AbstractClassCSharp
{
    //BankAccount.cs (Abstract Class)
    public abstract class BankAccount
    {
        public string AccountNumber { get; private set; }
        public string AccountHolder { get; private set; }
        protected double Balance { get; set; }

        public BankAccount(string accountNumber, string accountHolder)
        {
            AccountNumber = accountNumber;
            AccountHolder = accountHolder;
        }

        // Abstract methods
        public abstract void Deposit(double amount);
        public abstract void Withdraw(double amount);

        // Virtual method with a default implementation
        public virtual void DisplayBalance()
        {
            Console.WriteLine($"Account Number: {AccountNumber}, Account
Holder: {AccountHolder}, Balance: ${Balance}");
        }
    }

    //SavingsAccount.cs (Concrete Class)
    public class SavingsAccount : BankAccount
    {
        private double _interestRate = 0.03; // 3% annual interest
    }
}

```

```

        public SavingsAccount(string accountNumber, string
accountHolder)
            : base(accountNumber, accountHolder) { }

        public override void Deposit(double amount)
        {
            Balance += amount;
            Console.WriteLine($"Deposited ${amount} into Savings
Account.");
        }

        public override void Withdraw(double amount)
        {
            if (Balance - amount >= 0)
            {
                Balance -= amount;
                Console.WriteLine($"Withdrew ${amount} from Savings
Account.");
            }
            else
            {
                Console.WriteLine("Insufficient funds for withdrawal.");
            }
        }

        public void AddInterest()
        {
            Balance += Balance * _interestRate;
            Console.WriteLine($"Interest added. New Balance:
${Balance}");
        }
    }

    //CurrentAccount.cs (Concrete Class)
    public class CurrentAccount : BankAccount
    {
        private double _overdraftLimit = 1000.0;

```

```

        public CurrentAccount(string accountNumber, string
accountHolder)
            : base(accountNumber, accountHolder) { }

        public override void Deposit(double amount)
        {
            Balance += amount;
            Console.WriteLine($"Deposited ${amount} into Current
Account.");
        }

        public override void Withdraw(double amount)
        {
            if ((Balance - amount) >= -_overdraftLimit)
            {
                Balance -= amount;
                Console.WriteLine($"Withdrew ${amount} from Current
Account.");
            }
            else
            {
                Console.WriteLine("Withdrawal exceeds overdraft
limit.");
            }
        }
    }

    //Step 4: Testing the application.
    class Program
    {
        static void Main(string[] args)
        {
            SavingsAccount johnsSavings = new SavingsAccount("SA123456",
"John Doe");
            johnsSavings.Deposit(1000);
            johnsSavings.Withdraw(200);
            johnsSavings.AddInterest();
            johnsSavings.DisplayBalance();
        }
    }
}

```

```

        CurrentAccount janesCurrent = new CurrentAccount("CA654321",
"Jane Smith");
        janesCurrent.Deposit(500);
        janesCurrent.Withdraw(1000);
        janesCurrent.DisplayBalance();

        Console.ReadKey();
    }
}
}

```

In this example, the abstract class `BankAccount` defines any bank account's essential structure and functionalities, such as depositing and withdrawing money. The concrete classes, `SavingsAccount` and `CurrentAccount`, provide specific implementations for these functionalities and may have additional features like adding interest or handling overdrafts. When you run the above code, you will get the following output:

```

Deposited $1000 into Savings Account.
Withdrew $200 from Savings Account.
Interest added. New Balance: $824
Account Number: SA123456, Account Holder: John Doe, Balance: $824
Deposited $500 into Current Account.
Withdrew $1000 from Current Account.
Account Number: CA654321, Account Holder: Jane Smith, Balance: $-500

```

image_134.png

OOP Real-Time Exercise Scenarios

Encapsulation

Encapsulation Real-time Example in C#: Car's speedometer

Consider another real-world example: a car's speedometer and related components. A car's speedometer displays the current speed to the driver. Internally, this speed is determined by several factors and calculations related to wheel rotations and gear ratios, but the driver only needs to see the final speed value. They don't need to know (and often don't want to know) the complex calculations and mechanisms involved.

In the above Example:

- The car's internal mechanism (e.g., the Move method and wheelRotations variable) is hidden from the driver.
- The driver only interacts with the Drive and ResetSpeedometer methods, which are exposed for public use.
- The CalculateSpeed method, though internal, takes care of the logic related to determining the current speed based on the wheel rotations and gear ratio.

Here, the complexity of calculating the speed and the internal state management related to wheel rotations are encapsulated within the Car class, and the driver only sees the final speed result.

Encapsulation Principle Real-time Example in C#: Digital Wallet

Let's see a different example to understand the Encapsulation Principle in C#, i.e., a digital wallet. A digital wallet stores money and users can perform operations such as depositing, withdrawing, and checking their balance. Internally, there might be various checks, encryption, and transaction logging, but these details are abstracted from the user. Let us see how we can implement this example using the Encapsulation Principle in C#:

In this Example:

- The DigitalWallet class encapsulates the balance and the transaction log.
- Deposit and withdrawal methods are designed with safety checks, and the transaction log records all transactions.
- Password verification is required for certain actions, enhancing security.
- GetTransactionLog returns a copy of the transaction log list to prevent external modifications, further emphasizing encapsulation.

Abstraction

Real-Time Example of Abstraction Principle in C#: Payment Processing

Let's imagine a scenario where you're developing a system for an online store that supports multiple payment methods like Credit Cards, PayPal, and Bitcoin. Without abstraction, you'd potentially have disparate methods like ProcessCreditCardPayment(), ProcessPayPalPayment(), etc. This would make the system rigid and difficult to extend if you were to introduce new payment methods. Let us see how we can implement this example using the Abstraction Principle in C#:

In this example, With abstraction:

- We've streamlined the process of handling various payment methods.
- Introducing a new payment method in the future becomes straightforward: implement the IPaymentMethod interface.
- The CheckoutSystem class is decoupled from specific payment implementations, making the codebase easier to maintain and extend.

Real-Time Example of Abstraction Principle in C#: Zoo Management System

Consider a zoo management system where you have various types of animals, and each animal can make a sound. Without abstraction, you'd have separate methods for each animal's sound. But with the Abstraction Principle, you can simplify this. Let us see how we can implement this example using the Abstraction Principle in C#:

Advantages:

- Consistency: The ZooKeeper can interact with any animal using a consistent method, `MakeSound()`, regardless of the animal's specific type.
- Extensibility: To introduce a new animal into the system, you create a new class that implements the `IAnimal` interface. The rest of the system doesn't need to change.
- Decoupling: The `ZooKeeper` class is now decoupled from specific animal implementations. This makes the codebase more maintainable and easier to understand.

Inheritance

Real-Time Example of Inheritance Principle in C#: Animals in a Zoo

A zoo has different types of animals, like mammals, birds, and reptiles. All animals have common properties such as name, age, and diet, but each type might have unique behaviors. For instance, birds can fly, while mammals might have a specific communication method. Let us see how we can implement this example using the Inheritance Principle in C#:

In this example:

- The `Animal` class represents general properties and behaviors common to all animals.
- The `Bird` and `Mammal` classes inherit from `Animal`, implying they automatically obtain properties like `Name`, `Age`, `Diet`, and the `Eat` method. However, they also introduce specific behaviors: `Bird` has a `Fly` method, while `Mammal` has a `Communicate` method.
- The `Display` method is overridden in both `Bird` and `Mammal` classes to provide specialized behavior in addition to the base behavior.

Real-Time Example of Inheritance Principle in C#: Library System

A library contains various items, including books, magazines, and DVDs. All library items have a unique identifier and a title and can be borrowed or returned. However, each type might have unique properties; for instance, a book has an author and many pages, while a DVD might have a runtime. Let us see how we can implement this example using the Inheritance Principle in C#:

In this Example:

- The LibraryItem class defines basic properties and behaviors common to all library items, such as borrowing and returning.
- The Book and DVD classes inherit from LibraryItem, automatically obtaining properties like Id and Title and methods like Borrow and Return. Yet, they also add their own unique attributes: Book introduces Author and Pages, while DVD introduces Runtime.
- Each class has a method to display specific details: DisplayBookInfo for Book and DisplayDVDInfo for DVD.

Through this example, the inheritance principle in C# allows us to model a real-world library system, promote code reuse, and maintain an organized structure of the different items within the library.

Polymorphism

Real-Time Example of Polymorphism Principle in C#: Managing a Fleet of Vehicles

Let's understand polymorphism using another real-world example: Managing a Fleet of Vehicles. Each vehicle can be driven, but the driving experience and method might vary based on the type of vehicle. For example, you drive a car differently than you would pilot a boat.

- Base Class: Vehicle
- Derived Classes: Car, Boat, Bicycle

In this Example:

- The Vehicle is the base class with an abstract method, Drive.
- The derived classes Car, Boat, and Bicycle each provide their own unique implementations of the Drive method.
- The OperateVehicle function in the Program class can accept any object of type Vehicle (or derived from Vehicle) and invokes the appropriate Drive method due to polymorphism.

Should you need to introduce a new type of vehicle, such as a “Helicopter,” you’d create a new derived class from the Vehicle and implement a specific way to pilot it. No changes to the OperateVehicle method would be necessary.

Real-Time Example of Polymorphism Principle in C#: Notification System

Let’s understand polymorphism using another real-world example: a notification system in which different notification methods (e.g., Email, SMS, Push Notification) are utilized. Here’s the setup:

- Base Class: Notification
- Derived Classes: EmailNotification, SmsNotification, PushNotification

In this Example:

- The base class Notification has properties like Recipient and Message and an abstract method Send.
- The derived classes (EmailNotification, SmsNotification, PushNotification) each provide specific implementations for the Send method.
- The SendNotification method in the Program class accepts any notification object derived from the base Notification class and sends it using polymorphism.

This approach allows for easy scalability. If, in the future, you wanted to add a new type of notification, say a “SlackNotification,” you’d derive it from the base Notification class and provide a specific Send implementation.

Interface

Real-Time Example of Interface in C#: Database Operations

Suppose we’re working on an application where we need to support interactions with different types of databases. By using interfaces, we can make our application flexible, extensible, and decoupled from the specifics of each database. Let us implement this example using Interface in C#:

In this real-world example, the IDatabase interface provides a clear contract for database operations, ensuring consistency regardless of the underlying database system. As a

result, our DatabaseManager class becomes adaptable to any database system that implements the IDatabase interface. This means we can easily support new database systems without major changes to our application. When you run the above code, you will get the following output:

```
Connected to SQL Database.  
Inserted 'SampleData1' into SQL Database.  
Disconnected from SQL Database.  
Connected to NoSQL Database.  
Inserted 'SampleData2' into NoSQL Database.  
Disconnected from NoSQL Database.
```

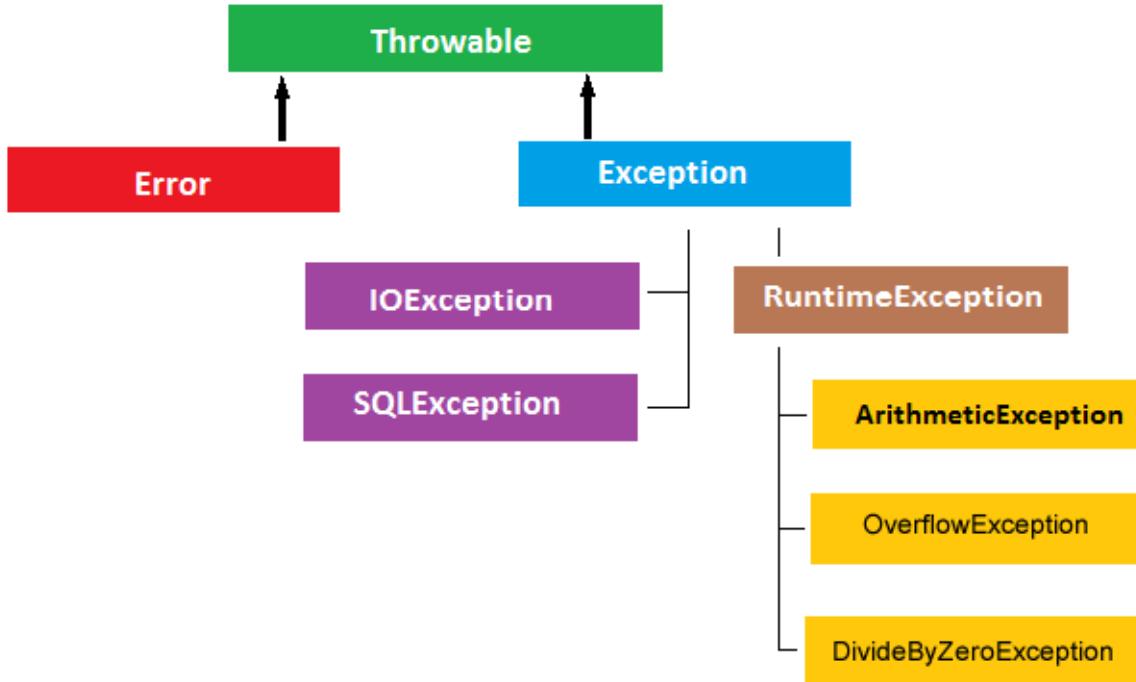
image_133.png

Real-Time Example of Interface in C#: User Authentication Methods

Imagine a scenario where an application allows users to authenticate using different methods, such as a password, fingerprint, or face recognition. Develop a mechanism that can support multiple authentication methods without changing the core authentication process. Let us implement this example using Interface in C#:

The IAuthenticator interface offers a consistent way to implement different authentication methods. The main AuthService class remains decoupled from the specifics of each method. This architecture makes it easy to add or change authentication methods in the future without affecting the core logic of the AuthService.

Exception Handling



image_287.png

This is one of the most important concepts in C#. As a developer, handling an exception is your key responsibility while developing an application. Exception Handling is a procedure for handling an exception that occurs during the execution of a program. As part of this article, we will discuss the following pointers in detail.

Types of Errors in C#

When we write and execute our code in the .NET framework, two types of error occurrences are possible. They are as follows:

- Compilation Errors
- Runtime Errors

Compilation Error in C#

The error that occurs in a program during compilation is known as a compilation error (compile-time error). These errors occur due to syntactical mistakes in the program. These errors occur by typing the wrong syntax, like missing double quotes in a string value and missing terminators in a statement, typing wrong spelling for keywords, assigning wrong data to a variable, trying to create an object for abstract class and interface, etc.

So, whenever we compile the program, the compiler recognizes these errors and shows us the list of errors.

Runtime Error in C#

The errors that occur at the time of program execution are called runtime errors. These errors occur at runtime due to various reasons, such as when we are entering the wrong data into a variable, trying to open a file for which there is no permission, trying to connect to the database with the wrong user ID and password, the wrong implementation of logic, and missing required resources, etc.

So, in simple words, we can say that the errors that come while running the program are called runtime errors.

What is an Exception

An Exception is a class in C# that is responsible for abnormal program termination when runtime errors occur while running the program. These errors (runtime) are very dangerous because whenever they occur, the program terminates abnormally on the same line where the error occurs without executing the next line of code.



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

Who is Responsible for the Abnormal Termination of the Program whenever Runtime Errors occur?

Objects of Exception classes are responsible for abnormal termination of the program whenever runtime errors occur. These exception classes are predefined under BCL (Base Class Libraries), where a separate class is provided for every different type of exception, like

- IndexOutOfRangeException
- FormatException
- NullReferenceException
- DivideByZeroException
- FileNotFoundException
- SQLException,
- OverFlowException, etc.

Each exception class provides a specific exception error message. All the above exception classes are responsible for abnormal program termination and will display an error message that specifies the reason for abnormal termination, i.e., they provide an error message specific to that error.



Exception class is the superclass of all Exception classes in C#.

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

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

The following example shows program execution with an exception. As you can see in the code below, we are dividing an integer number by 0, which is impossible in mathematics. So, it will throw the Divide By Zero Exception in this case. The statements present before the exception-causing statement, i.e., before `c = a / b`, are executed, and the statements present after the exception-causing statement will not be executed.

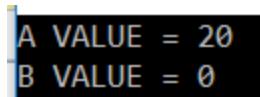
```
using System;
namespace ExceptionHandlingDemo
{
    class Program
    {
        static void Main(string[] args)
```

```

    {
        int a = 20;
        int b = 0;
        int c;
        Console.WriteLine("A VALUE = " + a);
        Console.WriteLine("B VALUE = " + b);
        c = a / b;
        Console.WriteLine("C VALUE = " + c);
        Console.ReadKey();
    }
}

```

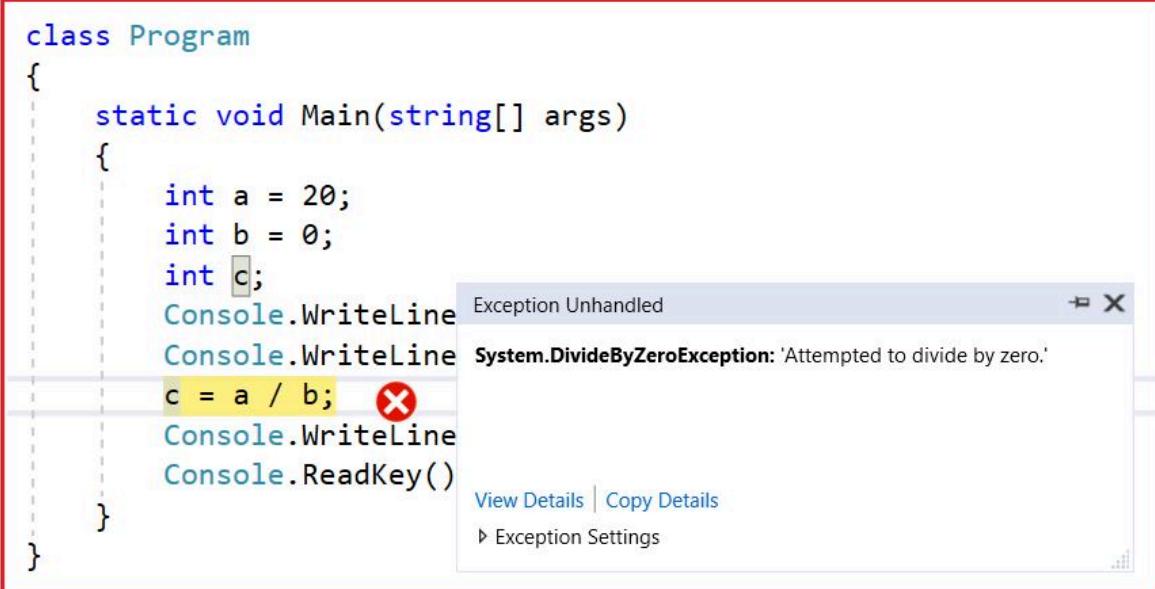
Output:



```
A VALUE = 20
B VALUE = 0
```

image_135.png

After printing the above value, we will get the error below.



image_136.png

The CLR terminates the program execution by throwing DivideByZeroException because the logical mistake we committed here is dividing an integer number by integer zero. As

we know, dividing an integer number by zero is impossible. So, what CLR will do in this case, first it will check what type of logical error is this.

How can we handle an Exception in .NET

There are two methods to handle the exception in .NET

- Logical Implementation
- Try Catch Implementation

What is the Logical Implementation in C# to Handle Exception?

In a logical implementation, we need to handle exceptions using logical statements. In real-time programming, the first and foremost importance is always given to logical implementation only. If it is impossible to handle an exception using logical implementation, then we need to go for try-catch implementation.

Handling Exceptions in C# using Logical Implementation

The following example shows how to handle exceptions in C# using logical implementation. Here, we are checking the second number, i.e., the variable Number2 value. If it equals 0, then we are printing one message saying the second number should not be zero. Otherwise, if the second number is not zero, then we are performing our division operation and showing the results on the console. Here, we are using an IF-ELSE logical statement to handle the exception.

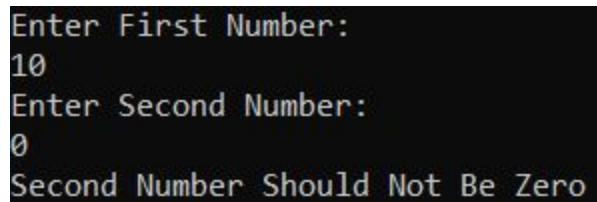
```
using System;
namespace ExceptionHandlingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int Number1, Number2, Result;
            Console.WriteLine("Enter First Number:");
            Number1 = int.Parse(Console.ReadLine());
            Console.WriteLine("Enter Second Number:");
            Number2 = int.Parse(Console.ReadLine());
```

```

        if (Number2 == 0)
    {
        Console.WriteLine("Second Number Should Not Be Zero");
    }
    else
    {
        Result = Number1 / Number2;
        Console.WriteLine($"Result = {Result}");
    }
    Console.ReadKey();
}
}

```

Output:



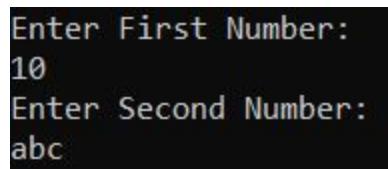
```

Enter First Number:
10
Enter Second Number:
0
Second Number Should Not Be Zero

```

image_137.png

In the above case, when the user enters the second number, a zero exception will be raised, and that is handled using the logical implementation in C#. But while we are entering two numbers, if we enter any character instead of a number, then it will give you one exception, which is FormatException, which is not handled in this program, as shown below.



```

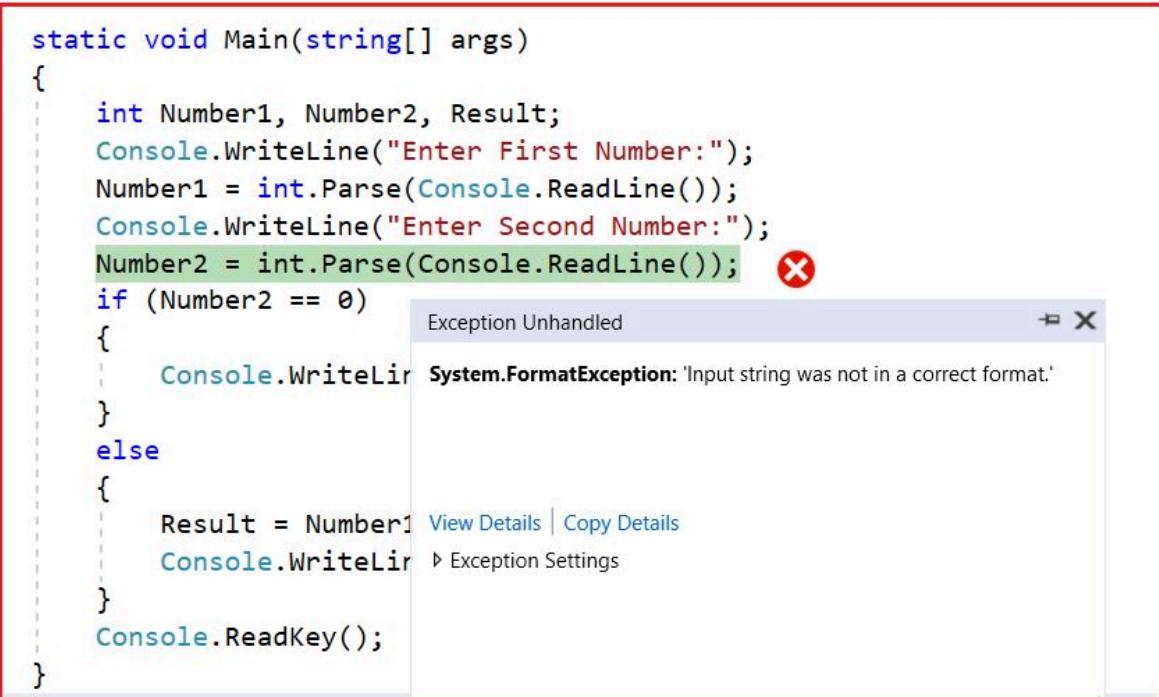
Enter First Number:
10
Enter Second Number:
abc

```

image_138.png

Here we entered the second value as “abc”. So, it will give us the below exception.

```
static void Main(string[] args)
{
    int Number1, Number2, Result;
    Console.WriteLine("Enter First Number:");
    Number1 = int.Parse(Console.ReadLine());
    Console.WriteLine("Enter Second Number:");
    Number2 = int.Parse(Console.ReadLine()); X
    if (Number2 == 0)
    {
        Console.WriteLine System.FormatException: 'Input string was not in a correct format.'
    }
    else
    {
        Result = Number1
        Console.WriteLine Exception Settings
    }
    Console.ReadKey();
}
```



image_139.png

To handle such types of exceptions in C#, we need to implement a try-catch. The point that you need to remember is that if we are unable to handle the exception using logical implementation, we need to implement a **try-catch**.

Exception handling in C# using Try Catch implementation

The .NET framework provides three keywords to implement the try-catch implementation. They are as follows:

- Try : The try keyword establishes a block in which we need to write the exception causing and its related statements. That means exception-causing statements and the related statements, which we should not execute when an exception occurs, must be placed in the try block.
- Catch : The catch block is used to catch the exception that is thrown from its corresponding try block. It has the logic to take necessary actions on that caught exception. The Catch block syntax in C# looks like a constructor. It does not take accessibility modifiers, normal modifiers, or return types. It takes only a single parameter of type Exception or any child class of the parent Exception class

- finally : The keyword finally establishes a block that definitely executes the statements placed in it irrespective of whether any exception has occurred or not. That means the statements that are placed in finally block are always going to be executed irrespective of whether any exception is thrown or not, irrespective of whether the thrown exception is handled by the catch block or not

Syntax to use Exception Handling

The following image shows the syntax for using exception handling in C#. It starts with the try block, followed by the catch block, and writing the finally block is optional.

```
void SomeMethod()
{
    //Normal Statements
    try
    {
        //Exception causing and its related statement which
        //we don't want to execute when exception occurred
    }
    catch (SomeException1 one)
    {
        //Statements which required to execute when SomeException1
        //Occurred in the program
    }
    catch (SomeException2 two)
    {
        //Statements which required to execute when SomeException2
        //Occurred in the program
    }
    catch (Exception generic)
    {
        //This is generic exception handling block. If none of the above
        //Exception catch block handles the exception, then this catch
        //block is going to be execute
    }
    //Normal Statements
}
```

image_140.png

Exception Handling in C# using Try-Catch Implementation with Exception Catch Block

In the below example, we have created a catch block that takes the Exception class as a parameter. Within the catch block, we print the exception information using the

Exception class properties, i.e., Message, Source, StackTrace, and HelpLink. As you can see in the below code, we are using the super Exception class.

```
using System;
namespace ExceptionHandlingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int Number1, Number2, Result;
            try
            {
                Console.WriteLine("Enter First Number:");
                Number1 = int.Parse(Console.ReadLine());
                Console.WriteLine("Enter Second Number:");
                Number2 = int.Parse(Console.ReadLine());
                Result = Number1 / Number2;
                Console.WriteLine($"Result = {Result}");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Message: {ex.Message}");
                Console.WriteLine($"Source: {ex.Source}");
                Console.WriteLine($"HelpLink: {ex.HelpLink}");
                Console.WriteLine($"StackTrace: {ex.StackTrace}");
            }
            Console.ReadKey();
        }
    }
}
```

Output1: Enter the value as 10 and 0

```
Enter First Number:  
10  
Enter Second Number:  
0  
Message: Attempted to divide by zero.  
Source: ExceptionHandlingDemo  
HelpLink:  
StackTrace:    at ExceptionHandlingDemo.Program.Main(String[] args) in D:\Projects\ExceptionHandlingDemo\ExceptionHandlingDemo\Program.cs:line 15
```

image_141.png

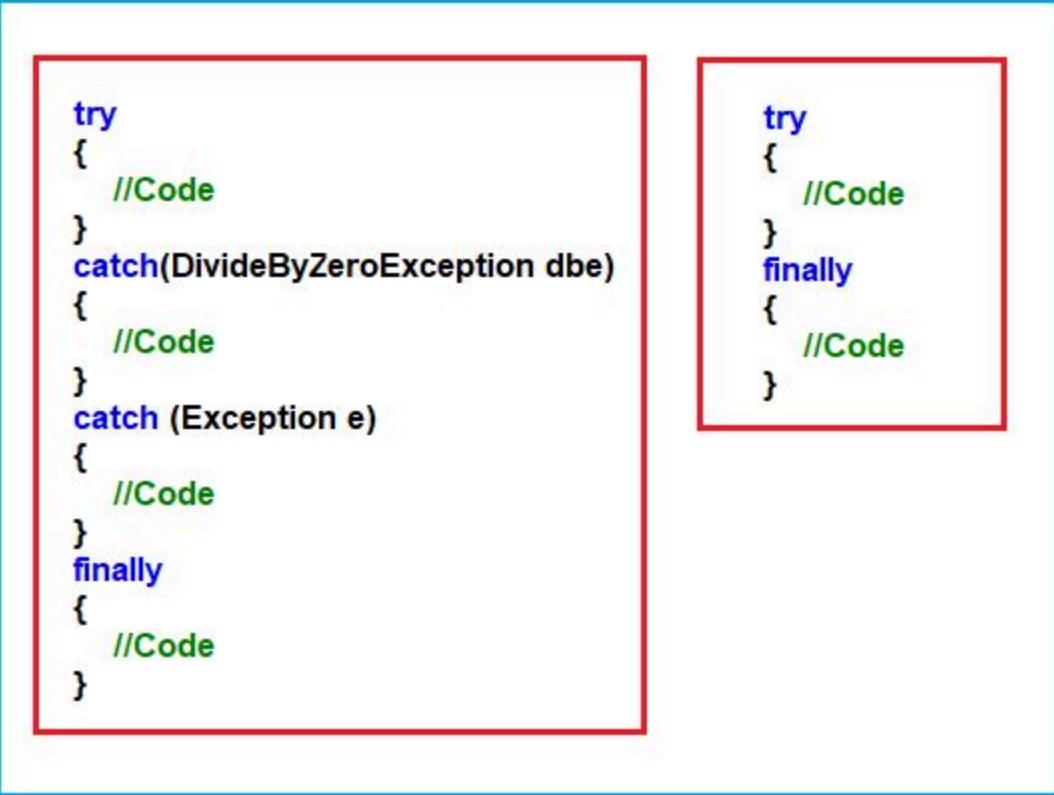
Output2: Enter the value as 10 and abc

```
Enter First Number:  
10  
Enter Second Number:  
abc  
Message: Input string was not in a correct format.  
Source: mscorelib  
HelpLink:  
StackTrace:    at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer & number, NumberFormatInfo info, Boolean parseDecimal)  
    at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)  
    at System.Int32.Parse(String s)  
    at ExceptionHandlingDemo.Program.Main(String[] args) in D:\Projects\ExceptionHandlingDemo\ExceptionHandlingDemo\Program.cs:line 14
```

image_142.png

The Finally Block

The keyword finally establishes a block that definitely executes the statements placed in it irrespective of whether any exception has occurred or not.



image_143.png

Example to Understand the use of finally block

```
using System;
namespace ExceptionHandlingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int Number1, Number2, Result;
            try
            {
                Console.WriteLine("Enter First Number");
                Number1 = int.Parse(Console.ReadLine());
                Console.WriteLine("Enter Second Number");
                Number2 = int.Parse(Console.ReadLine());
                Result = Number1 / Number2;
                Console.WriteLine($"Result: {Result}");
            }
            finally
            {
                // Clean up code here
            }
        }
    }
}
```

```
        }
        catch (DivideByZeroException DBZE)
        {
            Console.WriteLine("Second Number Should Not Be Zero");
        }
        catch (FormatException FE)
        {
            Console.WriteLine("Enter Only Integer Numbers");
        }
        finally
        {
            Console.WriteLine("Hello this is finally block...");
        }
        Console.ReadKey();
    }
}
```

Output:

Output1

```
Enter First Number  
100  
Enter Second Number  
10  
Result: 10  
Hello this is finally block...
```

Output2

```
Enter First Number  
100  
Enter Second Number  
0  
Second Number Should Not Be Zero  
Hello this is finally block...
```

Output3

```
Enter First Number  
100  
Enter Second Number  
abc  
Enter Only Integer Numbers  
Hello this is finally block...
```

image_144.png

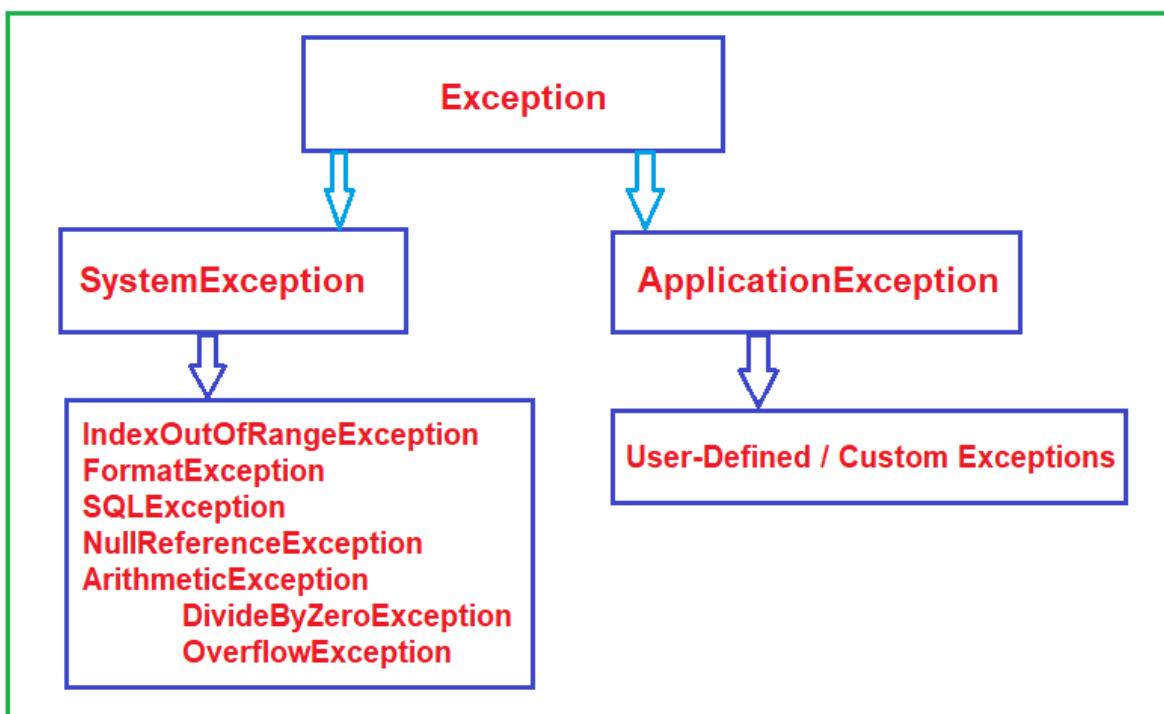
Custom Exceptions

Types of Exceptions in C#:

Before understanding how to create Custom Exceptions or used Defined Exceptions in C#, let us first understand what are the different types of Exceptions available. In C#, the exceptions are divided into two types. They are as follows:

- System Exception: These exceptions are caused by the CLR.
- Application Exception: These exceptions are caused by the programmer.

For a better understanding, please have a look at the below image. The Exception is the parent class of all Exception classes. From the Exception class, SystemException and ApplicationException classes are defined.



image_145.png

What are System Exceptions?

An exception that is raised implicitly under a program at runtime by the Exception Manager (Component of CLR) because of some logical mistakes (some predefined

conditions) is known as System Exception. For example, if you are diving an integer number by zero, then one system exception is raised called DivideByZero. Similarly, if you are taking an alphanumeric string value from the user and trying to store that value in an integer variable, then one system exception is raised called FormatException.

1. DivideByZeroException
2. IndexOutOfRangeException
3. FormatException
4. SQLException
5. NullReferenceException, Etc.

What are Application Exceptions

An exception that is raised explicitly under a program based on our own condition (i.e. user-defined condition) is known as an application exception. As a programmer, we can raise application exceptions at any given point in time. For example, our requirement is that while performing the division operation, we need to check that if the second number is an odd number, then we need to throw an exception.

To raise an Application Exception in C#, we need to adopt the following process. First, we need to create a custom Exception class by inheriting it from the Parent Exception class and then we need to create an instance of the Custom Exception class and then we need to throw that instance.

Step1: Create one Custom Exception Class

```
public class OddNumberException : Exception  
{}
```

Step2: Create an instance of the Custom Exception Class

```
OddNumberException ONE = new OddNumberException();
```

Step3: Throw the Custom Exception Instance

```
throw ONE;  
throw new OddNumberException();
```

image_146.png

Why do we need to create Custom Exceptions in C#?

If none of the already existing .NET exception classes serve our purpose then we need to go for custom exceptions in C#.

How to Create Own Custom Exception Class in C#?

Before creating the Custom Exception class, we need to see the class definition of the Exception class as our Custom Exception class is going to be inherited from the parent Exception class. If you go to the definition of Exception class, then you will see the following.

```

...public class Exception : ISerializable, _Exception
{
    ...public Exception();
    ...public Exception(string message);
    ...public Exception(string message, Exception innerException);
    ...protected Exception(SerializationInfo info, StreamingContext context);

    ...public virtual string Source { get; set; }
    ...public virtual string HelpLink { get; set; }
    ...public virtual string StackTrace { get; }
    ...public MethodBase TargetSite { get; }
    ...public Exception InnerException { get; }
    ...public virtual string Message { get; }
    ...public int HResult { get; protected set; }
    ...public virtual IDictionary Data { get; }

    ...protected event EventHandler<SafeSerializationEventArgs> SerializeObjectState;

    ...public virtual Exception GetBaseException();
    ...public virtual void GetObjectData(SerializationInfo info, StreamingContext context);
    ...public Type GetType();
    ...public override string ToString();
}

```

The diagram shows the `Exception` class structure with three main sections highlighted by red boxes and arrows:

- Constructors**: Points to the first four constructor definitions.
- Properties**: Points to the seven property definitions (Source, HelpLink, StackTrace, TargetSite, InnerException, Message, and Data).
- Methods**: Points to the five method definitions (GetBaseException, GetObjectData, GetType, ToString, and SerializeObjectState).

image_147.png

Now, to create a Custom Exception class in C#, we need to follow the below steps.

- Step1: Define a new class inheriting from the predefined **Exception class** so that the new class also acts as an Exception class.
- Step2: Then as per your requirement, **override the virtual members that are defined inside the Exception class like Message, Source, StackTrace, etc** with the required error message.

```

using System;
namespace ExceptionHandlingDemo
{
    //Creating our own Exception Class by inheriting Exception class
    public class OddNumberException : Exception
    {
        //Overriding the Message property
        public override string Message
        {
            get
            {

```

```

        return "Divisor Cannot be Odd Number";
    }
}

//Overriding the HelpLink Property
public override string HelpLink
{
    get
    {
        return "Get More Information from here:  

https://dotnettutorials.net/lesson/create-custom-exception-csharp/";
    }
}
}
}

```

Now, as per our business logic, we can explicitly create an instance of the OddNumberException class and we can explicitly throw that instance from our application code. For a better understanding, please have a look at the following code. Here, inside the Main method, we are taking two numbers from the user and then checking if the second number is odd or not. If the second number i.e. divisor is odd, then we are creating an instance of the OddNumberException class and throwing that instance.

```

using System;
namespace ExceptionHandlingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int Number1, Number2, Result;
            try
            {
                Console.WriteLine("Enter First Number:");
                Number1 = int.Parse(Console.ReadLine());

```

```

        Console.WriteLine("Enter Second Number:");
        Number2 = int.Parse(Console.ReadLine());

        if (Number2 % 2 > 0)
        {
            //OddNumberException ONE = new OddNumberException();
            //throw ONE;
            throw new OddNumberException();
        }
        Result = Number1 / Number2;
        Console.WriteLine(Result);
    }
    catch (OddNumberException one)
    {
        Console.WriteLine($"Message: {one.Message}");
        Console.WriteLine($"HelpLink: {one.HelpLink}");
        Console.WriteLine($"Source: {one.Source}");
        Console.WriteLine($"StackTrace: {one.StackTrace}");
    }

    Console.WriteLine("End of the Program");
    Console.ReadKey();
}
}

```

Output:

```

10
Enter Second Number:
3
Message: Divisor Cannot be Odd Number
HelpLink: Get More Information from here: https://dotnettutorials.net/lesson/create-custom-exception-c
sharp/
Source: ExceptionHandlingDemo
StackTrace:   at ExceptionHandlingDemo.Program.Main(String[] args) in D:\Projects\ExceptionHandlingDe
mo\ExceptionHandlingDemo\Program.cs:line 21
End of the Program

```

image_148.png

Collections in C#

Collections are nothing but groups of records that can be treated as one logical unit.

For example, you can have a country collection that can have a country code and country name. You can have a product collection that has the Product Id and Product Name. You can have an employee collection having the employee's name and employee id. You can have a book collection having an ISBN number and book name. For a better understanding, please have a look at the below image.

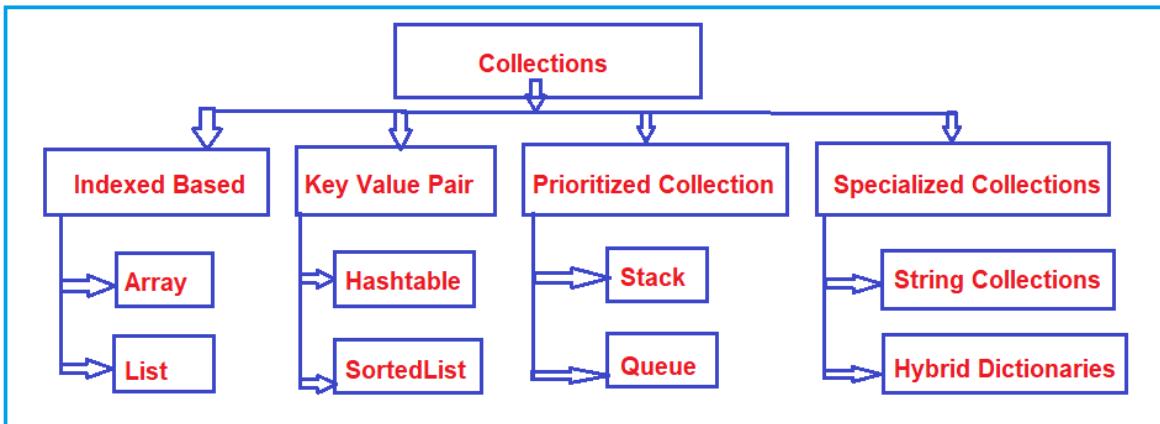
Country Collection		Product Collection	
Country Code	Country Name	Product Id	Product Name
IND	India	10001	Laptop
UK	United Kingdom	10002	Desktop
SL	Srilanka	10003	Tab

Employee Collection		Book Collection	
EmployeeId	EmployeeName	ISBN Number	Name
12345	James	10293847	C#.NET
12346	Smith	12340987	Linq
12347	Sara	13579753	.NET Core

image_157.png

General Categories of Collections:

Collections are classified into 4 types such as **Indexed Based**, **Key-Value Pair**, **Prioritized Collection**, and **Specialized Collections**. For a better understanding, please have a look at the below diagram.



image_158.png

Indexed Base Collections:

In Indexed Based, we have two kinds of collections i.e. Array and List. To understand the Indexed Based collection, please have a look at the following country collection. So, when we add any elements to .NET Collections like Array, List, or ArrayList, it maintains its own internal index number.

Country Collections		
Internal_Index	Country_Code	Country_Name
0	IND	India
1	UK	United Kingdom
2	SL	Srilanka
3	USA	United State of America
4	AUS	Australia

Array[0] = India
List[0] = India

image_159.png

Key-Value Pair Collections

In the Key-Value Pair collection, we have **Hashtable**, **Dictionary**, and **SortedList**. In real-time projects, we rarely accessed the records using the internal index numbers. We generally use some kind of keys to identify the records and there we use the Key-Value Pair collections like **Hashtable**, **Dictionary**, and **SortedList**.

Country Collections

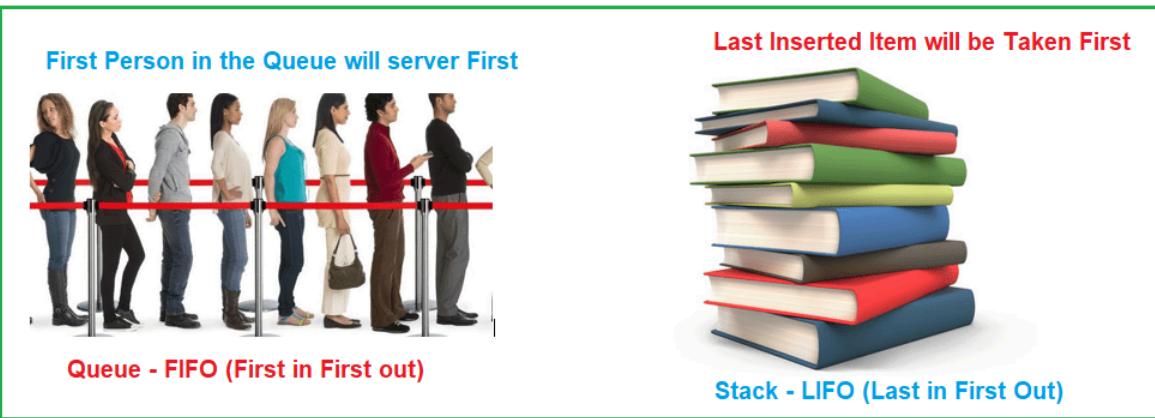
Internal_Index	Country_Code	Country_Name
0	IND	India
1	UK	United Kingdom
2	SL	Srilanka
3	USA	United State of America
4	AUS	Australia

Hashtable ('IND') = India

image_160.png

Prioritized Collections:

The Prioritized Collections help you to access the elements in a particular sequence. The **Stack** and **Queue** collections belong to the Prioritized Collections category. If you want First in First Out (FIFO) access to the elements of a collection, then you need to use Queue collection. On the other hand, if you want Last in First Out (LIFO) access to the elements of a collection, then you need to use the Stack collection.



image_161.png

Specialized Collections: The Specialized Collections are specifically designed for a specific purpose. For example, a Hybrid Dictionary starts as a list and then becomes a hashtable.

Array

The array is defined as a collection of similar data elements. If you have some sets of integers, and some sets of floats, you can group them under one name as an array. So, in simple words, **we can define an array as a collection of similar types of values that are stored in a contiguous memory location under a single name.**

```
int[] employeno = { 1, 2, 3, 4, 5};
```

How does using [] this work in real memory?



image_149.png

See using this [] after the data type, you are informing the compiler that the variable is an array and allocating a block of memory as specified by the array.

Types of Arrays in C#:

C# supports 2 types of arrays. They are as follows:

- Single dimensional array
- Multi-dimensional array

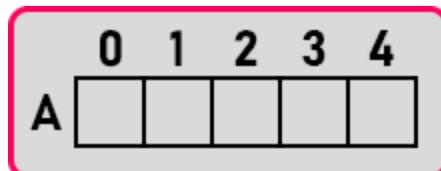
In the Single dimensional array, the data is arranged in the form of a row whereas in the multi-dimensional arrays, the data is arranged in the form of rows and columns.

How to declare an Array

General Syntax: <data type>[] VariableName = new <data type>[size of the array];

Example: int[] A = new int[5];

Here we have created an array with the name A and with a size of 5. So, you can store 5 values with the same name A. How it looks in the memory? It will allocate memory for 5 integers. These all 5 are types of int. For that memory, indexing will start from 0 onwards as shown in the below image.



image_150.png

Assigning Values to Array

Thus, the first way of assigning values to the elements of an array is by doing so at the time of its declaration i.e. `int[] n={1,2,3};` And the second method is declaring the array first and then assigning values to its elements as shown below.

```
int[] n = new int[3]; //Declaring an Array

//Initializing the array elements
n[0] = 10;
n[1] = 20;
n[2] = 30;
```

image_151.png

Creating and Initializing an Array at the Same Statement

```
using System;
namespace ArayDemo
{
    class Program
    {
        static void Main(string[] args)
        {
```

```

//Creating and Initializing an Array
//Here, the size will be decided based on the number of
elements
    //In this case size will be 3
    int[] Numbers = { 10, 20, 30 };

    //Accessing the Array Elements separately
    Console.WriteLine("Accessing the Array Elements
separately");
    Console.WriteLine($"Numbers[0] = {Numbers[0]}");
    Console.WriteLine($"Numbers[1] = {Numbers[1]}");
    Console.WriteLine($"Numbers[2] = {Numbers[2]}");

    //Accessing the Array Elements using for Loop
    Console.WriteLine("\nAccessing the Array Elements using For
Loop");
    for (int i = 0; i <= Numbers.Length - 1; i++)
    {
        Console.WriteLine($"Numbers[{i}] = {Numbers[i]}");
    }

    //Accessing the Array Elements using foreach Loop
    Console.WriteLine("\nAccessing the Array Elements using
ForEach Loop");
    foreach (int Number in Numbers)
    {
        Console.WriteLine($"{Number}");
    }
    Console.ReadKey();
}
}

```

Output:

```
Accessing the Array Elements separately
Numbers[0] = 10
Numbers[1] = 20
Numbers[2] = 30

Accessing the Array Elements using For Loop
Numbers[0] = 10
Numbers[1] = 20
Numbers[2] = 30

Accessing the Array Elements using ForEach Loop
10
20
30
```

image_152.png

Creating and Initializing an Array Separately

```
using System;
namespace ArayDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            //Creating an Integer Array with size 3
            int[] Numbers = new int[3];

            //Accessing the Array Elements Before Initialization
            Console.WriteLine("Accessing the Array Elements Before
Initialization");
            for (int i = 0; i <= Numbers.Length - 1; i++)
            {
                Console.WriteLine($"Numbers[{i}] = {Numbers[i]}");
            }
        }
    }
}
```

```

    //Initializing the Array Elements using the Index Position
    Numbers[0] = 10;
    Numbers[1] = 20;
    Numbers[2] = 30;

    //Accessing the Array Elements After Initialization
    Console.WriteLine("\nAccessing the Array Elements After
Initialization");
    for (int i = 0; i <= Numbers.Length - 1; i++)
    {
        Console.WriteLine($"Numbers[{i}] = {Numbers[i]}");
    }

    Console.ReadKey();
}
}

```

Output:

```

Accessing the Array Elements Before Initialization
Numbers[0] = 0
Numbers[1] = 0
Numbers[2] = 0

Accessing the Array Elements After Initialization
Numbers[0] = 10
Numbers[1] = 20
Numbers[2] = 30

```

image_153.png

Array Class in C#

The Array class in C# is a predefined class that is defined inside the System namespaces. This class is working as the base class for all the Arrays in C#. The Array class provides a set of members (methods and properties) to work with the arrays such as creating, manipulating, searching, reversing, and sorting the elements of an array, etc. The definition of the Array class in C# is given below.

The Array Class in C# is not a part of the **System.Collections** namespace. It is a part of the System namespace. But still, we considered it as a collection because it Implement the **IList interface**.

1. **Sort()**: Sorting the array elements
2. **Reverse ()**: Reversing the array elements
3. **Copy (src, dest, n)**: Copying some of the elements or all elements from the old array to the new array
4. **GetLength(int)**: A 32-bit integer that represents the number of elements in the specified dimension.
5. **Length**: It Returns the total number of elements in all the dimensions of the Array; zero if there are no elements in the array.

Example to Understand Array Class Methods and Properties in C#

```
using System;
namespace ArayDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            //Creating and Initializing an Array of Integers
            //Size of the Array is 10
            int[] Numbers = { 17, 23, 4, 59, 27, 36, 96, 9, 1, 3 };

            //Printing the Array Elements using a for Loop
            Console.WriteLine("Original Array Elements :");
            for (int i = 0; i < Numbers.Length; i++)
            {
                Console.Write(Numbers[i] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

```

        //Sorting the Array Elements by using the Sort method of
Array Class
        Array.Sort(Numbers);
        //Printing the Array Elements After Sorting using a foreach
loop
        Console.WriteLine("\nArray Elements After Sorting :");
        foreach (int i in Numbers)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();

        //Reversing the array elements by using the Reverse method
of Array Class
        Array.Reverse(Numbers);
        //Printing the Array Elements in Reverse Order
        Console.WriteLine("\nArray Elements in the Reverse Order
:");
        foreach (int i in Numbers)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();

        //Creating a New Array
        int[] NewNumbers = new int[10];
        //Copying Some of the Elements from Old array to new array
        //We declare the array with size 10 and we copy only 5
elements.
        //So the rest 5 elements will take the default value. In
this array, it will take 0
        Array.Copy(Numbers, NewNumbers, 5);

        //Printing the Array Elements using for Each Loop
        Console.WriteLine("\nNew Array Elements :");
        foreach (int i in NewNumbers)
        {

```

```

        Console.WriteLine(i + " ");
    }

    Console.WriteLine();
    Console.WriteLine($"\\nNew Array Length using Length Property
:{NewNumbers.Length}");
    Console.WriteLine($"New Array Length using GetLength Method
:{NewNumbers.GetLength(0)}");
    Console.ReadKey();
}
}
}

```

Output:

```

Original Array Elements :
17 23 4 59 27 36 96 9 1 3

Array Elements After Sorting :
1 3 4 9 17 23 27 36 59 96

Array Elements in the Reverse Order :
96 59 36 27 23 17 9 4 3 1

New Array Elements :
96 59 36 27 23 0 0 0 0 0

New Array Length using Length Property :10
New Array Length using GetLength Method :10

```

image_154.png

2Dimensional Array in C#:

Let us understand how to initialize a 2D Array with an example. Please have a look at the following statement which shows the declaration and initialization of a 2D Array.

```
int[,] A = {{2, 5, 9},{6, 9, 15}};
```

This is the declaration + initialization of a 2Dimensional array in C#. Here 2,5,9 is the 1st row and 6,9,15 is the 2nd row. This is how they will be filled and we can access any

element with the help of two indexes that is row number and column number. Now, the other way of initializing it is,

```
int[,] A = new int[2,3]
{
    {2, 5, 9},{6, 9, 15}
};
```

Accessing the Elements of the 2D array

For accessing all the elements of the rectangle 2D Array in C#, we require a nested for loop, one for loop for accessing the rows, and another for loop for accessing the columns.

```
//2D Array with 3 Rows and 3 Columns
int[,] A = new int[2, 3]
{
    {2, 5, 9},{6, 9, 15}
};

//Accessing Array Elements using nested for loop
for (int i = 0; i < 2; i++) //Accessing the Rows
{
    for (int j = 0; j < 3; j++) //Accessing the Columns
    {
        Console.WriteLine(A[i, j]); //Accessing the Array Elements
    }
}
```

image_155.png

```
using System;
namespace TwoDimensionalArrayDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            //Creating a 2D Array with 4 Rows and 5 Columns
```

```

int[,] RectangleArray = new int[4, 5];
int a = 0;

//Printing the values of 2D array using foreach loop
//It will print the default values as we are not assigning
//any values to the array
foreach (int i in RectangleArray)
{
    Console.WriteLine(i + " ");
}
Console.WriteLine("\n");

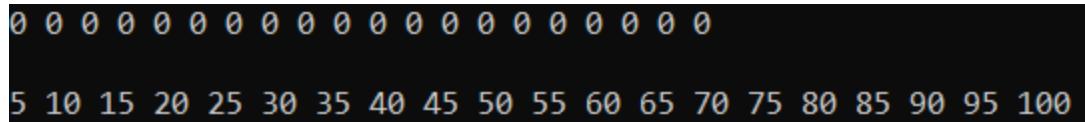
//Assigning values to the 2D array by using nested for loop
//arr.GetLength(0): Returns the size of the Row
//arr.GetLength(1): Returns the size of the Column
for (int i = 0; i < RectangleArray.GetLength(0); i++)
{
    for (int j = 0; j < RectangleArray.GetLength(1); j++)
    {
        a += 5;
        RectangleArray[i, j] = a;
    }
}

//Printing the values of array by using nested for loop
//arr.GetLength(0): Returns the size of the Row
//arr.GetLength(1): Returns the size of the Column
for (int i = 0; i < RectangleArray.GetLength(0); i++)
{
    for (int j = 0; j < RectangleArray.GetLength(1); j++)
    {
        Console.Write(RectangleArray[i, j] + " ");
    }
}
Console.ReadKey();
}

```

```
    }  
}
```

Output:



```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
```

image_156.png

Advantages of using an Array

1. It is used to store similar types of multiple data items using a single name.
2. We can use arrays to implement other data structures such as linked lists, trees, graphs, stacks, queues, etc.
3. The two-dimensional arrays in C# are used to represent matrices.
4. The Arrays in C# are strongly typed. That means they are used to store similar types of multiple data items using a single name.

Limitations of Array in C#:

1. The array size is fixed. Once the array is created we can never increase the size of the array. If we want then we can do it manually by creating a new array and copying the old array elements into the new array or by using the `Array` class `Resize` method which will do the same thing means to create a new array and copy the old array elements into the new array and then destroy the old array.
2. We can never insert an element into the middle of an array
3. Deleting or removing elements from the middle of the array.

ArrayList

The ArrayList in C# is a non-generic collection class that works like an array but provides the facilities such as dynamic resizing, adding, and deleting elements from the middle of a collection.

Properties of ArrayList Class in C#:

1. The Elements can be added and removed from the Array List collection at any point in time.
2. The ArrayList is not guaranteed to be sorted.
3. The capacity of an ArrayList is the number of elements the ArrayList can hold.
4. Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.
5. It allows duplicate elements.

How to Add Elements into ArrayList in C#?

The ArrayList non-generic class provides the Add() method which we can use to add elements to the array list or even we can use the object initializer syntax to add elements in an ArrayList. The most important point is that we can add multiple different types of elements in an ArrayList even though it is also possible to add duplicate elements.

```
using System;
using System.Collections;
namespace Csharp8Features
{
    public class ArrayListDemo
    {
        public static void Main()
        {
            //Adding elements to ArrayList using Add() method
            ArrayList arrayList1 = new ArrayList();
            arrayList1.Add(101); //Adding Integer Value
```

```

arrayList1.Add("James"); //Adding String Value
arrayList1.Add("James"); //Adding Duplicate Value
arrayList1.Add(" "); //Adding Empty
arrayList1.Add(true); //Adding Boolean
arrayList1.Add(4.5); //Adding double
arrayList1.Add(null); //Adding null

foreach (var item in arrayList1)
{
    Console.WriteLine(item);
}

//Adding Elements to ArrayList using object initializer
syntax
var arrayList2 = new ArrayList()
{
    102, "Smith", "Smith", true, 15.6
};

foreach (var item in arrayList2)
{
    Console.WriteLine(item);
}
Console.ReadKey();
}
}

```

Output:

```
101
James
James

True
4.5

102
Smith
Smith
True
15.6
```

image_162.png

How to Access an ArrayList in C#?

If you go to the definition of ArrayList, then you will see that the ArrayList class implements the IList interface as shown in the below image. As it implements the IList interface, so we can access the elements using an indexer, in the same way as an array. The index starts from zero and increases by one for each subsequent element. So, when a collection class implements the IList interface, then we can access the elements of that collection by using the integer indexes.

```
//
// Summary:
//     Implements the System.Collections.IList interface using an array whose size is
//     dynamically increased as required.
[DefaultMember("Item")]
public class ArrayList : ICollection, IEnumerable, IList, ICloneable
{
```

image_163.png

```
using System;
using System.Collections;
namespace Csharp8Features
{
    public class ArrayListDemo
    {
        public static void Main()
        {
```

```

//Adding elements to ArrayList using Add() method
ArrayList arrayList1 = new ArrayList();
arrayList1.Add(101); //Adding Integer Value
arrayList1.Add("James"); //Adding String Value
arrayList1.Add(true); //Adding Boolean
arrayList1.Add(4.5); //Adding double

//Accessing individual elements from ArrayList using Indexer
int firstElement = (int)arrayList1[0]; //returns 101
string secondElement = (string)arrayList1[1]; //returns
"James"
//int secondElement = (int) arrayList1[1]; //Error: cannot
cover string to int
Console.WriteLine($"First Element: {firstElement}, Second
Element: {secondElement}");

//Using var keyword without explicit casting
var firsItem = arrayList1[0]; //returns 101
var secondItem = arrayList1[1]; //returns "James"
//var fifthElement = arrayList1[5]; //Error: Index out of
range
Console.WriteLine($"First Item: {firsItem}, Second Item:
{secondItem}");

//update elements
arrayList1[0] = "Smith";
arrayList1[1] = 1010;
//arrayList1[5] = 500; //Error: Index out of range

foreach (var item in arrayList1)
{
    Console.Write($"{item} ");
}
Console.ReadKey();
}

}

```

Output:

```
First Element: 101, Second Element: James  
First Item: 101, Second Item: James  
Smith 1010 True 4.5
```

image_164.png

How to Iterate an ArrayList in C#?

If you go to the definition of ArrayList, then you will also see that the `ArrayList` non-generic collection class implements the `ICollection` interface as shown in the below image. And we know the `ICollection` interface supports iteration of the collection types. So, we can either use the `foreach` loop and `for` loop to iterate a collection of type `ArrayList`.

```
//  
// Summary:  
//     Implements the System.Collections.IList interface using an array whose size is  
//     dynamically increased as required.  
[DefaultMember("Item")]  
public class ArrayList : ICollection, IEnumerable, IList, ICloneable  
{
```

image_165.png

The `Count` property of `ArrayList` returns the total number of elements present in an `ArrayList`. Let us understand this with an example.

```
using System;  
using System.Collections;  
namespace Csharp8Features  
{  
    public class ArrayListDemo  
    {  
        public static void Main()  
        {  
            //Adding elements to ArrayList using Add() method  
            ArrayList arrayList1 = new ArrayList();  
            arrayList1.Add(101); //Adding Integer Value  
            arrayList1.Add("James"); //Adding String Value  
            arrayList1.Add(true); //Adding Boolean
```

```

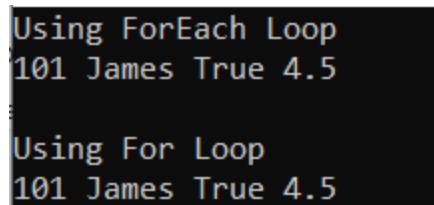
arrayList1.Add(4.5); //Adding double

//Iterating through foreach loop
Console.WriteLine("Using ForEach Loop");
foreach (var item in arrayList1)
{
    Console.Write($"{item} ");
}

//Iterating through for loop
Console.WriteLine("\n\nUsing For Loop");
for (int i = 0; i < arrayList1.Count; i++)
{
    Console.Write($"{arrayList1[i]} ");
}
Console.ReadKey();
}
}

```

Output:



```

Using ForEach Loop
101 James True 4.5

Using For Loop
101 James True 4.5

```

image_166.png

How to Insert an Element into a Specified Position in an ArrayList Collection

The Add method will add the element at the end of the collection i.e. after the last element. But, if you want to add the element at a specified position, then you need to use the **Insert()** method of the ArrayList class which will insert an element into the collection at the specified index position. The syntax to use the Insert method is given below.

```
void Insert(int index, object? value);
```

```
using System;
using System.Collections;
namespace Csharp8Features
{
    public class ArrayListDemo
    {
        public static void Main()
        {
            ArrayList arrayList = new ArrayList()
            {
                101,
                "James",
                true,
                10.20
            };

            //Insert "First Element" at First Position i.e. Index 0
            arrayList.Insert(0, "First Element");

            //Insert "Third Element" at Third Position i.e. Index 2
            arrayList.Insert(2, "Third Element");

            //Iterating through foreach loop
            foreach (var item in arrayList)
            {
                Console.WriteLine($"{item}");
            }
            Console.ReadKey();
        }
    }
}
```

Output:

```
First Element  
101  
Third Element  
James  
True  
10.2
```

image_167.png

How to Remove Elements from ArrayList

If you want to remove elements from ArrayList in C#, then you can use Remove(), RemoveAt(), or RemoveRange() methods of the ArrayList class in C#.

- **Remove(object? obj):** This method is used to remove the first occurrence of a specific object from the System.Collections.ArrayList. The parameter obj specifies the Object to remove from the ArrayList. The value can be null.
- **RemoveAt(int index):** This method is used to remove the element at the specified index of the ArrayList. The parameter index specifies the index position of the element to remove.
- **RemoveRange(int index, int count):** This method is used to remove a range of elements from the ArrayList. The parameter index specifies the starting index position of the range of elements to remove and the parameter count specifies the number of elements to remove.

```
using System;  
using System.Collections;  
namespace Csharp8Features  
{  
    public class ArrayListDemo  
    {  
        public static void Main()  
        {  
            ArrayList arrayList = new ArrayList()  
            {  
                "India",  
                "USA",  
                "China",  
                "Brazil",  
                "Germany"  
            };  
            arrayList.RemoveAt(2);  
            foreach (object item in arrayList)  
            {  
                Console.WriteLine(item);  
            }  
        }  
    }  
}
```

```

        "UK",
        "Nepal",
        "HongKong",
        "Srilanka",
        "Japan",
        "Britem",
        "HongKong",
    };

    Console.WriteLine("Array List Elements");
    foreach (var item in arrayList)
    {
        Console.Write($"{item} ");
    }

arrayList.Remove("HongKong"); //Removes first occurrence of
null
    Console.WriteLine("\n\nArray List Elements After Removing
First Occurrences of HongKong");
    foreach (var item in arrayList)
    {
        Console.Write($"{item} ");
    }

arrayList.RemoveAt(3); //Removes element at index position 3,
it is 0 based index
    Console.WriteLine("\n\nArray List1 Elements After Removing
Element from Index 3");
    foreach (var item in arrayList)
    {
        Console.Write($"{item} ");
    }

arrayList.RemoveRange(0, 2); //Removes two elements starting
from 1st item (0 index)
    Console.WriteLine("\n\nArray List Elements After Removing
First Two Elements");
    foreach (var item in arrayList)

```

```

        {
            Console.WriteLine($"{item} ");
        }
        Console.ReadKey();
    }
}

```

Output:

```

ArrayList Elements
India USA UK Nepal HongKong Srilanka Japan Britem HongKong

ArrayList Elements After Removing First Occurrences of HongKong
India USA UK Nepal Srilanka Japan Britem HongKong

ArrayList1 Elements After Removing Element from Index 3
India USA UK Srilanka Japan Britem HongKong

ArrayList Elements After Removing First Two Elements
UK Srilanka Japan Britem HongKong

```

image_168.png

How to Remove all the elements from the ArrayList

```

using System;
using System.Collections;
namespace Csharp8Features
{
    public class ArrayListDemo
    {
        public static void Main()
        {
            ArrayList arrayList = new ArrayList()
            {
                "India",
                "USA",
                "UK",
                "Denmark",
            }
            arrayList.Clear();
            foreach (var item in arrayList)
            {
                Console.WriteLine($"{item} ");
            }
            Console.ReadKey();
        }
    }
}

```

```
        "Nepal",  
    };  
  
    int totalItems = arrayList.Count;  
    Console.WriteLine(string.Format($"Total Items: {totalItems},  
Capacity: {arrayList.Capacity}"));  
    //Remove all items from the Array list  
    arrayList.Clear();  
  
    totalItems = arrayList.Count;  
    Console.WriteLine(string.Format($"Total Items After Clear():  
{totalItems}, Capacity: {arrayList.Capacity}"));  
    Console.ReadKey();  
}  
}  
}
```

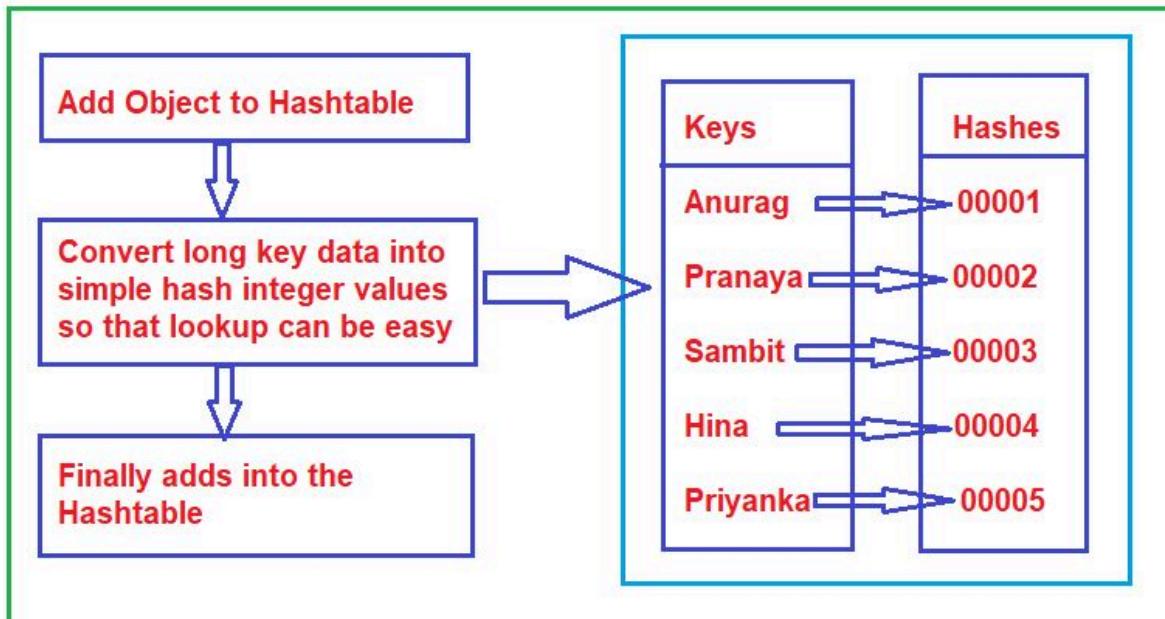
Hashtable

The Hashtable in C# is a Non-Generic Collection that stores the element in the form of “Key-Value Pairs”. The data in the Hashtable are organized based on the hash code of the key. The key in the HashTable is defined by us and more importantly, that key can be of any data type. Once we created the Hashtable collection, then we can access the elements by using the keys.

Hashtable Characteristics in C#

1. The Hashtable Collection Class in C# stores the elements in the form of key-value pairs.
2. Hashtable Class belongs to System.Collection namespace i.e. it is a Non-Generic Collection class.
3. It implements the IDictionary interface.
4. The Keys can be of any data type but they must be unique and not null.
5. The Hashtable accepts both null and duplicate values.
6. We can access the values by using the associated key.
7. The capacity of a Hashtable is the number of elements that a Hashtable can hold.
8. A hashtable is a non-generic collection, so we can store elements of the same type as well as of different types.

One more point that you need to remember is that the performance of the hashtable is less as compared to the ArrayList because of this key conversion (converting the key to an integer hashcode).



image_169.png

How to Add Elements into a Hashtable Collection in C#?

Now, if you want to add elements i.e. a key/value pair into the hashtable, then you need to use Add() method of the Hashtable class.

Add(object key, object? value): The Add(object key, object? value) method is used to add an element with the specified key and value into the Hashtable. Here, the parameter key specifies the key of the element to add and the parameter value specifies the value of the element to add. The value can be null.

For Example

```
hashtable.Add("EId", 1001);
```

```
hashtable.Add("Name", "James");
```

Even it is also possible to create a Hashtable using collection-initializer syntax as follows:

```
var cities = new Hashtable(){
    {"UK", "London, Manchester, Birmingham"}, 
    {"USA", "Chicago, New York, Washington"},
```

```
{“India”, “Mumbai, Delhi, BBSR”}  
};
```

How to access a Non-Generic Hashtable Collection

You can access the individual value of the Hashtable in C# by using the keys. In this case, we need to pass the key as a parameter to find the respective value. If the specified key is not present, then the compiler will throw an exception. The syntax is given below.

```
hashtable[“EId”]  
hashtable[“Name”]
```

Create a Hashtable and Add Elements

```
using System;  
using System.Collections;  
namespace HasntableExample  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            //Creating Hashtable collection with default constructor  
            Hashtable hashtable = new Hashtable();  
  
            //Adding elements to the Hash table using key value pair  
            hashtable.Add("EId", 1001); //Here key is Eid and value is  
1001  
            hashtable.Add("Name", "James"); //Here key is Name and value  
is James  
            hashtable.Add("Salary", 3500); //Here key is Salary and  
value is 3500  
            hashtable.Add("Location", "Mumbai"); //Here key is Location  
and value is Mumbai  
            hashtable.Add("EmailID", "a@a.com"); //Here key is EmailID  
and value is a@a.com  
  
            //Printing the keys and their values using foreach loop
```

```

Console.WriteLine("Printing Hashtable using Foreach Loop");
foreach (object obj in hashtable.Keys)
{
    Console.WriteLine(obj + " : " + hashtable[obj]);
}
//Or
//foreach (DictionaryEntry de in hashtable)
//{
//    Console.WriteLine($"Key: {de.Key}, Value:
{de.Value}");
//}

Console.WriteLine("\nPrinting Hashtable using Keys");
//I want to print the Location by using the Location key
Console.WriteLine("Location : " + hashtable["Location"]);

//I want to print the Email ID by using the EmailID key
Console.WriteLine("EmailID ID : " + hashtable["EmailID"]);

Console.ReadKey();
}
}
}

```

Output:

```

Printing Hashtable using Foreach Loop
EmailID : a@a.com
EId : 1001
Name : James
Location : Mumbai
Salary : 3500

Printing Hashtable using Keys
Location : Mumbai
EmailID ID : a@a.com

```

image_170.png

Add Elements to a Hashtable using Collection Initializer

```
using System;
using System.Collections;

namespace HashtableExample
{
    class Program
    {
        static void Main(string[] args)
        {
            //Creating a Hashtable using collection-initializer syntax
            Hashtable citiesHashtable = new Hashtable(){
                {"UK", "London, Manchester, Birmingham"},
                //Key:UK, Value:London, Manchester, Birmingham
                {"USA", "Chicago, New York, Washington"},
                //Key:USA, Value:Chicago, New York, Washington
                {"India", "Mumbai, New Delhi, Pune"}
                //Key:India, Value:Mumbai, New Delhi, Pune
            };

            Console.WriteLine("Creating a Hashtable Using Collection-
Initializer");
            foreach (DictionaryEntry city in citiesHashtable)
            {
                Console.WriteLine($"Key: {city.Key}, Value:
{city.Value}");
            }

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

Output:

```
Creating a Hashtable Using Collection-Initializer  
Key: India, Value: Mumbai, New Delhi, Pune  
Key: USA, Value: Chicago, New York, Washington  
Key: UK, Value: London, Manchester, Birmingham
```

image_171.png

How to Check the Availability of a key/value Pair in a Hashtable

If you want to check whether the key/value pair exists or not in the Hashtable, then you can use the following methods of the Hashtable class.

- 1. Contains(object key):** The Contains(object key) method of the Hashtable is used to check whether the Hashtable contains a specific key. The parameter key to locating in the Hashtable object. It returns true if the Hashtable contains an element with the specified key; otherwise, false. If the key is null, then it will throw System.ArgumentNullException.
- 2. ContainsKey(object key):** The ContainsKey(object key) method of the Hashtable is used to check if a given key is present in the Hashtable or not. The parameter key to locating in the Hashtable object. If the given key is present in the collection then it will return true else it will return false. If the key is null, then it will throw System.ArgumentNullException.
- 3. ContainsValue(object value):** The ContainsValue(object value) Method of the Hashtable class is used to check if a value is present in the Hashtable or not. The parameter value to locate in the hashtable object. If the given value is present in the collection then it will return true else it will return false.

```
using System;  
using System.Collections;  
  
namespace HashtableExample  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {
```

```

//Creating Hashtable collection with default constructor
Hashtable hashtable = new Hashtable();

//Adding elements to the Hash table using key value pair
hashtable.Add("EId", 1001); //Here key is Eid and value is
1001
    hashtable.Add("Name", "James"); //Here key is Name and value
is James
    hashtable.Add("Job", "Developer");
    hashtable.Add("Salary", 3500);
    hashtable.Add("Location", "Mumbai");
    hashtable.Add("Dept", "IT");
    hashtable.Add("EmailID", "a@a.com");

//Checking the key using the Contains method
Console.WriteLine("Is EmailID Key Exists : " +
hashtable.Contains("EmailID"));
    Console.WriteLine("Is Department Key Exists : " +
hashtable.Contains("Department"));

//Checking the key using the ContainsKey method
Console.WriteLine("Is EmailID Key Exists : " +
hashtable.ContainsKey("EmailID"));
    Console.WriteLine("Is Department Key Exists : " +
hashtable.ContainsKey("Department"));

//Checking the value using the ContainsValue method
Console.WriteLine("Is Mumbai value Exists : " +
hashtable.ContainsValue("Mumbai"));
    Console.WriteLine("Is Technology value Exists : " +
hashtable.ContainsValue("Technology"));

    Console.ReadKey();
}
}
}

```

Output:

```
Is EmailID Key Exists : True
Is Department Key Exists : False
Is EmailID Key Exists : True
Is Department Key Exists : False
Is Mumbai value Exists : True
Is Technology value Exists : False
```

image_172.png

How to Remove Elements from a Non-Generic Hashtable Collection

If you want to remove an element from the Hashtable, then you can use the following Remove method of the C# Hashtable class.

- `Remove(object key)`: This method is used to remove the element with the specified key from the Hashtable. Here, the parameter key specifies the element to remove.
- `Clear()`: This method is used to remove all elements from the Hashtable

```
using System;
using System.Collections;

namespace HashtableExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable employee = new Hashtable
            {
                { "EId", 1001 },
                { "Name", "James" },
                { "Salary", 3500 },
                { "Location", "Mumbai" },
                { "EmailID", "a@a.com" }
            };
        }
    }
}
```

```

        //Count Property returns the number of elements present in
the collection
        Console.WriteLine($"Hashtable Total Elements:
{employee.Count}");

        // Remove EId as this key exists
        employee.Remove("EId");
        Console.WriteLine($"After Removing EID Total Elements:
{employee.Count}");

        //Following statement throws run-time exception:
KeyNotFoundException
        //employee.Remove("City");

        //Check key before removing it
        if (employee.ContainsKey("City"))
        {
            employee.Remove("City");
        }
        else
        {
            Console.WriteLine("Hashtable doesnot contain City key");
        }

        //Removes all elements
        employee.Clear();
        Console.WriteLine($"After Clearing Hashtable Total Elements:
{employee.Count}");

        Console.ReadKey();
    }
}

```

Output:

```
Hashtable Total Elements: 5  
After Removing EID Total Elements: 4  
Hashtable doesnot contain City key  
After Clearing Hashtable Total Elements: 0
```

image_173.png

Generic Collections

Generic Collections are introduced as part of C# 2.0. You can consider this **Generic Collection as an extension to the Non-Generic Collection Classes** which we have already discussed in our previous articles such as **ArrayList**, **Hashtable**, **SortedList**, **Stack**, and **Queue**.

Problems with Non-Generic Collections in C#

The **Non-Generic Collection Classes** such as **ArrayList**, **Hashtable**, **SortedList**, **Stack**, and **Queue** are worked on the **object data type**. That means the elements added to the collection are of an **object type**. As these **Non-Generic CollectionClasses** worked on **object data type**, we can store any type of value that may lead to runtime exceptions due to **type mismatch**. But with **Generic Collections**, now we are able to store a specific type of data (whether a primitive type or a reference type) which provides **type safety** by **eliminating the type mismatch at run time**. For a better understanding, please have a look at the below example.

```
using System;
using System.Collections;
namespace CollectionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList Numbers = new ArrayList(3);
            Numbers.Add(100);
            Numbers.Add(200);
            Numbers.Add(300);

            //It is also possible to store string values
            Numbers.Add("Hi");

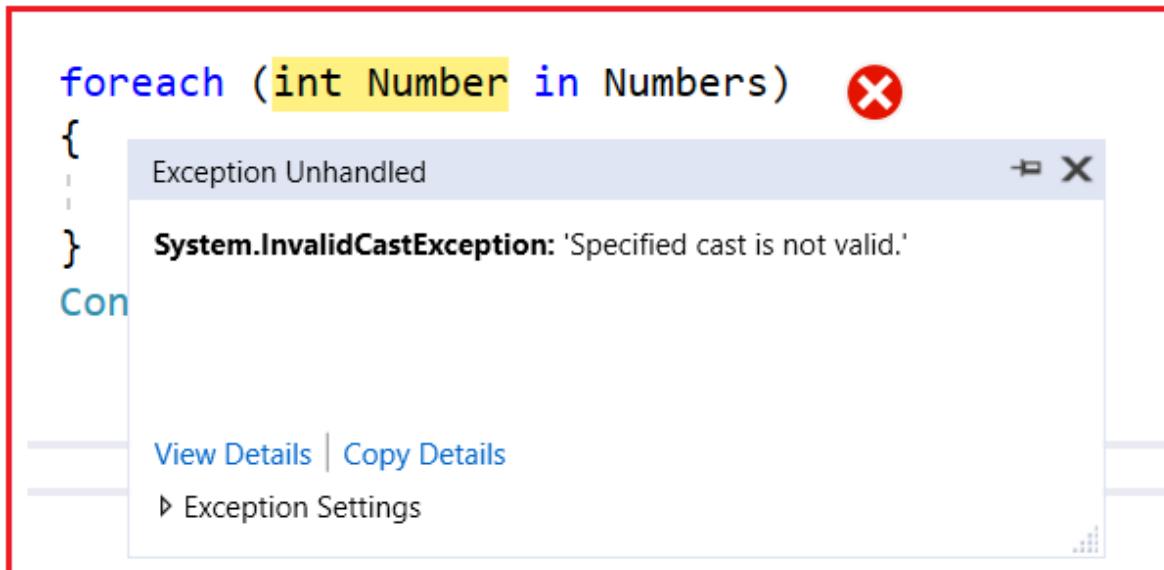
            foreach (int Number in Numbers)
            {
                Console.Write(Number + "   ");
            }
        }
    }
}
```

```

        }
        Console.ReadKey();
    }
}

```

Output:



image_174.png

The Solution to Non-Generic Collection Problems

The above two problems of Non-Generic Collections are overcome by using the Generic Collections in C#. The .NET Framework has re-implemented all the existing **Non-Generic Collection** classes such as **ArrayList**, **Hashtable**, **SortedList**, **Stack**, and **Queue**., etc. in **Generic Collections** such as **ArrayList<T>**, **Dictionary<TKey, TValue>**, **SortedList<TKey, TValue>**, **Stack<T>**, and **Queue<T>**.

- ⚠** Here **T** is nothing but the **type of values that we want to store in the collection**. So, the most important point that you need to remember is while creating the objects of Generic Collection Classes, you need to explicitly provide the type of values that you are going to store in the collection.

A Generic Collection is Strongly Type-Safe. Which type of data do you want to store in generic type, this information you have to provide at compile time. It means you can only put one type of data into it. This eliminates type mismatches at runtime.

The Generic Collection Classes are implemented under the System.Collections.Generic namespace. The classes which are present in this namespace are as follows.

1. **Stack<T>**: It represents a variable size last-in-first-out (LIFO) collection of instances of the same specified type.
2. **Queue<T>**: It represents a first-in, first-out collection of objects.
3. **HashSet<T>**: It represents a set of values. It eliminates duplicate elements.
4. **SortedList< TKey, TValue >**: It represents a collection of key/value pairs that are sorted by key based on the associated System.Collections.Generic.IComparer implementation. It automatically adds the elements in ascending order of key by default.
5. **List<T>**: It represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists. It grows automatically as you add elements to it.
6. **Dictionary< TKey, TValue >**: It represents a collection of keys and values.
7. **SortedSet<T>**: It represents a collection of objects that are maintained in sorted order.
8. **SortedDictionary< TKey, TValue >**: It represents a collection of key/value pairs that are sorted on the key.
9. **LinkedList<T>**: It represents a doubly linked list.



Here the <T> refers to the type of values we want to store under them

Examples:

```
List<int> listOfInteger = new List<int>(); // stores integer values only
List<float> listOfFloat = new List<float>(); // stores float values only
List<string> listOfString = new List<string>(); // stores string values only
```

image_175.png

List<T>

The `List<T>` class (from `System.Collections.Generic`) is a strongly typed, dynamic collection that stores elements in a sequential manner. It allows duplicates, maintains insertion order, and can grow or shrink dynamically at runtime.

It is the generic version of `ArrayList`, ensuring type safety and preventing type mismatches.

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

namespace GenericListDemo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Creating a generic list of integers
            List<int> numbers = new List<int>();

            // Adding elements using Add()
            numbers.Add(11);
            numbers.Add(22);
            numbers.Add(55);
            numbers.Add(65);
            numbers.Add(10);

            Console.WriteLine("List after Add operations:");
            DisplayList(numbers);

            // Adding multiple elements at once using AddRange()
            numbers.AddRange(new List<int> { 99, 77, 88 });
            Console.WriteLine("\nList after AddRange():");
            DisplayList(numbers);

            // Inserting an element at a specific position using
        }
    }
}
```

```
Insert()
{
    numbers.Insert(2, 100);
    Console.WriteLine("\nList after Insert(2, 100):");
    DisplayList(numbers);

    // Removing an element by value using Remove()
    numbers.Remove(55);
    Console.WriteLine("\nList after Remove(55):");
    DisplayList(numbers);

    // Removing an element by index using RemoveAt()
    numbers.RemoveAt(0);
    Console.WriteLine("\nList after RemoveAt(0):");
    DisplayList(numbers);

    // Contains() - checks if an element exists
    Console.WriteLine($"\\nList contains 65?
{numbers.Contains(65)}");

    // IndexOf() - gets the index of an element
    Console.WriteLine($"Index of 65: {numbers.IndexOf(65)}");

    // Count property - total number of elements
    Console.WriteLine($"\\nTotal number of elements:
{numbers.Count}");

    // Sort() - sorts elements in ascending order
    numbers.Sort();
    Console.WriteLine("\nList after Sort():");
    DisplayList(numbers);

    // Reverse() - reverses the order of elements
    numbers.Reverse();
    Console.WriteLine("\nList after Reverse():");
    DisplayList(numbers);

    // Clear() - removes all elements
    numbers.Clear();
}
```

```
        Console.WriteLine($"\\nList cleared. Count =\n{numbers.Count}");\n\n        Console.ReadKey();\n    }\n\n    // Helper method to display list elements\n    public static void DisplayList(List<int> list)\n    {\n        foreach (var item in list)\n        {\n            Console.Write(item + " ");\n        }\n        Console.WriteLine();\n    }\n}
```

Method / Property	Description
<code>Add(T item)</code>	Adds an element to the end of the list.
<code>AddRange(IEnumerable<T>)</code>	Adds multiple elements at once.
<code>Insert(int index, T item)</code>	Inserts an element at a specified position.
<code>Remove(T item)</code>	Removes the first matching element.
<code>RemoveAt(int index)</code>	Removes an element at a specific index.
<code>Contains(T item)</code>	Checks if an element exists.
<code>IndexOf(T item)</code>	Returns the index of the first matching element.
<code>Sort()</code>	Sorts the list in ascending order.
<code>Reverse()</code>	Reverses the list order.
<code>Count</code>	Returns the number of elements.
<code>Clear()</code>	Removes all elements.
<code>foreach</code>	Iterates over the list elements.

HashSet<T>

The `HashSet<T>` class in C# (from the `System.Collections.Generic` namespace) represents a collection of unique elements — meaning duplicates are not allowed. Unlike `SortedSet<T>`, it does not maintain order, but provides fast lookups and set operations like union, intersection, and difference.

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

namespace GenericHashSetDemo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Creating a HashSet of integers
            HashSet<int> numbers = new HashSet<int>();

            // Adding elements using Add()
            numbers.Add(11);
            numbers.Add(22);
            numbers.Add(55);
            numbers.Add(65);

            // Trying to add duplicate element (ignored automatically)
            numbers.Add(55);

            Console.WriteLine("Elements in HashSet:");
            foreach (var num in numbers)
            {
                Console.Write(num + " ");
            }

            // Count property
            Console.WriteLine($"\\n\\nTotal Elements: {numbers.Count}");
        }
    }
}
```

```

// Contains() - checks if a specific element exists
Console.WriteLine($"Contains 22? {numbers.Contains(22)}");

// Remove() - removes an element
numbers.Remove(11);
Console.WriteLine("\nAfter Removing 11:");
foreach (var num in numbers)
{
    Console.Write(num + " ");
}

// Working with another HashSet
HashSet<int> otherNumbers = new HashSet<int>() { 33, 44, 55,
66 };

// UnionWith() - adds all unique elements from another set
numbers.UnionWith(otherNumbers);
Console.WriteLine("\n\nAfter UnionWith(otherNumbers):");
foreach (var num in numbers)
{
    Console.Write(num + " ");
}

// IntersectWith() - keeps only common elements
numbers.IntersectWith(otherNumbers);
Console.WriteLine("\n\nAfter IntersectWith(otherNumbers):");
foreach (var num in numbers)
{
    Console.Write(num + " ");
}

// ExceptWith() - removes elements found in another set
numbers.ExceptWith(new HashSet<int>() { 44 });
Console.WriteLine("\n\nAfter ExceptWith({44}):");
foreach (var num in numbers)
{
    Console.Write(num + " ");
}

```

```

    }

    // Relationship checks
    HashSet<int> subset = new HashSet<int>() { 55 };
    Console.WriteLine($"\\n\\nIs Subset of otherNumbers?
{subset.IsSubsetOf(otherNumbers)}");
    Console.WriteLine($"Is Superset of otherNumbers?
{numbers.IsSupersetOf(otherNumbers)}");

    // Clear() - removes all elements
    numbers.Clear();
    Console.WriteLine($"\\nAfter Clear(): Count =
{numbers.Count}");

    Console.ReadKey();
}
}
}

```

⚠ If you notice, we have added 55 elements two times. Now, run the application and you will see that it removes the duplicate element and shows 55 only once as shown in the below image.

output:

List of Elements: 11 22 55 65

Method / Property	Description
<code>Add(T item)</code>	Adds an element to the set (duplicates ignored).
<code>Remove(T item)</code>	Removes a specific element from the set.
<code>Contains(T item)</code>	Checks whether an element exists.
<code>Count</code>	Returns the number of elements in the set.
<code>UnionWith(IEnumerable<T>)</code>	Adds all unique elements from another set.
<code>IntersectWith(IEnumerable<T>)</code>	Keeps only common elements between sets.
<code>ExceptWith(IEnumerable<T>)</code>	Removes elements found in another set.
<code>IsSubsetOf(IEnumerable<T>)</code>	Checks if the current set is a subset of another.
<code>IsSupersetOf(IEnumerable<T>)</code>	Checks if the current set contains another set.
<code>Clear()</code>	Removes all elements from the set.
<code>foreach</code>	Iterates through elements (unordered).

SortedSet<T>

The `SortedSet<T>` class in C# (in the `System.Collections.Generic` namespace) represents a collection of unique elements that are automatically sorted in ascending order by default.

- Duplicate elements are ignored.
- The sorting order can be customized using an `IComparer<T>` if needed.
- It's ideal when you want fast lookups, no duplicates, and sorted order.

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

namespace GenericSortedSetDemo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Creating a SortedSet of integers
            SortedSet<int> numbers = new SortedSet<int>();

            // Adding elements using Add() method
            numbers.Add(55);
            numbers.Add(22);
            numbers.Add(88);
            numbers.Add(11);
            numbers.Add(77);
            numbers.Add(66);

            // Trying to add a duplicate element (ignored automatically)
            numbers.Add(55);

            Console.WriteLine("Elements in SortedSet (Ascending
Order):");
        }
    }
}
```

```

foreach (var num in numbers)
{
    Console.Write(num + " ");
}

// Count property
Console.WriteLine($"\\n\\nTotal Elements: {numbers.Count}");

// Contains() - checks if an element exists
Console.WriteLine($"Contains 77? {numbers.Contains(77)}");

// Remove() - deletes an element
numbers.Remove(22);
Console.WriteLine("\\nAfter Removing 22:");
foreach (var num in numbers)
{
    Console.Write(num + " ");
}

// Min and Max properties
Console.WriteLine($"\\n\\nMinimum Element: {numbers.Min}");
Console.WriteLine($"Maximum Element: {numbers.Max}");

// Working with another SortedSet
SortedSet<int> otherNumbers = new SortedSet<int>() { 33, 44,
55, 66, 77 };

// UnionWith() - combines unique elements from both sets
numbers.UnionWith(otherNumbers);
Console.WriteLine("\\nAfter UnionWith(otherNumbers):");
foreach (var num in numbers)
{
    Console.Write(num + " ");
}

// IntersectWith() - keeps only common elements
numbers.IntersectWith(otherNumbers);
Console.WriteLine("\\n\\nAfter IntersectWith(otherNumbers):");

```

```

foreach (var num in numbers)
{
    Console.Write(num + " ");
}

// ExceptWith() - removes elements found in another set
numbers.ExceptWith(new SortedSet<int>() { 55, 77 });
Console.WriteLine("\n\nAfter ExceptWith({55, 77}):");
foreach (var num in numbers)
{
    Console.Write(num + " ");
}

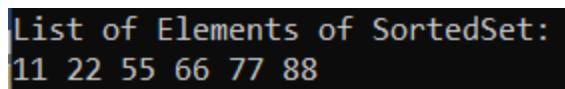
// Clear() - removes all elements
numbers.Clear();
Console.WriteLine($"\\n\\nAfter Clear(): Count = {numbers.Count}");

Console.ReadKey();
}
}
}

```

A As you notice in the above-sorted set, we have added 55 elements two times. Now, run the application and you will see that it removes the duplicate element and shows 55 only once as well as it will sort the elements in ascending order as shown in the below image.

output:



```
List of Elements of SortedSet:
11 22 55 66 77 88
```

image_177.png

Method / Property	Description
<code>Add(T item)</code>	Adds an element to the set (ignores duplicates).
<code>Remove(T item)</code>	Removes a specified element.
<code>Contains(T item)</code>	Checks whether an element exists.
<code>Count</code>	Returns the number of elements in the set.
<code>Min/Max</code>	Returns the smallest and largest element.
<code>UnionWith(IEnumerable<T>)</code>	Adds all elements from another set (unique only).
<code>IntersectWith(IEnumerable<T>)</code>	Retains only elements found in both sets.
<code>ExceptWith(IEnumerable<T>)</code>	Removes elements found in another set.
<code>Clear()</code>	Removes all elements from the set.
<code>foreach</code>	Iterates through the elements in ascending order.

Stack<T>

The `Stack<T>` class in C# (in the `System.Collections.Generic` namespace) stores elements in LIFO order — Last In, First Out. You add items using `Push()` and remove the most recent one with `Pop()`.

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

namespace GenericStackDemo
{
    class Program
    {
        static void Main()
        {
            // Creating a Generic Stack of string type
            Stack<string> countries = new Stack<string>();

            // Pushing elements into the Stack
            countries.Push("India");
            countries.Push("USA");
            countries.Push("Japan");
            countries.Push("UK");

            Console.WriteLine("Stack Elements After Push Operations:");
            foreach (var country in countries)
            {
                Console.WriteLine(country);
            }

            // Peek() - returns the top element without removing it
            Console.WriteLine($"\\nTop element (Peek): {countries.Peek()}");

            // Pop() - removes and returns the top element
            Console.WriteLine($"\\nPopped Element: {countries.Pop()}");
        }
    }
}
```

```

Console.WriteLine("\nStack Elements After Pop:");
foreach (var country in countries)
{
    Console.WriteLine(country);
}

// Contains() - checks if a specific element exists
Console.WriteLine($"\\nStack contains 'India'?
{countries.Contains("India")}");

// Count property - returns the number of elements
Console.WriteLine($"\\nNumber of Elements in Stack:
{countries.Count}");

// ToArray() - converts Stack to an array
string[] stackArray = countries.ToArray();
Console.WriteLine("\nStack Elements Converted to Array:");
foreach (var item in stackArray)
{
    Console.WriteLine(item);
}

// Clear() - removes all elements
countries.Clear();
Console.WriteLine($"\\nStack Cleared. Number of Elements Now:
{countries.Count}");

Console.ReadKey();
}
}
}

```

Method	Description
<code>Push(T item)</code>	Adds an item to the top of the stack.
<code>Pop()</code>	Removes and returns the top item.
<code>Peek()</code>	Returns the top item without removing it.
<code>Contains(T item)</code>	Checks if an item exists in the stack.
<code>Count</code>	Returns the number of elements in the stack.
<code>ToArray()</code>	Copies elements into a new array.
<code>Clear()</code>	Removes all items from the stack.
<code>foreach</code>	Iterates through the stack elements.

File Handling

What is a File?

A file is a collection of data stored on a disk with a specific name, extension, and directory path. When you open File using C# for reading and writing purposes it becomes Stream.

What is Stream?

A stream is a sequence of bytes travelling from a source to a destination over a communication path. There are two main streams: the Input Stream and the Output Stream. The input stream is used for reading data from the file (read operation) and the output stream is used for writing into the file (write operation).

- **Input Stream:** This Stream is used to read data from a file, which is known as a read operation.
- **Output Stream:** This Stream is used to write data into a file, which is known as a write operation.

Why do I need to Learn File Handling in C#?

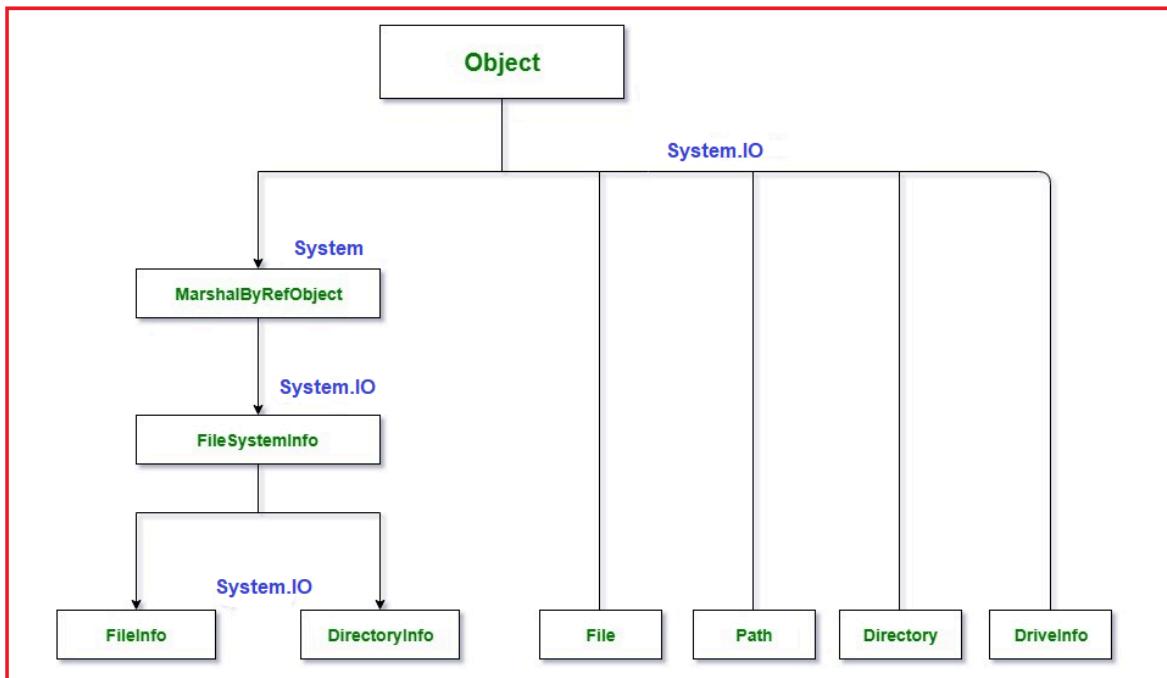
As a C# Programmer, we need to save information on a disk. We will not get a database everywhere to save the information and our project may require saving information in a TEXT file, DOC file, XLS file, PDF file, or any other file type. So, we must know the concept of saving data in a file i.e. How to Handle Files in C#.

Generally, we mostly perform three basic operations on a file: reading data from a file, writing data to a file and appending data to a file.

- Reading
- Writing
- Appending

System.IO Namespace in C#

In C#, The System.IO namespace contains the required classes used to handle the input and output streams and provide information about file and directory structure



image_178.png

⚠ The **FileInfo**, **DirectoryInfo**, and **DriveInfo** classes have instance methods. **File**, **Directory**, and **Path** classes have static methods. The following table describes commonly used classes in the System.IO namespace.

Class Name:	Description
FileStream:	It is used to read from and write to any location within a file
BinaryReader:	It is used to read primitive data types from a binary stream
BinaryWriter:	It is used to write primitive data types in binary format
StreamReader:	It is used to read characters from a byte Stream
StreamWriter:	It is used to write characters to a stream.
StringReader:	It is used to read from a string buffer
StringWriter:	It is used to write into a string buffer
DirectoryInfo:	It is used to perform operations on directories
FileInfo:	It is used to perform operations on files

image_179.png

FileStream Class

The FileStream class is used for reading and writing bytes directly to a file. It's the lowest-level file I/O operation and works with raw byte data.

```
using System;
using System.IO;

namespace FileHandlingDemo
{
    class FileStreamExample
    {
        static void Main()
        {
            string path = "example1.txt";

            // Writing bytes to file
            using (FileStream fs = new FileStream(path,
FileMode.Create))
            {
                byte[] data = System.Text.Encoding.UTF8.GetBytes("Hello,
FileStream!");
                fs.Write(data, 0, data.Length);
            }

            // Reading bytes from file
            using (FileStream fs = new FileStream(path, FileMode.Open,
 FileAccess.Read))
            {
                byte[] buffer = new byte[fs.Length];
                fs.Read(buffer, 0, buffer.Length);
                string content =
System.Text.Encoding.UTF8.GetString(buffer);
                Console.WriteLine($"FileStream Read: {content}");
            }
        }
    }
}
```

```
    }  
}
```

- FileMode.Create → Creates a new file or overwrites if it exists.
- fs.Write() → Writes byte array to the file.
- fs.Read() → Reads byte data from the file.
- Encoding.UTF8 converts text ↔ bytes.

StreamReader & StreamWriter

Used to **read/write text data** from and to files. These are higher-level classes built on top of streams.

```
using System;
using System.IO;

namespace FileHandlingDemo
{
    class StreamReaderWriterExample
    {
        static void Main()
        {
            string path = "example2.txt";

            // Writing text using StreamWriter
            using (StreamWriter writer = new StreamWriter(path))
            {
                writer.WriteLine("This is StreamWriter.");
                writer.WriteLine("It writes text easily!");
            }

            // Reading text using StreamReader
            using (StreamReader reader = new StreamReader(path))
            {
                string content = reader.ReadToEnd();
                Console.WriteLine("StreamReader Read:\n" + content);
            }
        }
    }
}
```

- `StreamWriter.WriteLine()` → Writes text line by line.*
- `StreamReader.ReadToEnd()` → Reads all the text from a file.*

- Automatically handles text encoding.

File Class

The File class provides static helper methods to quickly read, write, and manage files without manually opening streams.

```
using System;
using System.IO;

namespace FileHandlingDemo
{
    class FileClassExample
    {
        static void Main()
        {
            string path = "example3.txt";

            // Write all text
            File.WriteAllText(path, "This text is written using File
class!");

            // Read all text
            string content = File.ReadAllText(path);
            Console.WriteLine("File Content: " + content);

            // Append text
            File.AppendAllText(path, "\nAdding another line!");
            Console.WriteLine("\nAfter Append:");
            Console.WriteLine(File.ReadAllText(path));
        }
    }
}
```

- `File.WriteAllText()` → Creates or overwrites text files.
- `File.ReadAllText()` → Reads entire file content.

- `File.AppendAllText()` → Adds new text without overwriting

FileInfo Class

The FileInfo class gives detailed information about a file and provides instance-based methods (unlike File, which is static).

```
using System;
using System.IO;

namespace FileHandlingDemo
{
    class FileInfoExample
    {
        static void Main()
        {
            string path = "example4.txt";
            File.WriteAllText(path, "FileInfo class demo text!");

            FileInfo fileInfo = new FileInfo(path);

            Console.WriteLine($"File Name: {fileInfo.Name}");
            Console.WriteLine($"Full Path: {fileInfo.FullName}");
            Console.WriteLine($"Extension: {fileInfo.Extension}");
            Console.WriteLine($"Size (bytes): {fileInfo.Length}");
            Console.WriteLine($"Created On: {fileInfo.CreationTime}");

            // Copy file
            fileInfo.CopyTo("copy_example4.txt", true);

            // Delete file
            // fileInfo.Delete();
        }
    }
}
```

- `FileInfo.Length` → Size of the file in bytes.

- `FileInfo.CreationTime` → Date/time created.
- `CopyTo()` and `Delete()` → Manipulate the file physically.

DirectoryInfo Class

Used for managing directories (folders) — creation, deletion, listing files, and subdirectories.

```
using System;
using System.IO;

namespace FileHandlingDemo
{
    class DirectoryInfoExample
    {
        static void Main()
        {
            string folderPath = "TestFolder";

            // Create directory if not exists
            DirectoryInfo dir = new DirectoryInfo(folderPath);
            if (!dir.Exists)
            {
                dir.Create();
                Console.WriteLine("Directory created successfully.");
            }

            // Create a file in the directory
            File.WriteAllText(Path.Combine(folderPath, "file1.txt"),
"Hello from file1!");
            File.WriteAllText(Path.Combine(folderPath, "file2.txt"),
"Hello from file2!");

            // Get files
            FileInfo[] files = dir.GetFiles();
            Console.WriteLine("\nFiles in Directory:");
            foreach (FileInfo file in files)
            {
                Console.WriteLine($"{file.Name} ({file.Length} bytes)");
            }
        }
    }
}
```

```

// Get subdirectories (if any)
DirectoryInfo[] subDirs = dir.GetDirectories();
Console.WriteLine("\nSubdirectories:");
foreach (DirectoryInfo sub in subDirs)
{
    Console.WriteLine(sub.Name);
}

// dir.Delete(true); // Deletes directory and all contents
}
}

```

- `dir.Create()` → Creates a new directory.
- `GetFiles()` → Returns all files in the directory.
- `GetDirectories()` → Lists subfolders.
- `Delete(true)` → Deletes directory and its contents.

Asynchronous Programming

What is Concurrency?

Concurrency means doing several things at the same time. For example, if we have to do a million tasks, then instead of doing them sequentially one by one, we can do them simultaneously, thus reducing the duration of the program execution.

This concept of having a **Set of Tasks** and dividing them into several parts and those several parts can be performed simultaneously is called **Parallelism**.

How we can Achieve Parallelism in Programming?

In Programming, we can achieve Parallelism by using threads. *A thread is a sequence of instructions that can be executed independently of other code.* Since they are independent within a process, so we can have several threads. And if our processor allows, then we can run several threads simultaneously. When we are executing multiple threads simultaneously, then it is called multithreading. So, Parallelism uses multiple threads to perform multiple tasks simultaneously. Therefore, parallelism uses multithreading and multithreading is a form of concurrency.



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

Introduction to Asynchronous Programming

Asynchronous Programming allows us to handle the threads of our processes in a more efficient way. The idea is to avoid blocking a thread while waiting for a response, either from an external system such as a Web service or from the computer's file management system.

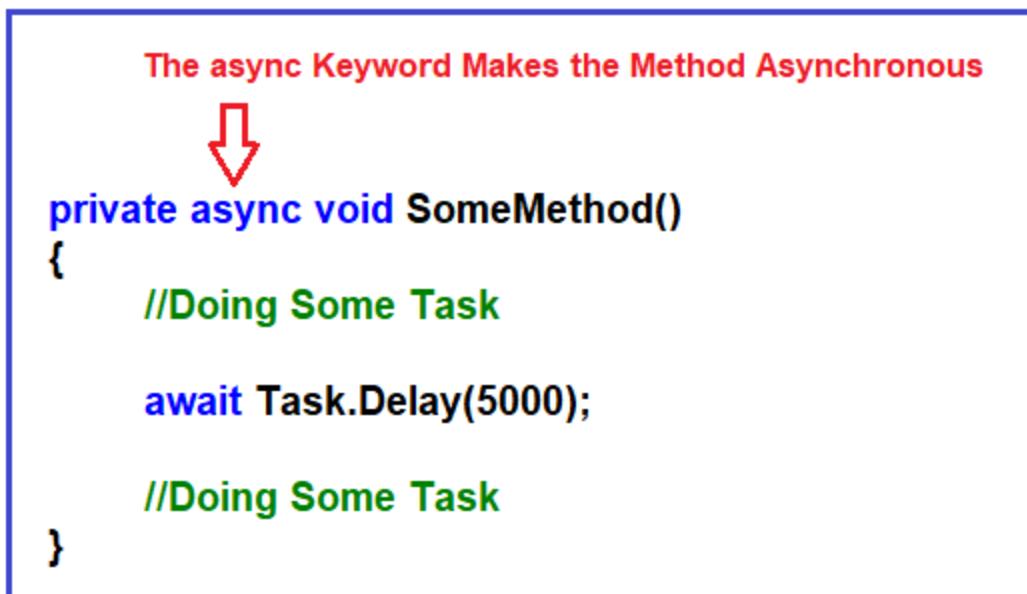
For example, if we have a web application, it will be able to serve more **HTTP requests** at the same time by using Asynchronous Programming. This is because each HTTP request is handled by a thread, and if we avoid blocking threads, then there will be more threads available to process HTTP requests.

To work with asynchronous programming in C# we use **async** and **await** keywords. The idea is that we need to use the **async** keyword to mark a method as asynchronous and with **await**, we can wait for an asynchronous operation in such a way that the original thread is not blocked.

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

Async and Await

In modern C# code, in order to use asynchronous programming, we need to use **async** and **await** keywords. The idea is that if we have a method in which we want to use asynchronous programming, then we need to mark the method with the **async** keyword as shown in the below image.



image_180.png

For those asynchronous operations for which we do not want to block the execution thread i.e. the current thread, we can use the **await** operator as shown in the below image.

```

private async void SomeMethod()
{
    //Doing Some Task

    await Task.Delay(5000);
    //Doing Some Task
}

```

The await keyword allows us to suspend the current thread, avoiding blocking it

image_181.png

Example to Understand Async and Await

Please have a look at the below example. It's a very simple example. Inside the main method, first, we print that main method started, then we call the SomeMethod. Inside the SomeMethod, first, we print that SomeMethod started and then the thread execution is sleep for 10. After 10 seconds, it will wake up and execute the other statement inside the SomeMethod method. Then it will come back to the main method, where we called SomeMethod. And finally, it will execute the last print statement inside the main method.

```

using System;
using System.Threading;
namespace AsynchronousProgramming
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Main Method Started.....");

            SomeMethod();

            Console.WriteLine("Main Method End");
            Console.ReadKey();
        }
    }
}

```

```

}

public static void SomeMethod()
{
    Console.WriteLine("Some Method Started.....");

    Thread.Sleep(TimeSpan.FromSeconds(10));
    Console.WriteLine("\n");
    Console.WriteLine("Some Method End");
}
}

```

When you execute the above code, you will see that after printing SomeMethod Started....., the console window is frozen for 10 seconds. This is because here we are not using asynchronous programming. One thread i.e. the Main thread is responsible for executing the code And when we call Thread.Sleep method the current thread is blocked for 10 seconds. This is a bad user experience



image_182.png

Now, let us see how we can overcome this problem by using asynchronous programming.

Task in C#

A task is basically a “promise” that the operation to be performed will not necessarily be completed immediately, but that it will be completed in the future.

What is the difference between Task and Task<T> in C#?

Task and Task<T> in C# for the return data type of an asynchronous method, the difference is that the Task is for methods that do not return a value while the Task<T> is for methods that do return a value of type T where T can be of any data type, such as a string, an integer, and a class, etc.



We know from basic C# that a method that does not return a value is marked with a void. This is something to avoid in asynchronous methods. So, don't use async void except for event handlers.

Example to Understand Task in C#

In our previous example, we have written the following SomeMethod.

```
public async static void SomeMethod()
{
    Console.WriteLine("Some Method Started.....");

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

    Console.WriteLine("\nSome Method End");
}
```

Now, what we will do is we will move the Task.Dealy to separate method and call that method inside the SomeMethod. So, let's create a method with the name Wait as follows. Here, we mark the method as async so it is an asynchronous method that will not block the currently executing thread. And when calling this method it will wait for 10 seconds. And more importantly, here we use the return type as Task as this method is not going to return anything.

```
private static async Task Wait()
{
    await Task.Delay(TimeSpan.FromSeconds(10));
    Console.WriteLine("\n10 Seconds wait Completed\n");
}
```

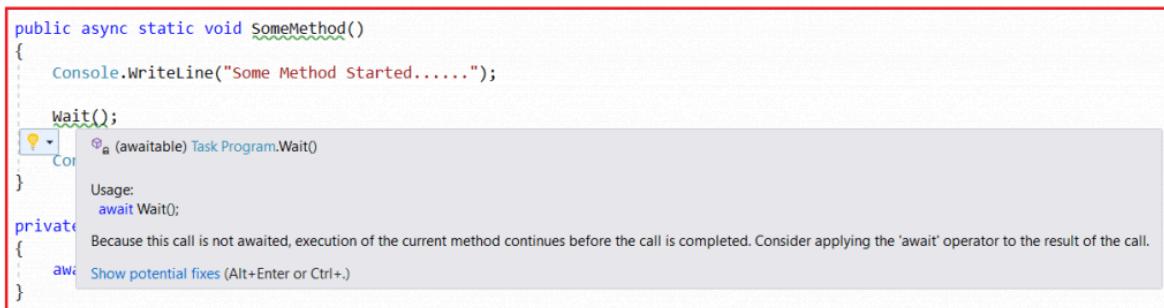
In asynchronous programming when your method does not return anything, then instead of using void you can use Task. Now, from the SomeMethod we need to call the Wait method. If we call the Wait method like the below then we will get a warning.

```
public async static void SomeMethod()
{
    Console.WriteLine("Some Method Started.....");

    Wait();

    Console.WriteLine("Some Method End");
}
```

Here, you can see green lines under the Wait method as shown in the below image.



image_183.png

Why is that? This is because the Wait method returns a Task and because it does return a Task, then it means that this will be a promise. So, this warning of the Wait method informed us that, if we don't use the await operator while calling the Wait method, the Wait method is not going to wait for this operation to finish, which means that once we call the Wait method, the next line of code inside the SomeMethod is going to be executed immediately. Let us see that practically. The following is the complete example code.

```
using System;
using System.Threading.Tasks;

namespace AsynchronousProgramming
{
    class Program
```

```

{
    public static void Main(string[] args)
    {
        Console.WriteLine("Main Method Started.....");

        SomeMethod();

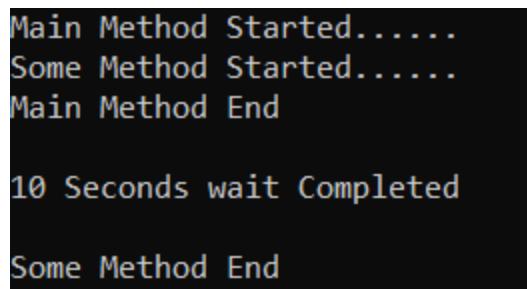
        Console.WriteLine("Main Method End");
        Console.ReadKey();
    }

    public async static void SomeMethod()
    {
        Console.WriteLine("Some Method Started.....");
        await Wait();
        Console.WriteLine("Some Method End");
    }

    private static async Task Wait()
    {
        await Task.Delay(TimeSpan.FromSeconds(10));
        Console.WriteLine("\n10 Seconds wait Completed\n");
    }
}

```

Output:



```

Main Method Started.....  

Some Method Started.....  

Main Method End  

10 Seconds wait Completed  

Some Method End

```

image_184.png

Now, you can observe in the above output that once it calls the `Wait` method, then the `SomeMethod` will wait for the `Wait` method to complete its execution. You can see that

before printing the last print statement of the SomeMethod, it prints the printing statement of the Wait method. Hence, it proves that when we use await operator then the current method execution waits until the called async method completes its execution. Once the async method, in our example Wait method, complete its example, then the calling method, in our example SomeMethod, will continue its execution i.e. it will execute the statement which is present after the async method call.

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

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

Example to Understand Task

Let us understand this with an example. What are we going to do is, we are going to communicate with a Web API that is deployed and is live, and we will try to retrieve the users that we receive from the Web API.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

namespace AsynchronousProgramming
{
    class Program
    {
        // Create a single HttpClient instance to reuse across calls
        // (singleton pattern)
        private static readonly HttpClient _httpClient;

        // Static constructor to initialize the HttpClient
        static Program()
        {
            _httpClient = new HttpClient
            {
```

```
        BaseAddress = new
Uri("https://jsonplaceholder.typicode.com/"),
    Timeout = TimeSpan.FromSeconds(30)
};

_httpClient.DefaultRequestHeaders.Accept.Clear();
_httpClient.DefaultRequestHeaders.Accept.Add(
    new
System.Net.Http.Headers.MediaTypeWithQualityHeaderValue("application/json"));
}

public static async Task Main(string[] args)
{
    Console.WriteLine("Main Method Started.....");

    try
    {
        // get all users
        Console.WriteLine("\nAll User info:");
        string users = await FetchUsersAsync();
        Console.WriteLine(users);

        // get a user with id
        Console.Write("Enter the id to search users: ");
        string id = Console.ReadLine();

        string userJson = await FetchUserAsync(id);
        Console.WriteLine("\nUser infor:");
        Console.WriteLine(userJson);

    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }

    Console.WriteLine("\nMain Method End");
}
```

```

        Console.ReadKey();
    }

    public static async Task<string> FetchUsersAsync()
    {
        HttpResponseMessage response = await
        _httpClient.GetAsync("users");
        response.EnsureSuccessStatusCode();

        string users = await response.Content.ReadAsStringAsync();
        return users;
    }

    public static async Task<string> FetchUserAsync(string id)
    {
        if (string.IsNullOrWhiteSpace(id))
            throw new ArgumentException("ID must be provided",
nameof(id));

        HttpResponseMessage response = await
        _httpClient.GetAsync($"users/{id}");
        response.EnsureSuccessStatusCode();

        string user = await response.Content.ReadAsStringAsync();
        return user;
    }
}

```

Key Concepts in the Code:

1. HttpClient as a Singleton

- **HttpClient** is used to make HTTP requests. The code creates a static instance of **HttpClient** (a singleton pattern) so that it can be reused across multiple requests. This is important because creating new **HttpClient** instances for each request can lead to resource exhaustion (e.g., socket exhaustion).

```

private static readonly HttpClient _httpClient;
static Program()
{
    _httpClient = new HttpClient
    {
        BaseAddress = new Uri("https://jsonplaceholder.typicode.com/"),
        Timeout = TimeSpan.FromSeconds(30)
    };
    _httpClient.DefaultRequestHeaders.Accept.Clear();
    _httpClient.DefaultRequestHeaders.Accept.Add(
        new
        System.Net.Http.Headers.MediaTypeWithQualityHeaderValue("application/json"));
}

```

- The static constructor ensures that the `HttpClient` instance is initialized once when the program starts. This is crucial for performance and resource management.

2. Asynchronous Methods: `FetchUsersAsync` and `FetchUserAsync`

The methods `FetchUsersAsync` and `FetchUserAsync` are asynchronous methods. They are declared with the `async` keyword and return a `Task<string>`, which represents an operation that will eventually produce a string result.

FetchUsersAsync: This method retrieves all users from the API asynchronously:

```

public static async Task<string> FetchUsersAsync()
{
    HttpResponseMessage response = await _httpClient.GetAsync("users");
    response.EnsureSuccessStatusCode();

    string users = await response.Content.ReadAsStringAsync();
    return users;
}

```

- `await _httpClient.GetAsync("users")`: This makes an HTTP GET request to retrieve the list of users. The `await` keyword means that this line will **not block** the thread while

waiting for the HTTP request to complete. Instead, the control returns to the calling method (`Main`) and the program continues executing other code.

- After the request is complete, the program continues and checks whether the request was successful using `response.EnsureSuccessStatusCode()`. If the request fails, it throws an exception.
- `await response.Content.ReadAsStringAsync()`: This asynchronously reads the response content as a string.

FetchUserAsync: This method retrieves information for a specific user by ID:

```
public static async Task<string> FetchUserAsync(string id)
{
    if (string.IsNullOrWhiteSpace(id))
        throw new ArgumentException("ID must be provided", nameof(id));

    HttpResponseMessage response = await
    _httpClient.GetAsync($"users/{id}");
    response.EnsureSuccessStatusCode();

    string user = await response.Content.ReadAsStringAsync();
    return user;
}
```

- It first checks if the provided `id` is valid. If the `id` is invalid (empty or null), it throws an exception.
- The same `await` pattern is used to make the HTTP GET request to fetch a user by ID. The program does not block while waiting for the response.
- After the request completes successfully, it reads the content and returns it.

3. Main Method (Entry Point)

The `Main` method is marked as `async Task`, which allows it to contain asynchronous code using the `await` keyword.

```

public static async Task Main(string[] args)
{
    Console.WriteLine("Main Method Started.....");

    try
    {
        Console.WriteLine("\nAll User info:");
        string users = await FetchUsersAsync(); // Fetch all users
        Console.WriteLine(users);

        Console.Write("Enter the id to search users: ");
        string id = Console.ReadLine();

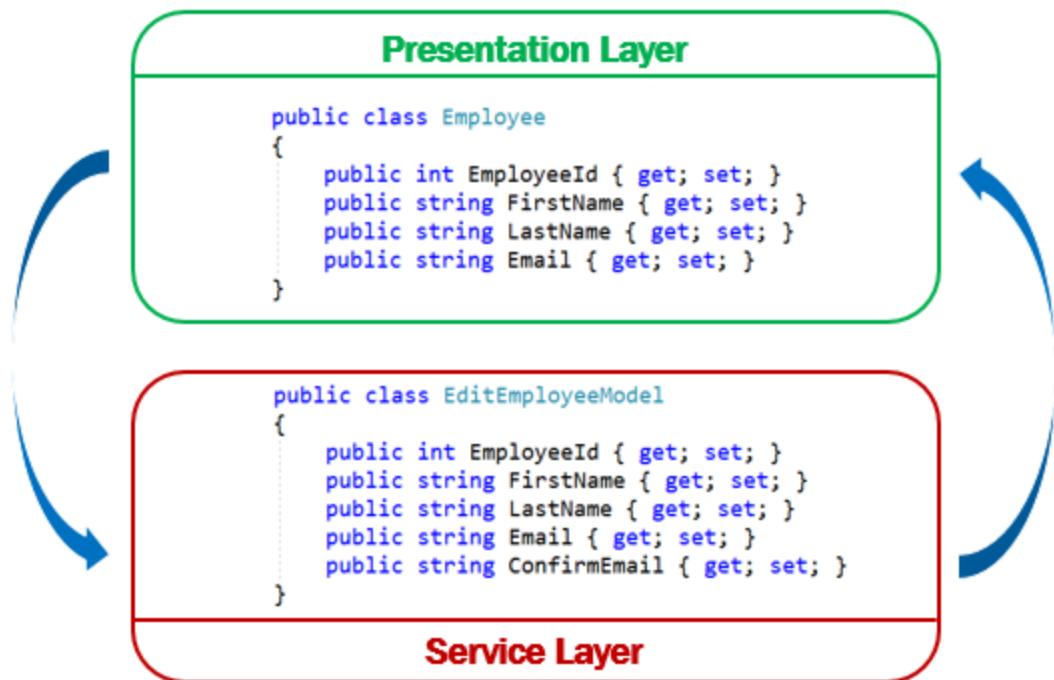
        string userJson = await FetchUserAsync(id); // Fetch a specific
        user by ID
        Console.WriteLine("\nUser info:");
        Console.WriteLine(userJson);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }

    Console.WriteLine("\nMain Method End");
    Console.ReadKey();
}

```

- **First Task:** The program first calls `FetchUsersAsync` using `await`, which fetches all users asynchronously. The program doesn't block while waiting for the response.
- **Second Task:** After displaying the users, it asks the user to input an ID and fetches that specific user using `FetchUserAsync` with the `await` keyword.
- The method does not block the main thread during these operations, meaning the application remains responsive, and the user can continue interacting with it (e.g., entering the user ID).

AutoMapper



image_185.png

Why do we need AutoMapper in C#?

Let's understand why we need Automapper in C# with an example. Let's say we have the following two classes: `Employee` and `EmployeeDTO`. First, create a class file named `Employee.cs`, and then copy and paste the following code. This is a very simple class having 4 properties.

```
namespace AutoMapperDemo
{
    public class Employee
    {
        public string Name { get; set; }
        public int Salary { get; set; }
        public string Address { get; set; }
        public string Department { get; set; }
    }
}
```

Next, create another class file with the name EmployeeDTO.cs and then copy and paste the following code into it. This class is identical to the Employee class, i.e., having 4 properties.

```
namespace AutoMapperDemo
{
    public class EmployeeDTO
    {
        public string Name { get; set; }
        public int Salary { get; set; }
        public string Address { get; set; }
        public string Department { get; set; }
    }
}
```

⚠ Now, what is our business requirement is to copy the data or transfer the data from the Employee object to the EmployeeDTO object

In the traditional approach (without using Automapper), first, we need to create and populate the Employee object, as shown in the image below.

```
//Create and Populate Employee Object
Employee emp = new Employee
{
    Name = "James",
    Salary = 20000,
    Address = "London",
    Department = "IT"
};
```

image_186.png

Once you have the employee object, you need to create the EmployeeDTO object and copy the data from the Employee object to the EmployeeDTO object, as shown in the

image below.

```
//Mapping Employee Object to EmployeeDTO Object
EmployeeDTO empDTO = new EmployeeDTO
{
    Name = emp.Name,
    Salary = emp.Salary,
    Address = emp.Address,
    Department = emp.Department
};
```

image_187.png

Mapping Object in Traditional Approach in C#

Let us understand how the Object is Mapped to another Object in C# using the Traditional Approach. For a better understanding, please have a look at the following example. First, we create an instance of the Employee object and populate the four properties with the required data. Then, we create an instance of the EmployeeDTO class and populate the EmployeeDTO properties with the values from the Employee object. Finally, we displayed the values of the EmployeeDTO object.

```
using System;
namespace AutoMapperDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create and Populate Employee Object
            Employee emp = new Employee
            {
                Name = "James",
                Salary = 20000,
                Address = "London",
                Department = "IT"
            };
        }
    }
}
```

```

//Mapping Employee Object to EmployeeDTO Object
EmployeeDTO empDTO = new EmployeeDTO
{
    Name = emp.Name,
    Salary = emp.Salary,
    Address = emp.Address,
    Department = emp.Department
};

Console.WriteLine("Name:" + empDTO.Name + ", Salary:" +
empDTO.Salary + ", Address:" + empDTO.Address + ", Department:" +
empDTO.Department);
Console.ReadLine();
}
}
}

```

If you run the application, you will get the output as expected. But tomorrow, what will you do if the data, i.e., the properties in the Employee and EmployeeDTO classes, are increased? Then, you must write the data moving code for each property from the source class to the destination class. That means the code mapping is repeated between the source and the destination. Again, if the same mapping is at different places, then you need to make the changes at different places, which is time-consuming and error-prone.

In Real-Time Projects, we often need to map the objects between the UI Layer or Presentation Layer and Business Logic layers. Mapping the objects between them is very hectic using the abovementioned traditional approach. So, is there any simplest solution by which we can map two objects? Yes, there is, and the solution is **AutoMapper**

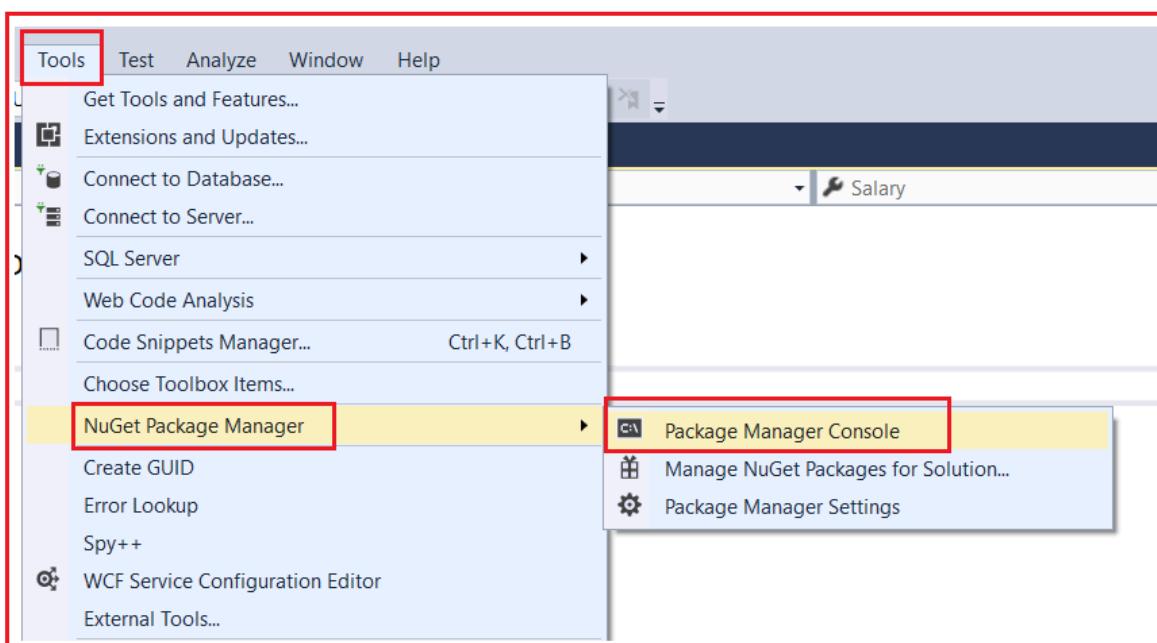
What is AutoMapper in C#

AutoMapper is a popular open-source library in C# that simplifies mapping data between different classes or objects. It helps eliminate repetitive and error-prone code when copying data from one object to another. AutoMapper is especially useful in scenarios like mapping database entities to DTOs (Data Transfer Objects) or ViewModel objects.

The AutoMapper in C# is a mapper between two objects. That is, AutoMapper is an Object-Object Mapper. It maps the properties of two different objects by transforming the input object of one type to the output object of another.

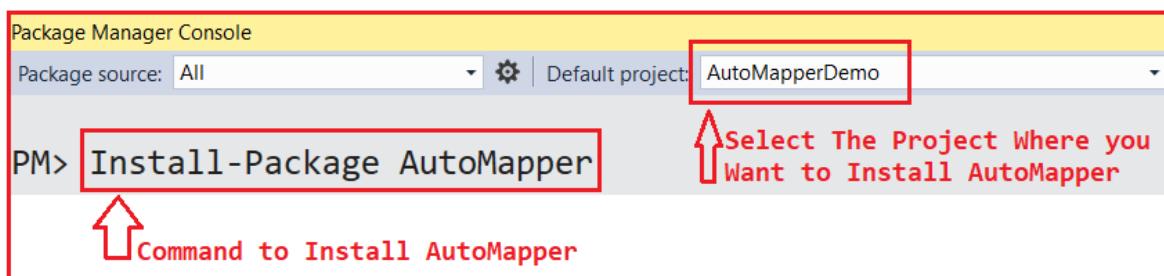
Installing AutoMapper Library in Your Project

AutoMapper is an open-source library present in GitHub (<https://github.com/AutoMapper>). To install this library, open the Package Manager Console window. To Open the Package Manager Console window, select Tools => NuGet Package Manager => Package Manager Console from the context menu, as shown in the image below.



image_188.png

Once you open the Package Manager Console window, type the command `Install-Package AutoMapper` and press the enter key to install the AutoMapper Library in your project, as shown in the image below.



image_189.png

AutoMapper Tutorial Project

A comprehensive C# project demonstrating AutoMapper for object-to-object mapping, including Entity models, DTOs (Data Transfer Objects), and ViewModels with practical examples.

Overview

This project demonstrates the proper use of AutoMapper in a .NET 8 console application. It showcases mapping between three different layers of data representation:

1. **Entity Layer** (`Employee.cs`) - Database model
2. **DTO Layer** (`EmployeeDto.cs`) - Data transfer object for API/service layer
3. **View Layer** (`EmployeeViewModel.cs`) - UI representation model

The project uses AutoMapper 15.1.0 with proper configuration and licensing.

What is AutoMapper?

AutoMapper is a powerful .NET library that automatically maps objects from one type to another. It reduces boilerplate code and makes your application more maintainable.

Key Benefits:

- **Convention-Based Mapping:** Automatically maps properties with matching names
- **Flattening:** Combines nested properties into a single flat object
- **Custom Transformations:** Supports complex mapping logic via `ForMember()`
- **Type Safety:** Uses generic types for compile-time safety
- **Lazy Loaded:** Only executes mappings when needed
- **Validation:** Can validate mapping configurations at startup

Why Use AutoMapper?

- Reduces repetitive mapping code
- Improves code maintainability
- Separates concerns between layers
- Makes unit testing easier
- Supports complex transformation scenarios

Project Structure

```

AutomapperTutorial/
├── Models/
│   └── Employee.cs           # Entity model (database
                                representation)
├── DTOs/
│   └── EmployeeDto.cs        # Data Transfer Object
└── View/
    └── EmployeeViewModel.cs   # ViewModel for UI
└── EmployeeProfile.cs       # AutoMapper configuration/profile
└── Program.cs               # Application entry point
└── Properties/
    └── launchSettings.json    # Launch settings
└── appsettings.json          # Application configuration
└── appsettings.Development.json # Development-specific settings
└── AutomapperTutorial.csproj # Project file
└── README.md                 # This file

```

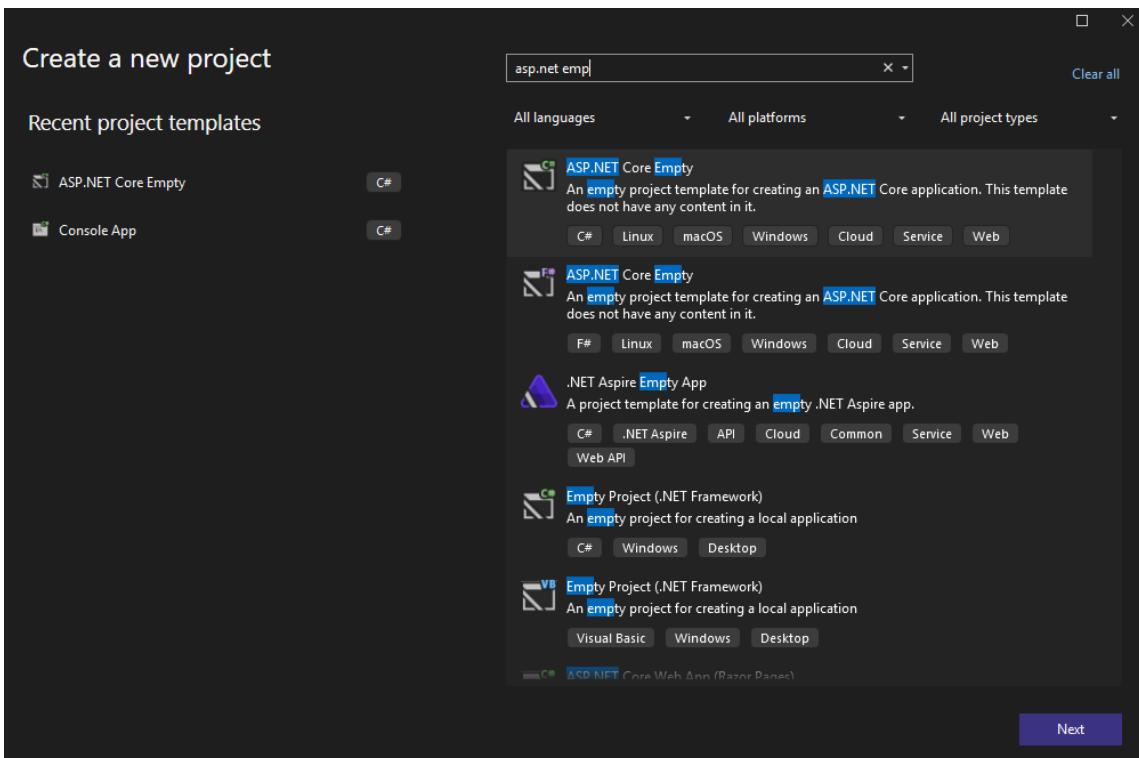
Getting Started

Prerequisites

- .NET 8 SDK or later
- Visual Studio, Visual Studio Code, or any .NET IDE

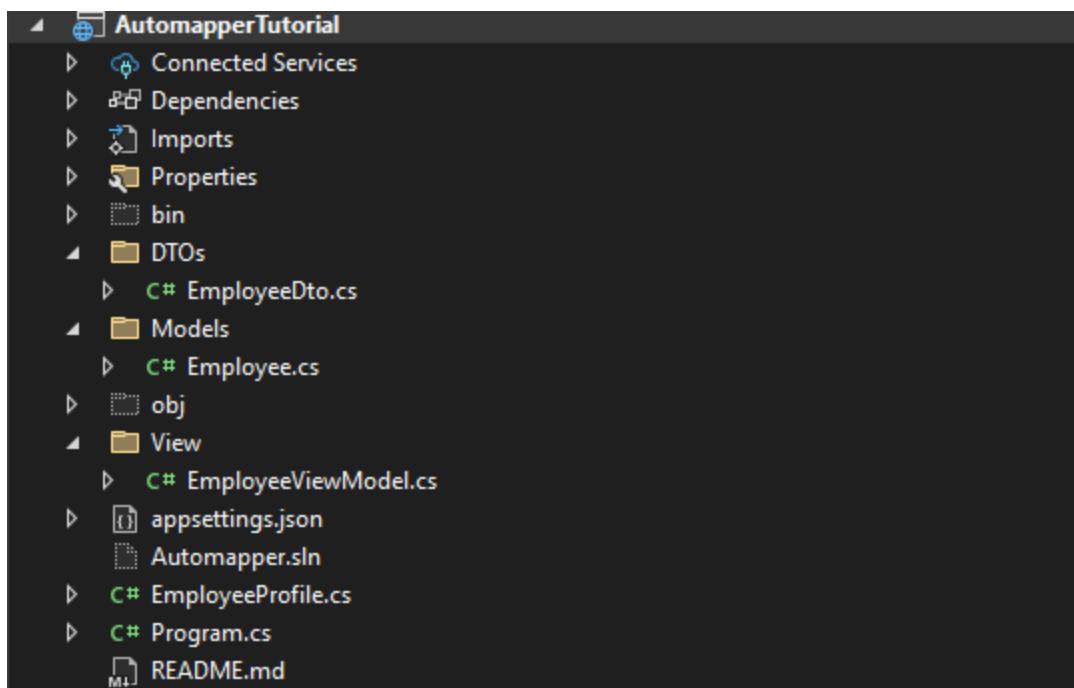
Project creation

1. Create a new project ASP.NET Core Empty



image_190.png

2. Create below folder structure



image_191.png

Running the Project using CLI

1. Build the project

```
dotnet build
```

2. Run the application

```
dotnet run --project AutomapperTutorial.csproj
```

Architecture & Mapping Patterns

Layered Architecture

This project follows a **3-layer mapping pattern**:

Layer 1: Entity Model

The **Entity** represents data stored in your database.

```
// Employee.cs - Database representation
public class Employee
{
    public int Id { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string? Department { get; set; }
}
```

Characteristics:

- Directly maps to database schema
- May contain sensitive information
- Not suitable for external APIs
- Can have navigation properties for related entities

Layer 2: Data Transfer Object (DTO)

The DTO transfers data between services, combining and transforming data as needed.

```
// EmployeeDto.cs - Service layer representation
public class EmployeeDto
{
    public int Id { get; set; }
    public string? FullName { get; set; }          // Flattened (FirstName
+ LastName)
    public string? Department { get; set; }
    public string? DateOfBirthFormatted { get; set; } // Formatted date
}
```

Characteristics:

- Combines related fields (flattening)
- Formats data appropriately
- Controls what data is exposed
- Enables API versioning

Layer 3: ViewModel

The ViewModel provides UI-specific representation, hiding unnecessary details.

```
// EmployeeViewModel.cs - UI representation
public class EmployeeViewModel
{
    public string? FullName { get; set; }
    public string? Department { get; set; }
    public string? DateOfBirthFormatted { get; set; }
}
```

Characteristics:

- Contains only UI-needed properties

- Already formatted for display
- Lightweight and focused
- No unnecessary identifiers

Mapping Flow

```

Employee (Entity)
↓
[Mapper using EmployeeProfile]
↓
EmployeeDto (DTO)
↓
[Mapper using EmployeeProfile]
↓
EmployeeViewModel (ViewModel)

```

Key Components

1. EmployeeProfile.cs - Mapping Configuration

Employee AutoMapper Profile - Defines all mapping configurations for employee-related classes.

This profile centralizes mapping logic between:

1. Employee (Entity) → EmployeeDto (DTO): Data transformation with flattening and formatting
2. EmployeeDto (DTO) → EmployeeViewModel (ViewModel): UI-specific data projection

AutoMapper Concepts Used:

- CreateMap: Defines source-to-destination type mapping
- ForMember: Customizes individual property mapping
- MapFrom: Specifies custom transformation logic

- Flattening: Combining multiple source properties into one destination property
- Value Transformation: Converting data types and formats

Profile Best Practices:

- Keep profiles organized by domain (Employee, Order, etc.)
- Register profiles in MapperConfiguration at application startup
- Use meaningful names for clarity
- Document complex mapping logic

```
using AutoMapper;
using AutomapperTutorial.DTOs;
using AutomapperTutorial.Models;
using AutomapperTutorial.View;

public class EmployeeProfile : Profile
{
    public EmployeeProfile()
    {
        // =====
        // MAPPING 1: Employee (Entity) → EmployeeDto (DTO)
        // =====
        // Purpose: Transform database entity into API/service layer DTO
        // Features:
        //   - Flattens FirstName + LastName into single FullName
        property
            //   - Formats DateTime to string in ISO 8601 format
            //   - Simplifies the data structure for external consumption
            // =====
            CreateMap<Employee, EmployeeDto>()
                // Flattening Example: Combine two source properties into
                one
                    // Source: Employee.FirstName (string) + Employee.LastName
                (string)
```

```

// Destination: EmployeeDto.FullName (string)
// Logic: Concatenate with space separator
.ForMember(
    dest => dest.FullName,
    opt => opt.MapFrom(src => $"{src.FirstName}{src.LastName}"))

// Value Transformation Example: Format DateTime to string
// Source: Employee.DateOfBirth (DateTime)
// Destination: EmployeeDto.DateOfBirthFormatted (string)
// Logic: Convert to ISO 8601 format (yyyy-MM-dd)
.ForMember(
    dest => dest.DateOfBirthFormatted,
    opt => opt.MapFrom(src =>
src.DateOfBirth.ToString("yyyy-MM-dd")));

// =====
// MAPPING 2: EmployeeDto (DTO) → EmployeeViewModel (ViewModel)
// =====
// Purpose: Project DTO data into UI-specific ViewModel
// Features:
//   - Excludes non-UI properties (like Id)
//   - Passes already-formatted data through
//   - Optimizes for presentation layer consumption
// =====
CreateMap<EmployeeDto, EmployeeViewModel>()
    // Direct property mapping - Uses convention-based mapping
    // These ForMember declarations are optional (properties
match by name)
    // but included here for documentation and explicit control
    .ForMember(
        dest => dest.FullName,
        opt => opt.MapFrom(src => src.FullName))

    .ForMember(
        dest => dest.Department,
        opt => opt.MapFrom(src => src.Department))

```

```

        .ForMember(
            dest => dest.DateOfBirthFormatted,
            opt => opt.MapFrom(src => src.DateOfBirthFormatted));
    }
}

```

Key Features:

- **Inheritance from Profile:** Organizes related mappings
- **CreateMap<TSource, TDestination>():** Defines mapping rules
- **ForMember():** Customizes individual property mapping
- **MapFrom():** Specifies source for destination property

2. Program.cs - Application Entry Point & Configuration

Initializes AutoMapper with license and runs the demonstration:

```

using AutoMapper;
using AutomapperTutorial.DTOs;
using AutomapperTutorial.Models;
using AutomapperTutorial.View;

class Program
{
    /// Static IMapper instance. Should be created once per AppDomain.
    /// In production, this would typically be injected via dependency
    /// injection.

    private static IMapper? _mapper;

    /// Application entry point. Demonstrates complete AutoMapper
    /// workflow.

    /// Steps:
}

```

```

/// 1. Initialize logging factory
/// 2. Configure AutoMapper with profiles and license
/// 3. Create mapper instance
/// 4. Create sample entity
/// 5. Map through layers (Entity → DTO → ViewModel)
/// 6. Display results

static void Main(string[] args)
{
    // =====
    // STEP 1: Initialize AutoMapper Configuration
    // =====
    // Create logger factory for diagnostics and error reporting
    // In production, this would be injected from DI container
    var loggerFactory = LoggerFactory.Create(builder =>
builder.AddConsole());

    // Create mapper configuration expression
    var configExpression = new MapperConfigurationExpression();

    // =====
    // STEP 2: Set AutoMapper License (Required for AutoMapper
15.0+)
    // =====
    // IMPORTANT: For production deployment, use one of these
approaches:
    //
    // Option A: Environment Variable (Recommended)
    //     var licenseKey =
Environment.GetEnvironmentVariable("AUTOMAPPER_LICENSE_KEY");
    //     configExpression.LicenseKey = licenseKey;
    //
    // Option B: Configuration File
    //     var licenseKey = configuration["AutoMapper:LicenseKey"];
    //     configExpression.LicenseKey = licenseKey;
    //
    // Option C: Azure Key Vault (Production)
    //     var licenseKey = await

```

```

keyVaultClient.GetSecretAsync("automapper-license");
    // configExpression.LicenseKey = licenseKey.Value;
    //
    // Current approach: Direct string (development/demo only)
    configExpression.LicenseKey =
"eyJhbGciOiJSUzI1NiIsImtpZCI6Ikx1Y2t5UGVubn1Tb2Z0d2FyZUxpY2Vuc2VLZXkvYmJ
iMTNhY2I1OTkwNGQ40WI0Y2IxYzg1Zja40GNjZjkilCJ0eXAiOiJKV1QifQ.eyJpc3MiOiJo
dHRwczovL2x1Y2t5cGVubnlzb2Z0d2FyZS5jb20iLCJhdWQiOiJMdWNreVB1bm55U29mdHdh
cmUiLCJleHAiOiIxNzk0MzU1MjAwIiwiaWF0IjoiMTc2Mjg2NzE4NCIsImFjY291bnRfaWQi
OiIwMTlhNzMxMjAwZGE3YmY3ODhhNjVkJM2NjNDdjNDBmYiIsImN1c3RvbWVyX2lkIjoiY3Rt
XzAxazlzaDUxMHZ0MjVjZTgxcDB0djIy0Hl3Iiwig3ViX2lkIjoiLSIsImVkaXRpb24i0iIw
IiwidHlwZSI6IjIifQ.P8lFhXbHdb9408QLFnwseIHsvcpP3xR7yEnJXM7z-Oi-
jEO_kwVM5d5uB0vXrP6uGa91nPz6Bx7uVdnh7xQp01XyFHXWoXRNAfwQn27CnxCeQz5hvD0
JiqwOd1X1wysFp2ZaotoEuF88R4kMC-
WP1IPKJr7Pt2gaJURY0ysav3xxG2rfevX_a9XNXz2uZx5RCivu0Q5IDPKjguk8xYMTNsUFzr
i_twv2NID5fWDkyA9lqlBg2bi33xxiZw9YnLXUbqxD7FP_7QtxrawvrR2J2GFg8CaIod3KRa
hnMzFiFS0co6N5vBB1KyPejUhhp42bKV8jcPx4bc0E2GNxY0CRQ"; // Get your
license at https://luckypennysoftware.com

    // =====
    // STEP 3: Register Mapping Profiles
    // =====
    // AddProfile registers the EmployeeProfile which contains
    // all mapping rules for Employee, EmployeeDto, and
EmployeeViewModel
    configExpression.AddProfile<EmployeeProfile>();

    // =====
    // STEP 4: Create MapperConfiguration and Mapper
    // =====
    // Note: AutoMapper 15.0+ requires ILoggerFactory as second
parameter
    // This is a breaking change from earlier versions
    var config = new MapperConfiguration(configExpression,
loggerFactory);
    _mapper = config.CreateMapper();

    Console.WriteLine("== AutoMapper Tutorial ==\n");

```

```

// =====
// STEP 5: Create Sample Entity (Database Model)
// =====
// Create an Employee entity representing database data
Employee employee = new Employee
{
    Id = 1,
    FirstName = "John",
    LastName = "Doe",
    DateOfBirth = new DateTime(1985, 5, 20),
    Department = "IT"
};

Console.WriteLine("1. Source Entity (Employee):");
Console.WriteLine($"    FirstName: {employee.FirstName}");
Console.WriteLine($"    LastName: {employee.LastName}");
Console.WriteLine($"    DateOfBirth: {employee.DateOfBirth}");
Console.WriteLine($"    Department: {employee.Department}\n");

// =====
// STEP 6a: Map Entity → DTO
// =====
// AutoMapper transforms:
// - FirstName + LastName → FullName (flattening)
// - DateOfBirth → DateOfBirthFormatted (value transformation)
EmployeeDto employeeDto = _mapper.Map<EmployeeDto>(employee);

Console.WriteLine("2. Mapped to DTO (EmployeeDto):");
Console.WriteLine($"    ID: {employeeDto.Id}");
Console.WriteLine($"    Full Name: {employeeDto.FullName}
(flattened from FirstName + LastName)");
Console.WriteLine($"    Department: {employeeDto.Department}");
Console.WriteLine($"    Date of Birth:
{employeeDto.DateOfBirthFormatted} (formatted as yyyy-MM-dd)\n");

// =====
// STEP 6b: Map DTO → ViewModel

```

```

// =====
// AutoMapper projects DTO data to UI-specific ViewModel
// Excludes unnecessary properties like Id
EmployeeViewModel employeeViewModel =
_mapper.Map<EmployeeViewModel>(employeeDto);

Console.WriteLine("3. Mapped to ViewModel
(EmployeeViewModel):");
Console.WriteLine($"    Full Name:
{employeeViewModel.FullName}");
Console.WriteLine($"    Department:
{employeeViewModel.Department}");
Console.WriteLine($"    Date of Birth:
{employeeViewModel.DateOfBirthFormatted}\n");

Console.WriteLine("== Mapping Complete ==");
Console.WriteLine("\nKey Takeaways:");
Console.WriteLine("• AutoMapper automates object
transformation");
Console.WriteLine("• Flattening combines multiple properties
into one");
Console.WriteLine("• Value transformation converts data types
and formats");
Console.WriteLine("• Profiles organize mapping logic by
domain");
Console.WriteLine("• License key required for AutoMapper 15.0+
production use");
}
}

```

Important Notes:

- MapperConfiguration requires ILoggerFactory parameter (AutoMapper 15.0+)
- Must be created once per AppDomain
- Should be initialized during application startup

- License key is required for production environments

3. EmployeeViewModel.cs - Represents the UI-specific data model.

```
namespace AutomapperTutorial.View
{
    /// Employee ViewModel - Represents the UI-specific data model.

    /// This class is designed specifically for display in the user interface.
    /// It contains only the properties needed for UI rendering and excludes unnecessary information like database IDs.
    /// Source: EmployeeDto (Data Transfer Object)

    /// Display Properties:
    /// - FullName: Employee's complete name (already formatted from DTO)
    /// - Department: Employee's department assignment
    /// - DateOfBirthFormatted: Employee's date of birth in human-readable format

    /// Note: The Id property is intentionally excluded as it's not needed for UI display.

    public class EmployeeViewModel
    {
        /// Employee's full name for display.
        /// Mapped directly from EmployeeDto.FullName (already formatted).
        /// Example: "John Doe"

        public string? FullName { get; set; }
    }
}
```

```

    /// Employee's department for display.
    /// Mapped directly from EmployeeDto.Department.
    /// Example: "IT", "HR", "Finance"

    public string? Department { get; set; }

    /// Employee's date of birth in human-readable format.
    /// Mapped directly from EmployeeDto.DateOfBirthFormatted.
    /// Format: "yyyy-MM-dd"
    /// Example: "1985-05-20"

    public string? DateOfBirthFormatted { get; set; }
}

}

```

4. EmployeeDto.cs - Represents the data transfer layer.

```

namespace AutomapperTutorial.DTOs
{

    /// Employee Data Transfer Object (DTO) - Represents the data
    transfer layer.

    /// This class is used for transferring employee data between
    services and APIs.

    /// It combines and transforms data from the Employee entity:
    /// - FirstName + LastName are combined into FullName (flattening)
    /// - DateOfBirth is formatted as a string in "yyyy-MM-dd" format
    /// - Excludes unnecessary properties from the entity

    /// Source: Employee (Entity Model)
    /// Target: EmployeeViewModel (View Model)

    /// Mapping Pattern:
    /// Employee.FirstName + Employee.LastName → EmployeeDto.FullName
    /// Employee.DateOfBirth → EmployeeDto.DateOfBirthFormatted (as

```

```
"yyyy-MM-dd")
    /// Employee.Department → EmployeeDto.Department (direct copy)

    public class EmployeeDto
    {

        /// Unique identifier for the employee. Copied from Employee.Id.

        public int Id { get; set; }

        /// Full name of the employee. Created by combining
        Employee.FirstName and Employee.LastName.

        /// This is an example of property flattening - combining
        multiple source properties into one.

        /// Example: "John Doe"

        public string? FullName { get; set; }

        /// Employee's department assignment.
        /// Directly mapped from Employee.Department.

        public string? Department { get; set; }

        /// Employee's date of birth in string format. Created from
        Employee.DateOfBirth.

        /// Format: "yyyy-MM-dd" (ISO 8601 format)
        /// Example: "1985-05-20"

        /// This is an example of value transformation - converting
        DateTime to formatted string.

        public string? DateOfBirthFormatted { get; set; }

    }
}
```

5. Employee.cs - Represents the database model for an employee.

```
namespace AutomapperTutorial.Models
{
    /// Employee Entity - Represents the database model for an employee.
    /// This class directly maps to the database table structure and contains all employee-related information as stored in the database.
    /// Mapping Targets:
    /// - Maps to: EmployeeDto (flattens FirstName + LastName into FullName)
    public class Employee
    {
        /// Unique identifier for the employee. Primary key in the database.
        public int Id { get; set; }

        /// Employee's first name. Nullable string.
        /// Combined with LastName to create FullName in EmployeeDto.
        public string? FirstName { get; set; }

        /// Employee's last name. Nullable string.
        /// Combined with FirstName to create FullName in EmployeeDto.
        public string? LastName { get; set; }

        /// Employee's date of birth. Used for age calculation and
    }
}
```

```

formatted display.

    /// Formatted as "yyyy-MM-dd" when mapped to EmployeeDto.

    public DateTime DateOfBirth { get; set; }

    /// Employee's department assignment. Examples: IT, HR, Finance,
    Sales.

    /// Passed through to EmployeeDto without modification.

    public string? Department { get; set; }
}

}

```

Examples

Basic Mapping

```

// Create source object
Employee employee = new Employee
{
    Id = 1,
    FirstName = "John",
    LastName = "Doe",
    DateOfBirth = new DateTime(1985, 5, 20),
    Department = "IT"
};

// Map to DTO
EmployeeDto employeeDto = _mapper.Map<EmployeeDto>(employee);

// Result: FullName = "John Doe", DateOfBirthFormatted = "1985-05-20"

```

Chained Mapping

```

// Map from Entity → DTO → ViewModel in sequence
EmployeeDto employeeDto = _mapper.Map<EmployeeDto>(employee);

```

```
EmployeeViewModel viewModel = _mapper.Map<EmployeeViewModel>(employeeDto);

// Or use ProjectTo for LINQ queries
var viewModels = dbContext.Employees
    .ProjectTo<EmployeeViewModel>(_mapper.ConfigurationProvider)
    .ToList();
```

Custom Transformations

The `ForMember()` method allows custom transformations:

```
CreateMap<Employee, EmployeeDto>()
    // Concatenate first and last names
    .ForMember(dest => dest.FullName,
        opt => opt.MapFrom(src => $"{src.FirstName} {src.LastName}"))

    // Format date
    .ForMember(dest => dest.DateOfBirthFormatted,
        opt => opt.MapFrom(src => src.DateOfBirth.ToString("yyyy-MM-
dd")))

    // Conditional mapping
    .ForMember(dest => dest.Department,
        opt => opt.Condition(src => src.Department != null));
```

Running the Application

Standard Execution

```
dotnet run --project AutomapperTutorial.csproj
```

With License Key (Environment Variable)

```
# Set environment variable first
$env:AUTOMAPPER_LICENSE_KEY="your-license-key"
```

```
# Then run  
dotnet run --project AutomapperTutorial.csproj
```

Build and Run

```
# Build  
dotnet build  
  
# Run specific executable  
.bin/Debug/net8.0/AutomapperTutorial.exe
```

Expected Output

```
Employee DTO:  
ID: 1  
Full Name: John Doe  
Department: IT  
Date of Birth: 1985-05-20
```

```
Employee ViewModel:  
Full Name: John Doe  
Department: IT  
Date of Birth: 1985-05-20
```

License Information

AutoMapper 15.1.0 License Requirement

Starting with AutoMapper 15.0, the library requires a valid license key for production use. This is provided by Lucky Penny Software.

License Types:

- **Development/Testing:** Free - no license key required (with warning)
- **Production:** Requires a valid license key

Obtaining a License:

1. Visit: <https://luckypennyssoftware.com>
2. Register an account
3. Subscribe to a plan (monthly or annual)
4. Copy your license key from the dashboard

Adding Your License Key:

In Program.cs, replace the license key:

```
configExpression.LicenseKey = "your-actual-license-key-here";
```

Best Practices for License Management:

Option 1: Environment Variable (Recommended)

```
var licenseKey =
Environment.GetEnvironmentVariable("AUTOMAPPER_LICENSE_KEY");
configExpression.LicenseKey = licenseKey;
```

Set the environment variable:

```
# Windows PowerShell
[Environment]::SetEnvironmentVariable("AUTOMAPPER_LICENSE_KEY", "your-
key", "User")

# Windows Command Prompt
setx AUTOMAPPER_LICENSE_KEY "your-key"

# Linux/macOS
export AUTOMAPPER_LICENSE_KEY="your-key"
```

Option 2: Configuration File

```
{
  "AutoMapper": {
    "LicenseKey": "your-license-key-here"
```

```
    }  
}
```

In Program.cs:

```
var licenseKey = builder.Configuration["AutoMapper:LicenseKey"];  
configExpression.LicenseKey = licenseKey;
```

Option 3: Azure Key Vault (Production)

```
var licenseKey = await keyVaultClient.GetSecretAsync("automapper-  
license-key");  
configExpression.LicenseKey = licenseKey.Value;
```

Current License Status:

Current License Key: Valid (expires January 15, 2027)

- Account ID: 019a7312-00da-7bf7-88a6-5d3cc47c40fb
- Customer ID: ctm_01k9sh510vt25ce81p0tv228yw
- Edition: 0
- Type: 2

License

This project uses AutoMapper, which requires a license key for production use.

AutoMapper License: <https://luckypennyssoftware.com>

Project Code: MIT (or your chosen license)

Lambda Expressions

A lambda expression is a **short, inline anonymous function** you write with the `=>` operator (pronounced “goes to”).

- Syntax for a single expression:

```
parameter => expression
```

- Syntax for multiple statements:

```
(parameters) => { statement1; statement2; ... }
```

- Example:

```
Func<int,int> square = x => x * x;  
Console.WriteLine(square(5)); // Output: 25
```



Why this matters: lambdas let you define short logic **where it's used**, rather than creating separate named methods.

Anatomy of a Lambda Expression

Part	Description
Input parameters	On the left of <code>=></code> . If only one parameter you can omit parentheses.
<code>=></code> operator	Separates parameters and body.
Body	Right side: either a single expression (expression lambda) or a block <code>{ ... }</code> of statements (statement lambda).

Key tip: Use expression lambdas when you have a simple return value; use statement lambdas when you need multiple lines (e.g., logging + calculation). Example statement lambda:

```
Action<string> greet = name => { var message = $"Hello {name}!";  
Console.WriteLine(message); };  
greet("World"); // Hello World!
```

How Lambdas Work with LINQ

LINQ (Language Integrated Query) enables querying collections (and other data sources) in a fluent, functional style. Lambda expressions are the backbone of LINQ's "method syntax".

Here are common operations:

```
### 1. Filtering (`Where`)  
```C#  
List<int> numbers = new List<int>{1,2,3,4,5};
var evens = numbers.Where(n => n % 2 == 0);
// evens: 2, 4
```

- `n => n % 2 == 0` is a predicate lambda (returns `bool`).
- `Where` takes a `Func<T, bool>` delegate under the hood.

## 2. Projection (Select)

```
List<Person> people = /* ... */;
var names = people.Select(p => p.Name);
// yields sequence of names
```

- `p => p.Name` is a mapping lambda (takes `Person`, returns `string`).
- `Select` takes a `Func<TSource, TResult>`.

### 3. Sorting, Grouping, etc.

```
var sorted = numbers.OrderBy(n => n);
var grouped = people.GroupBy(p => p.Department);
```

- Each uses a lambda to say *how* to order or group.
- Enables concise, expressive queries without loops.

## Why Use Lambdas + LINQ?

- **Conciseness:** Eliminates boilerplate.
- **Readability:** Logic stays close to data operations.
- **Expressiveness:** Easily chain operations (Where, Select, OrderBy, etc).
- **Functional style:** Encourages thinking in terms of data transformations rather than imperative loops.

## Quick “Skills” Example – Employee Scenario

Suppose you have a list of employees:

```
public class Employee
{
 public int Id { get; set; }
 public string Name { get; set; }
 public string Department { get; set; }
 public decimal Salary { get; set; }
}

List<Employee> employees = /* ... */;
```

*Filter employees in “Sales” with salary > 50,000, then select their names:*

```
var result = employees
 .Where(emp => emp.Department == "Sales" && emp.Salary > 50000m)
```

```
.Select(emp => emp.Name);
```

Here:

- `emp => emp.Department == "Sales" && emp.Salary > 50000m` → predicate lambda for `Where`.
- `emp => emp.Name` → projection lambda for `Select`.

*If you wanted a statement lambda to log each employee before projection:*

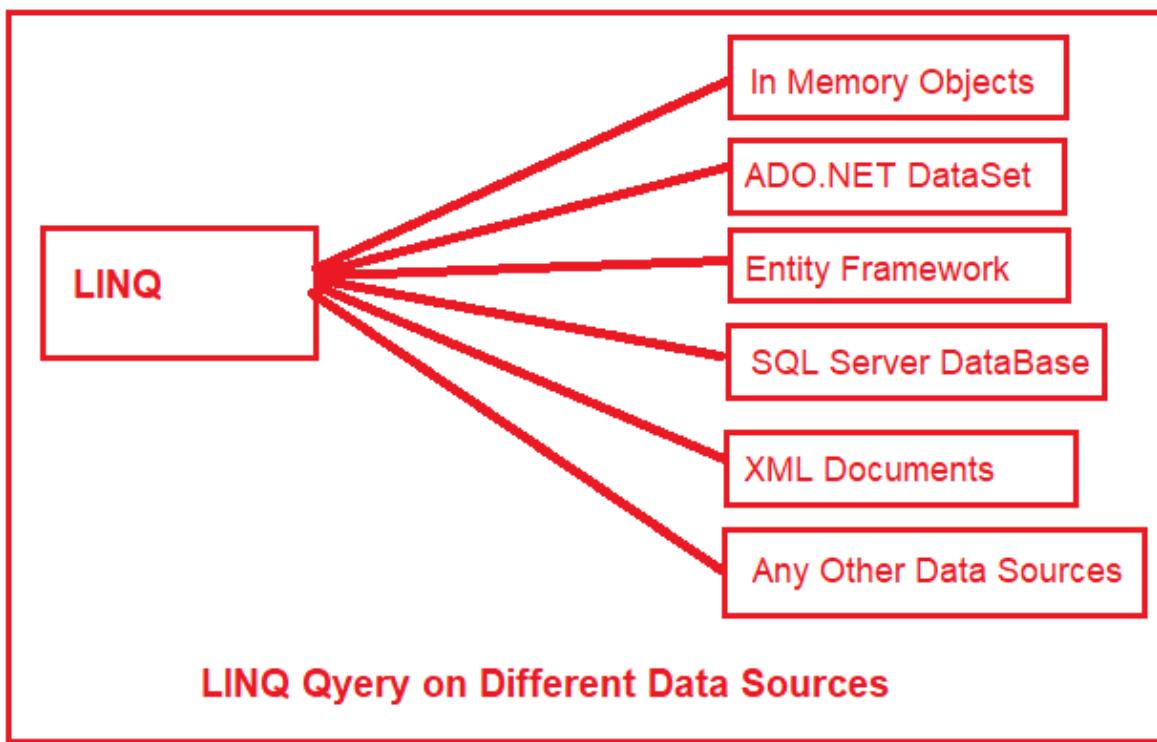
```
var result2 = employees
 .Where(emp =>
{
 Console.WriteLine($"Checking employee {emp.Name}");
 return emp.Salary > 50000m;
})
.Select(emp => emp.Name);
```

## Best Practices & Tips

- Keep lambdas **short and clear**. If it becomes complex, extract a named method.
- Use meaningful parameter names (not just `x`, `y`) when dealing with domain objects (e.g., `emp` rather than `e`).
- Be mindful of deferred execution: LINQ queries using lambdas evaluate only when iterated.
- Avoid side-effects in lambdas if the intent is purely querying/filtering data—maintaining functional style aids readability and maintainability.

# LINQ

Most queries in the introductory **Language Integrated Query** (LINQ) documentation are written by using the LINQ declarative query syntax. The C# compiler translates query syntax into method calls. These method calls implement the standard query operators, and have names such as `Where`, `Select`, `GroupBy`, `Join`, `Max`, and `Average`. You can call them directly by using method syntax instead of query syntax.



image\_192.png

Microsoft introduced LINQ (Language Integrated Query) with .NET Framework 3.5 and C# 3.0, available in the `System.Linq` namespace.

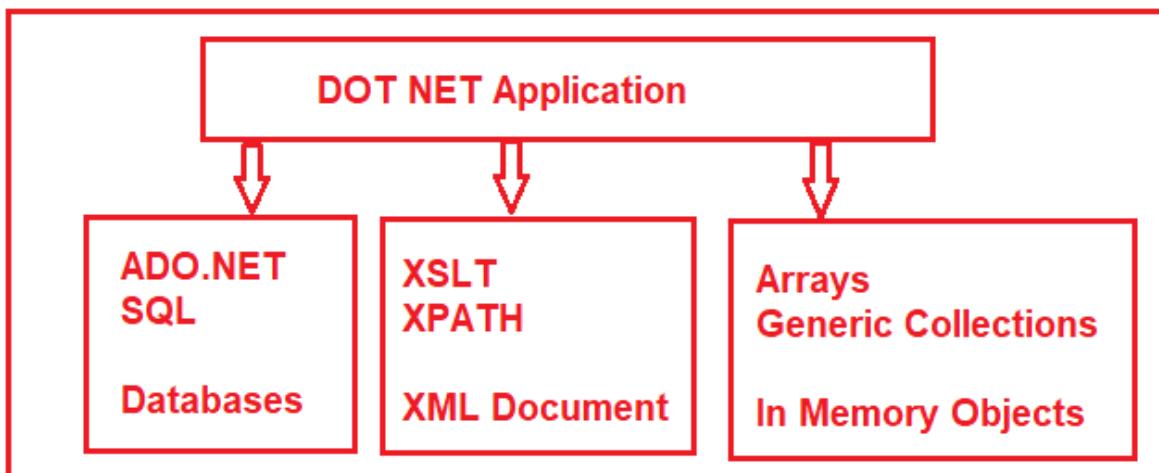
## Why Should We Learn LINQ?

Suppose we are developing a .NET Application that requires data from different sources. For example

1. The application needs data from the SQL Server Database. So, as a developer, to access the data from the SQL Server Database, ***we need to understand ADO.NET and***

*SQL Server-specific syntaxes. We need to learn SQL Syntax specific to Oracle Database if the database is Oracle.*

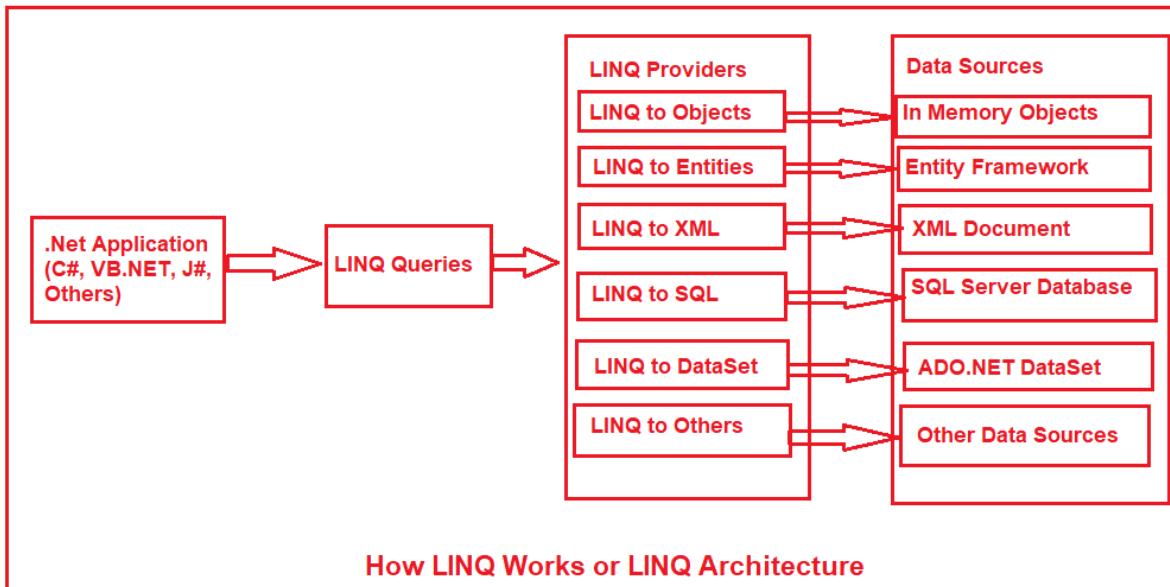
2. The application also needs data from an XML Document. So, as a developer, to work with XML documents, we need to understand XPath and XSLT queries.
3. The application also needs to manipulate the data (objects) in memory, such as `List<Products>`, `List<Orders>`, etc. So, as a developer, we should also understand how to work with in-memory objects.



image\_193.png

- ⚠ LINQ provides a Uniform Programming Model (i.e., Common Query Syntax), which allows us to work with different data sources such as databases, XML Documents, in-memory objects, etc., but using a standard or, you can say, unified coding style. As a result, we are not required to learn different syntaxes to query different data sources.

## How Does LINQ Work?



image\_194.png

As shown in the above diagram, you can write the LINQ queries using any DOT NET Supported Programming Language such as C#, VB.NET, J#, F#

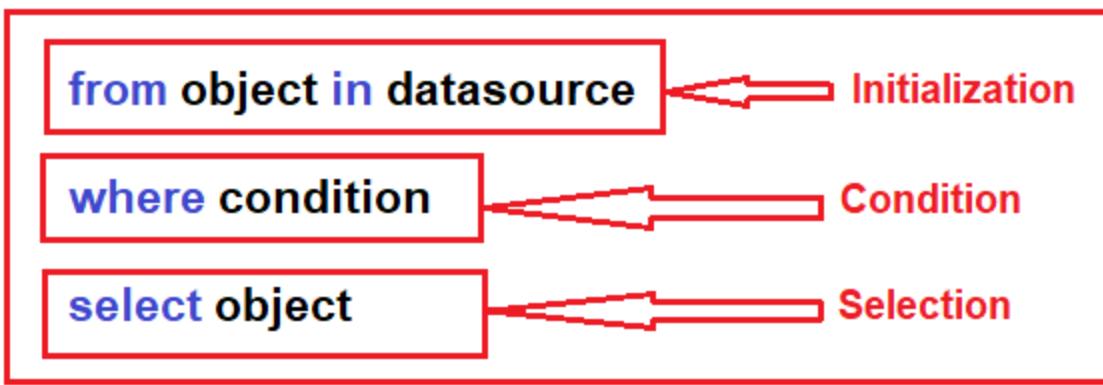
## Different Ways to Write LINQ Queries in C#

LINQ queries can be written in two ways:

- 1. Query Syntax:** It is similar to SQL and is often more readable for those familiar with SQL. It starts with a from clause followed by a range variable and includes standard query operations like where, select, group, join, etc.

Each query is a combination of three things. They are as follows:

- **Initialization** (to work with a particular data source)
- **Condition** (where, filter, sorting condition)
- **Selection** (single selection, group selection, or joining)



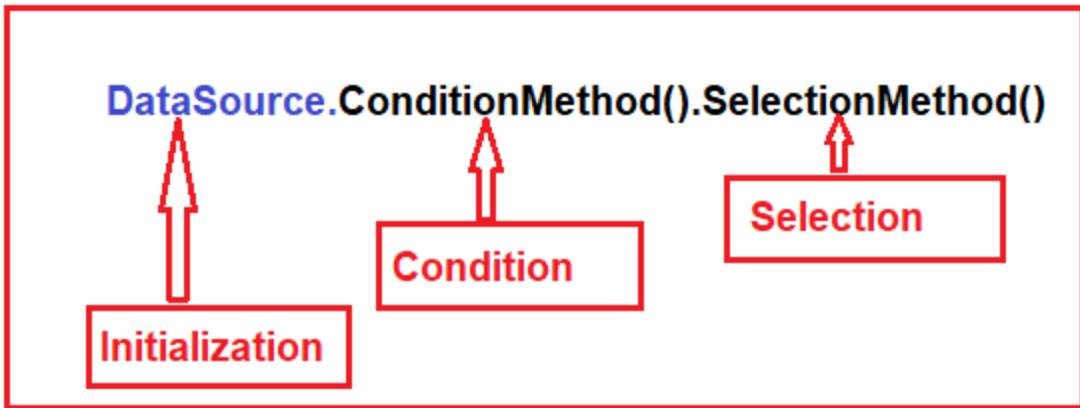
image\_195.png

#### Characteristics:

- Resembles SQL-like declarative style.
- It can be more readable, especially for those familiar with SQL.
- Not all operations can be expressed in Query Syntax; some require a switch to Method Syntax.

```
var query = from c in customers
 where c.City == "London"
 select c.Name;
```

**2. Method Syntax (Fluent/Lambda Syntax):** It uses extension methods and lambda expressions. It can be more concise and is preferred when writing complex queries because it can be easier to read and compose.



image\_196.png

```
var query = customers.Where(c => c.City == "London").Select(c =>
c.Name);
```

#### Characteristics:

- Utilizes lambda expressions.
- It can be more concise for complex queries.
- Offers slightly more methods and flexibility than Query Syntax.
- It can be easier to understand for those familiar with lambda expressions and functional programming.

### Example Using LINQ Query Syntax in C#:

The following Example code is self-explained, so please go through the comment lines. Here, we have created a collection of integers, i.e., going to be our data source

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace LINQDemo
{
 class Program
 {
 static void Main(string[] args)
```

```

{
 //Step1: Data Source
 List<int> integerList = new List<int>()
 {
 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
 };

 //Step2: Query
 //LINQ Query using Query Syntax to fetch all numbers which
 are > 5

 var QuerySyntax = from obj in integerList //Data Source
 where obj > 5 //Condition
 select obj; //Selection

 //Step3: Execution
 foreach (var item in QuerySyntax)
 {
 Console.WriteLine(item + " ");
 }

 Console.ReadKey();
}
}
}

```

Now run the application, and it will display the values 6 7 8 9 10 as expected in the console window. Let us understand what we did in the above code.

### 1. Data Source

```
List<int> integerList = new List<int>()
{
 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};
```

### 2.Query

```
var QuerySyntax = from obj in integerList
 where obj > 5
 select obj;
```

The diagram illustrates the three components of LINQ query syntax. The 'Initialization' component is 'from obj in integerList'. The 'Condition' component is 'where obj > 5'. The 'Selection' component is 'select obj;'. Arrows point from the labels to their respective parts in the code.

### 3. Execution

```
foreach(var item in QuerySyntax)
{
 Console.WriteLine(item + " ");
}
```

image\_197.png

## Example Using LINQ Method Syntax(Fluent/Lambda) in C#:

Let us rewrite the previous example using the LINQ Method Syntax. The following Example code is self-explained, so please go through the comment lines.

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace LINQDemo
{
 class Program
 {
 static void Main(string[] args)
 {
 //Step1: Data Source
```

```

 List<int> integerList = new List<int>()
 {
 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
 };

 //Step2: Query
 //LINQ Query using Query Syntax to fetch all numbers which
are > 5
 var QuerySyntax = integerList.Where(obj => obj >
5).ToList();

 //Step3: Execution
 foreach (var item in QuerySyntax)
 {
 Console.WriteLine(item + " ");
 }

 Console.ReadKey();
}
}

```

Now, run the application, and you will get the output as expected. Let us have a look at the following diagram to understand Method Syntax.

**1. Data Source**

```
List<int> integerList = new List<int>()
{
 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};
```

**2. Linq Query Using Method Syntax**

```
var MethodSyntax = integerList.Where(obj => obj > 5).ToList();
```

**3. Execution**

```
foreach(var item in MethodSyntax)
{
 Console.WriteLine(item + " ");
}
```

image\_198.png

# IEnumerable and IQueryable in C#

Let us understand these two interfaces with examples. Please look at the following program we wrote using the LINQ Query Syntax in our previous article.

```
class Program
{
 static void Main(string[] args)
 {
 List<int> integerList = new List<int>()
 {
 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
 };

 var QuerySyntax = from obj in integerList
 select obj;
 }

 foreach (var item in QuerySyntax)
 {
 Console.Write(item + " ");
 }

 Console.ReadKey();
}
```

image\_199.png

In the above example, we use the **var** keyword to create the variable and store the result of the **LINQ query**. So, let's check what the type of variable is. To check this, *just mouse over the pointer onto the **QuerySyntax** variable*, and you will see that the type is **IEnumerable**, which is a generic type. So, it is important to understand what is **IEnumerable**.

So, in the above example, instead of writing the **var** keyword, you can also write **IEnumerable<int>**, and it should work as expected, as shown in the below example.

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace LINQDemo
{
 class Program
 {
 static void Main(string[] args)
 {
 List<int> integerList = new List<int>()
 {
 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
 };

 IEnumerable<int> QuerySyntax = from obj in integerList
 where obj > 5
 select obj;

 foreach (var item in QuerySyntax)
 {
 Console.Write(item + " ");
 }

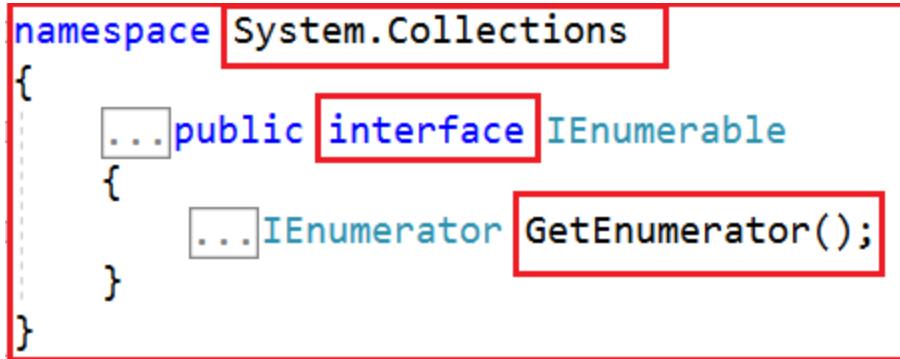
 Console.ReadKey();
 }
 }
}

```

With this kept in mind, let us understand what **IEnumerable** is.

## What is **IEnumerable** in C#?

**IEnumerable** in C# is an interface that defines one method, **GetEnumerator**, which returns an **IEnumerator** object. This interface is found in the **System.Collections** namespace. It is a key part of the .NET Framework and is used to iterate over a collection of objects. The following is the definition of the **IEnumerable** interface.



image\_200.png

## Example to understand IEnumerable with Complex Type using C#

Whenever we want to work with in-memory objects, we need to use the `IEnumerable` interface, and the reason for this will be discussed in our next article.

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

namespace LINQDemo
{
 class Program
 {
 static void Main(string[] args)
 {
 List<Student> studentList = new List<Student>()
 {
 new Student(){ID = 1, Name = "James", Gender = "Male"},
 new Student(){ID = 2, Name = "Sara", Gender = "Female"},
 new Student(){ID = 3, Name = "Steve", Gender = "Male"},
 new Student(){ID = 4, Name = "Pam", Gender = "Female"}
 };

 // Query Syntax: Fetch all students with Gender Male
 //IEnumerable<Student> QuerySyntax = from std in studentList
 // where std.Gender ==
 "Male"
 // select std;
```

```

 //// Iterate through the collection returned by query syntax
 //foreach (var student in QuerySyntax)
 //{
 // Console.WriteLine($"ID : {student.ID} Name :
{student.Name}");
 //}

 // Method (Fluent) Syntax Equivalent (commented
demonstration):
 IEnumerable<Student> MethodSyntax = studentList.Where(std =>
std.Gender == "Male");
 foreach (var student in MethodSyntax)
 {
 Console.WriteLine($"ID : {student.ID} Name :
{student.Name}");
 }

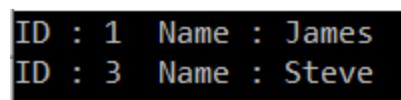
 Console.ReadKey();
}

}

public class Student
{
 public int ID { get; set; }
 public string Name { get; set; }
 public string Gender { get; set; }
}

```

When we execute the program, the result will be displayed as expected, as shown in the image below.



```

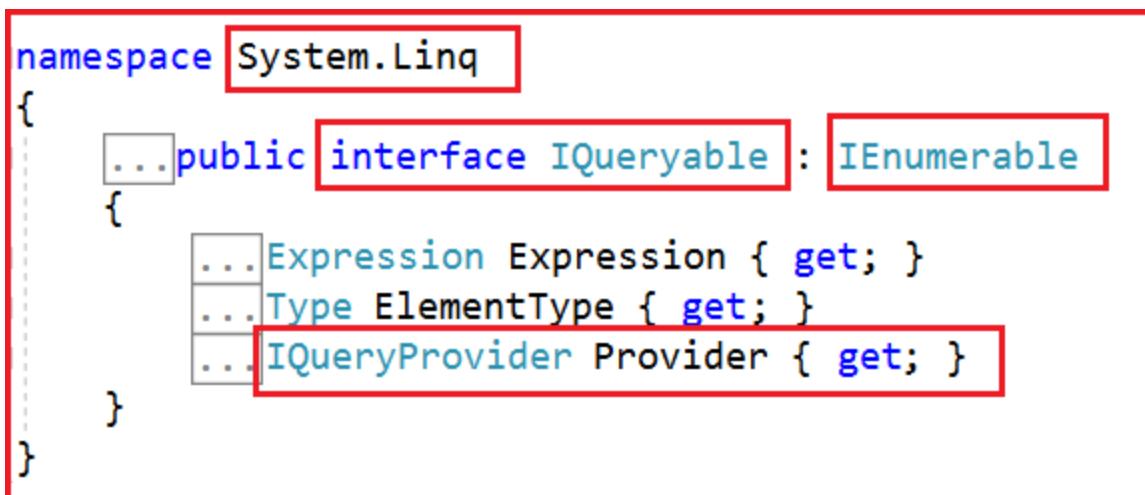
ID : 1 Name : James
ID : 3 Name : Steve

```

image\_201.png

## What is IQueryable in C#?

IQueryable in C# is an interface that is used to query data from a data source. It is part of the System.Linq namespace and is a key component in LINQ (Language Integrated Query). Unlike IEnumerable, which is used for iterating over in-memory collections, IQueryable is designed for querying data sources where the query is not executed until the object is enumerated. This is particularly useful for remote data sources, like databases, enabling efficient querying by allowing the query to be executed on the server side. The following is the definition of the IQueryable interface.



image\_202.png

### Example to Understand IQueryable Interface in C#.

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

namespace LINQDemo
{
 class Program
 {
 static void Main(string[] args)
 {
 List<Student> studentList = new List<Student>()
 {
 new Student(){ID = 1, Name = "James", Gender = "Male"},
```

```

 new Student(){ID = 2, Name = "Sara", Gender = "Female"},

 new Student(){ID = 3, Name = "Steve", Gender = "Male"},

 new Student(){ID = 4, Name = "Pam", Gender = "Female"}

 };

 //Linq Query to Fetch all students with Gender Male

 IQueryable<Student> MethodSyntax = studentList.AsQueryable()

 .Where(std => std.Gender == "Male");

 //Iterate through the collection

 foreach (var student in MethodSyntax)

 {

 Console.WriteLine($"ID : {student.ID} Name :

{student.Name}");

 }

 Console.ReadKey();

}

}

public class Student

{

 public int ID { get; set; }

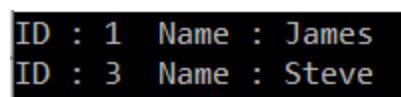
 public string Name { get; set; }

 public string Gender { get; set; }

}
}

```

Now run the application, and you will see the data as expected, as shown in the below image.



```

ID : 1 Name : James
ID : 3 Name : Steve

```

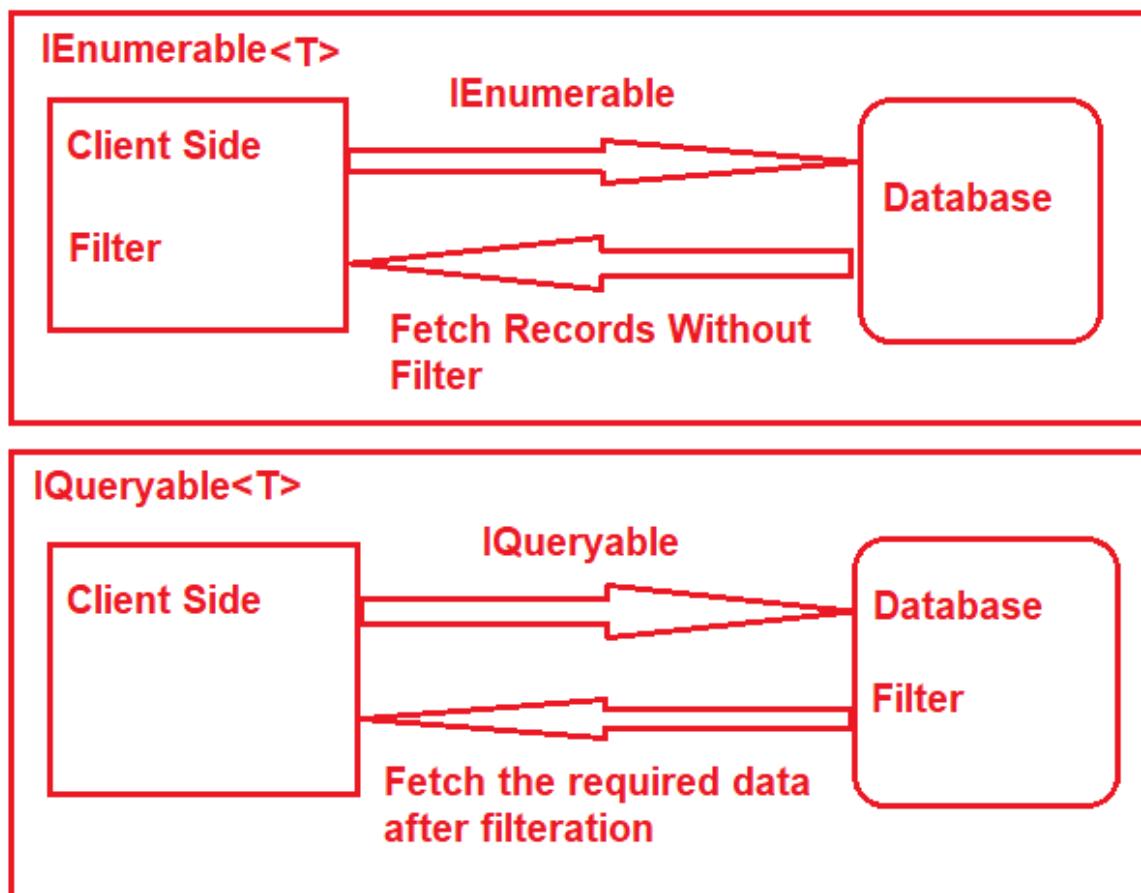
image\_203.png

**A** The point you need to remember is to return a collection of `IQueryable` type and call the `AsQueryable()` method on the data source, as we did in the above

example.

## Differences Between IEnumerable and IQueryable in C#

The `IEnumerable` and `IQueryable` in C# are used to hold a collection of data and also to perform data manipulation operations such as filtering, ordering, grouping, etc., based on the business requirements. This article will show you the difference between `IEnumerable` and `IQueryable` in C# with Examples. For a better understanding, please have a look at the following image. As you can see, `IEnumerable` fetches the record from the database without applying the filter. But `IQueryable` fetches the record from the database by applying the filter.



image\_204.png

# Fluent API in LINQ

LINQ (Language Integrated Query) is a powerful feature in C# that allows querying data in a declarative manner. Fluent API refers to the ability to chain LINQ method calls together to express complex queries clearly. In this document, we will explain how to use LINQ's Fluent API with a simple `Game` class example in a console application.

## Setup the Console Application

First, create a .NET console application. Define the `Game` class in `Game.cs` and add sample data in the `Program.cs`.

```
// Game.cs
using System;

namespace LearningLINQ
{
 internal class Game
 {
 public required string Title { get; set; }
 public string? Genre { get; set; }
 public int ReleaseYear { get; set; }
 public double Rating { get; set; }
 public int Price { get; set; }
 }
}
```

```
// Program.cs
using LearningLINQ;

var games = new List<Game>
{
 new Game { Title = "Galactic Frontier", Genre = "Sci-Fi Strategy",
ReleaseYear = 2022, Rating = 4.8, Price = 59 },
 new Game { Title = "Mystic Quest: The Hidden Temple", Genre =
"Adventure / Puzzle", ReleaseYear = 2021, Rating = 4.5, Price = 49 },
}
```

```
 new Game { Title = "CyberStrike Arena", Genre = "FPS Multiplayer",
ReleaseYear = 2023, Rating = 4.2, Price = 39 },
 new Game { Title = "Legends of the Shadow Realm", Genre = "Fantasy
RPG", ReleaseYear = 2020, Rating = 4.7, Price = 54 },
 new Game { Title = "Pixel Racer: Turbo Drift", Genre = "Arcade
Racing", ReleaseYear = 2023, Rating = 4.1, Price = 29 }
};
```

## Retrieving All Games

In the beginning, you may want to iterate through the entire list of games. This can be done using a `foreach` loop, but with LINQ, this can be handled more elegantly if needed for filtering or projection.

Example to retrieve all games (which would be similar to just using the `games` list directly):

```
foreach (var game in games)
{
 Console.WriteLine($"Title: {game.Title} | Genre: {game.Genre} |
Release Year: {game.ReleaseYear} | Rating: {game.Rating}");
}
```

## Filtering Data

LINQ provides a powerful `Where` method to filter data based on conditions. This is useful when you want to find games of a specific genre, or that meet certain criteria.

### Example: Filter by Genre

```
var sciFiGames = games.Where(game => game.Genre == "Sci-Fi Strategy");

foreach (var game in sciFiGames)
{
 Console.WriteLine($"Title: {game.Title} | Genre:
{game.Genre.ToLower()} | Release Year: {game.ReleaseYear} | Rating:
```

```
{game.Rating});
}
```

## Checking Conditions

LINQ also offers methods like `Any`, which allow you to check if any items in the collection satisfy a condition. This is useful for quick validations.

### Example: Check if there are modern games

```
var modernGamesExist = games.Any(game => game.ReleaseYear > 2022);
Console.WriteLine($"Does there exist a modern game?
{modernGamesExist}");
```

## Sorting Data

LINQ provides `OrderBy` and `OrderByDescending` methods to sort data in ascending or descending order based on one or more properties.

### Example: Sorting by Release Year (Ascending)

```
var sortedGamesByYear = games.OrderBy(game => game.ReleaseYear);
Console.WriteLine("Games sorted by release year (ascending):");
foreach (var game in sortedGamesByYear)
{
 Console.WriteLine($"Title: {game.Title} | Genre:
{game.Genre.ToLower()} | Release Year: {game.ReleaseYear} | Rating:
{game.Rating}");
}
```

### Example: Sorting by Release Year (Descending)

```
var sortedGamesByYearDesc = games.OrderByDescending(game =>
game.ReleaseYear);
Console.WriteLine("Games sorted by release year (descending):");
foreach (var game in sortedGamesByYearDesc)
{
 Console.WriteLine($"Title: {game.Title} | Genre:
```

```
{game.Genre.ToLower()} | Release Year: {game.ReleaseYear} | Rating:
{game.Rating});
}
```

## Aggregating Data

LINQ offers several aggregation methods such as `Average`, `Max`, and `Min` for performing calculations on data sets.

### Example: Calculate the Average Price

```
var averagePrice = games.Average(game => game.Price);
Console.WriteLine($"Average Game Price: ${averagePrice}");
```

### Example: Find the Game with the Highest Rating

```
var highestRating = games.Max(game => game.Rating);
var bestGame = games.First(g => g.Rating == highestRating);
Console.WriteLine($"Highest rated game: {bestGame.Title}
({bestGame.Rating})");
```

## Chaining Multiple LINQ Methods

One of the strengths of Fluent API in LINQ is the ability to chain multiple methods together for more complex queries. You can filter, sort, and then aggregate the data all in a single chain.

### Example: Filter, Sort, and Get Top Rated Game

```
var topRatedGame = games
.Where(game => game.ReleaseYear > 2020)
.OrderByDescending(game => game.Rating)
.First();

Console.WriteLine($"Top rated game since 2020: {topRatedGame.Title} |
Rating: {topRatedGame.Rating}");
```

## Projecting Data

You can also project data into different formats using the `Select` method. This can be useful when you only need certain fields or want to create new objects from the original data.

### Example: Project Games with Title and Rating

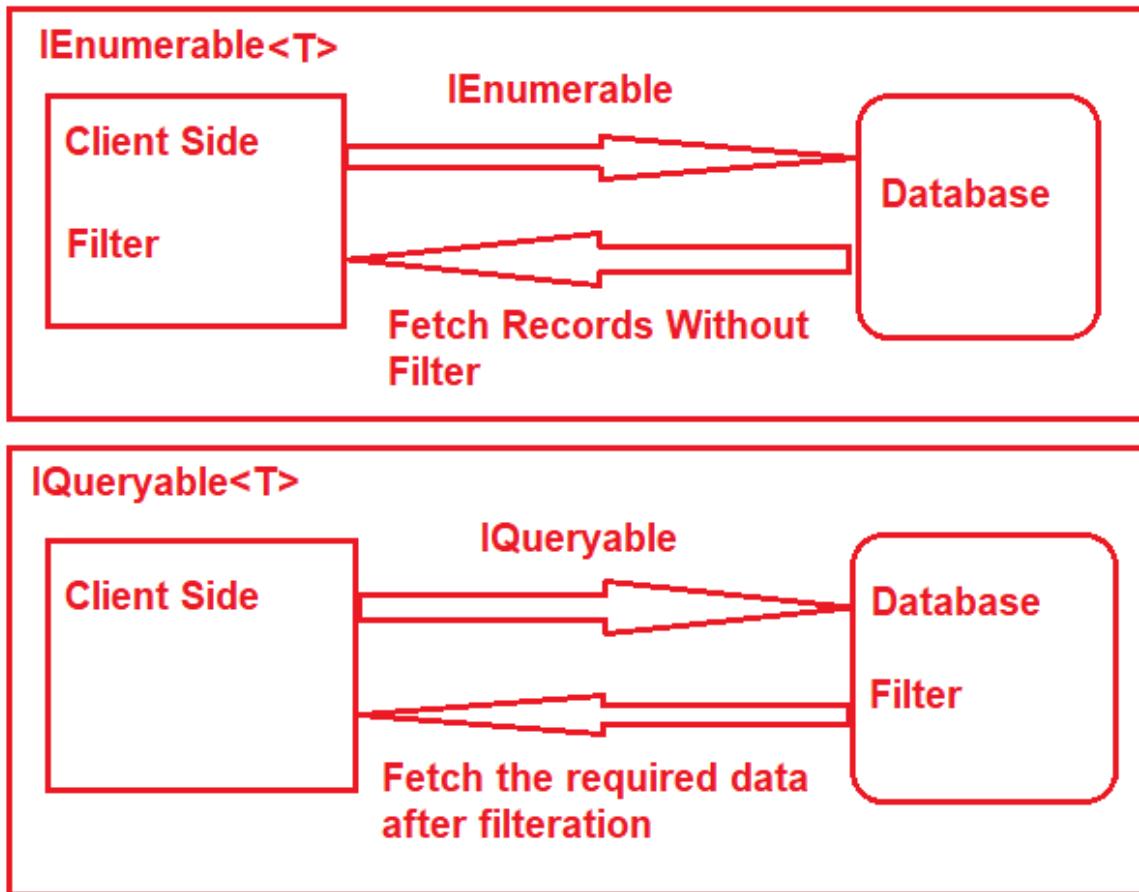
```
var gameTitlesAndRatings = games.Select(game => new { game.Title,
 game.Rating });

foreach (var item in gameTitlesAndRatings)
{
 Console.WriteLine($"Game: {item.Title} | Rating: {item.Rating}");
}
```

## Summary of LINQ Concepts Used

- **Where:** Filter data based on a condition.
- **Any:** Check if any elements in the collection satisfy a condition.
- **OrderBy/OrderByDescending:** Sort data in ascending or descending order.
- **Average/Max:** Perform aggregation like calculating averages or finding maximum values.
- **First:** Get the first element that satisfies a condition.
- **Select:** Project elements into a new shape or object.
- **Chaining Methods:** Combine multiple LINQ methods to achieve complex queries.

# IQueryable Workshop



image\_205.png

In this workshop, we will walk through the steps to create a Console Application that retrieves data from an SQL Server database using the Entity Framework Database First approach. We will focus on retrieving information from a Student table, as shown in the provided images and SQL script.

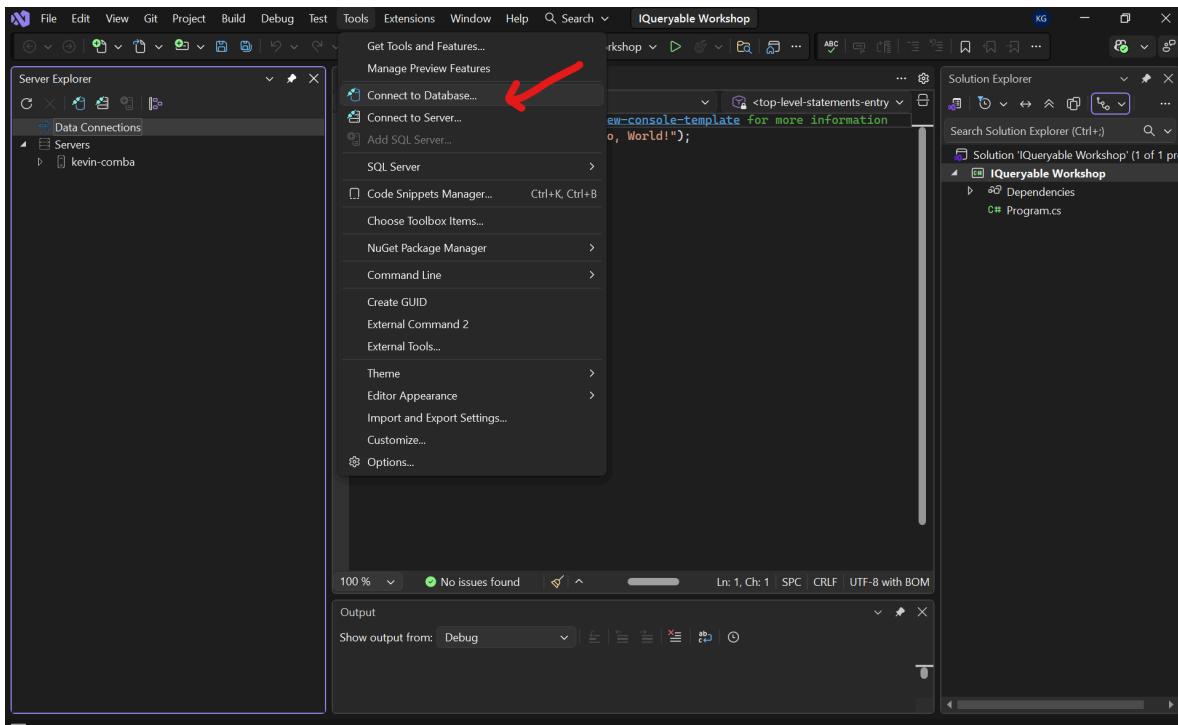
## Database Creation

ID	FirstName	LastName	Gender
101	Steve	Smith	Male
102	Sara	Pound	Female
103	Ben	Stokes	Male
104	Jos	Butler	Male
105	Pam	Semi	Female

image\_206.png

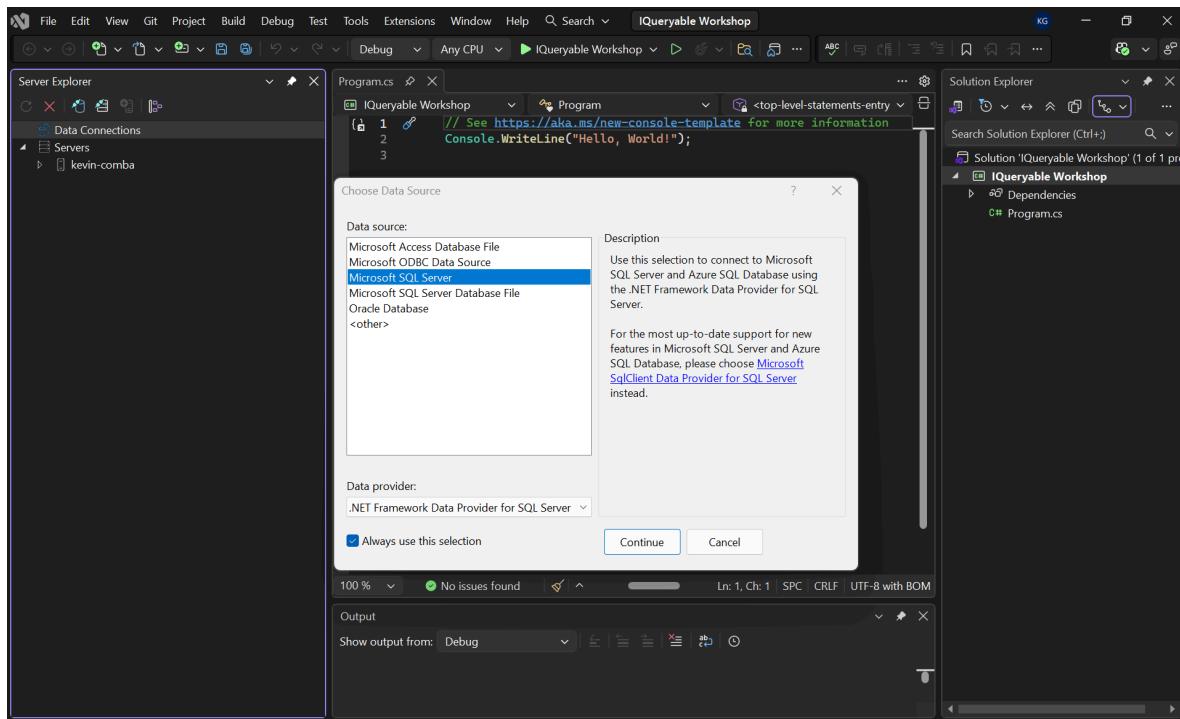
Create a new Console application call it IQueryable Workshop. Once you create the Console Application, add the ADO.NET Entity Data Model using the Database First Approach pointing to the above database.

Click on tools and select Connect to database



image\_208.png

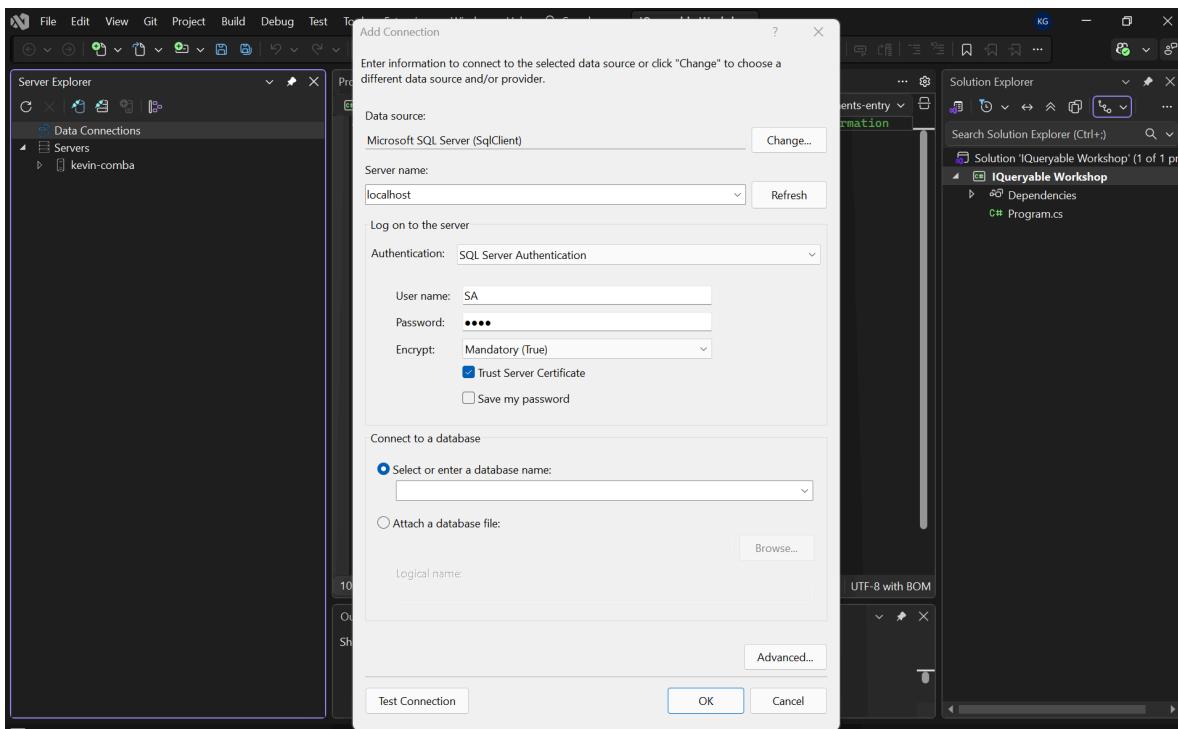
Select Microsoft SQL Server



image\_207.png

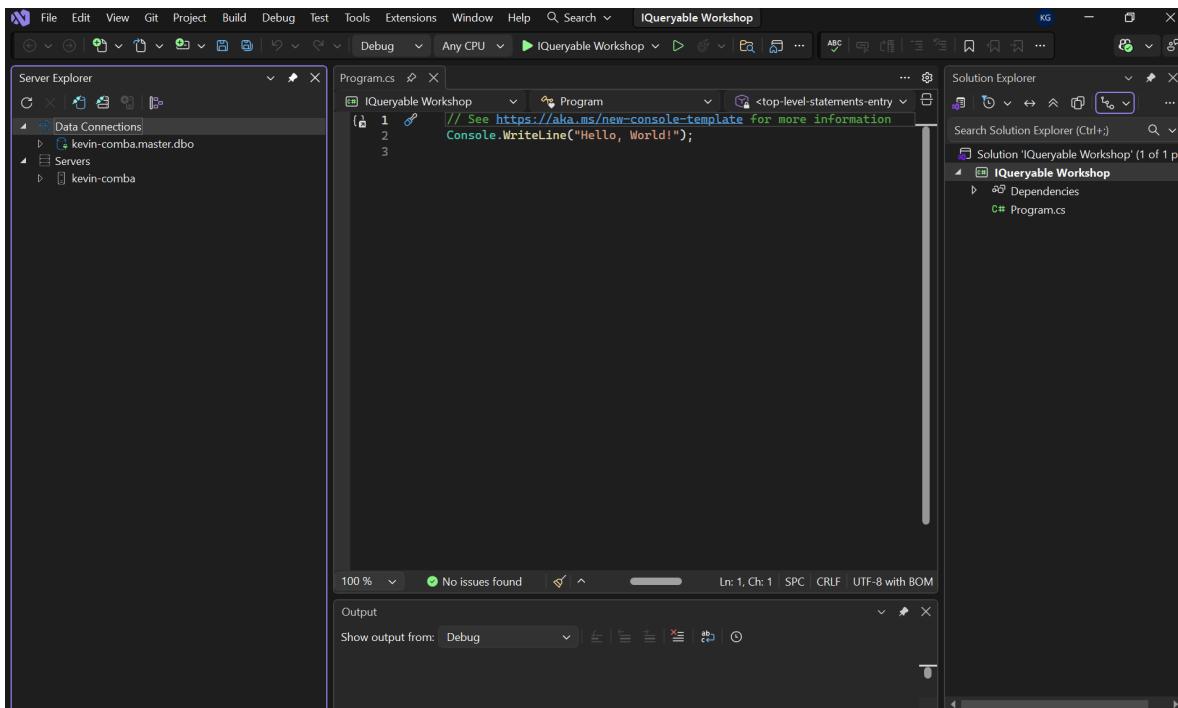
## Add

- Server name: localhost
- User name: SA
- Password :



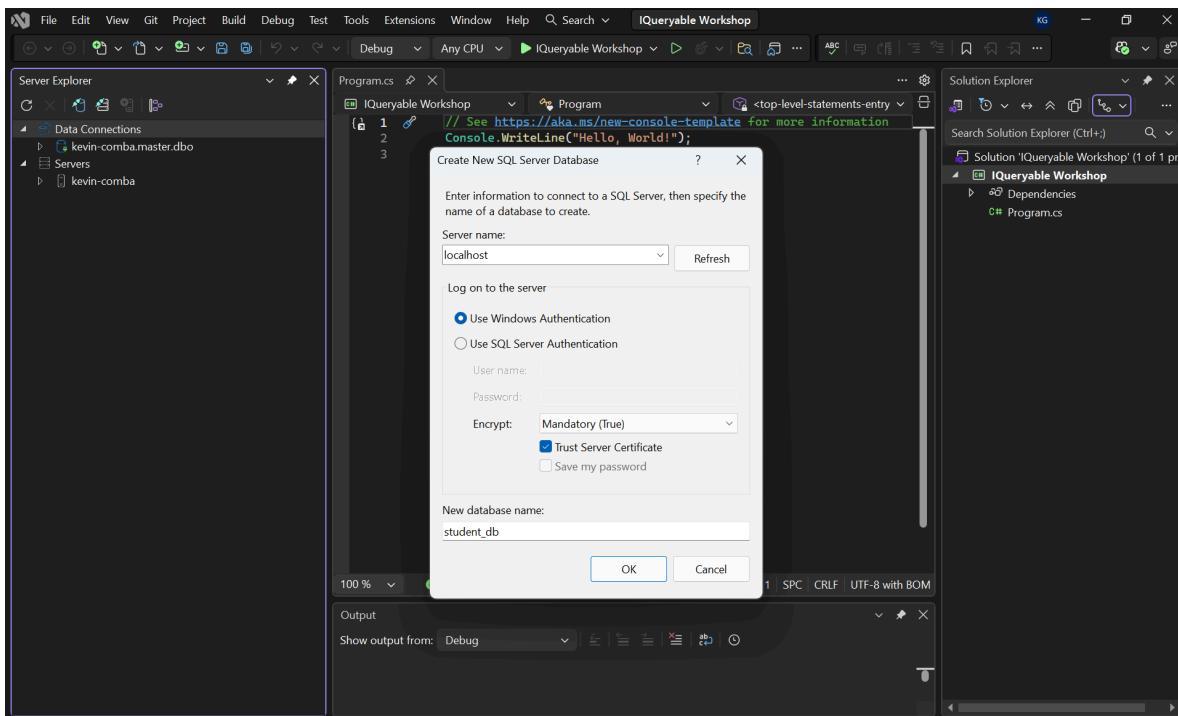
image\_209.png

After successfully connecting, you should see this.



image\_210.png

Right-click on data Connections and create a new SQL Server database.



image\_211.png

Right-click on your database and click on New Query, then paste the sql code below

```
-- Create the required Student table
CREATE TABLE Student
(
 ID INT PRIMARY KEY,
 FirstName VARCHAR(50),
 LastName VARCHAR(50),
 Gender VARCHAR(50)
)
GO

-- Insert the required test data
INSERT INTO Student VALUES (101, 'Steve', 'Smith', 'Male')
INSERT INTO Student VALUES (102, 'Sara', 'Pound', 'Female')
INSERT INTO Student VALUES (103, 'Ben', 'Stokes', 'Male')
INSERT INTO Student VALUES (104, 'Jos', 'Butler', 'Male')
INSERT INTO Student VALUES (105, 'Pam', 'Semi', 'Female')
GO
```

Then run the SQL script

A screenshot of the Visual Studio IDE interface. The top menu bar includes File, Edit, View, Git, Project, Build, Debug, SQL, Test, Tools, Extensions, Window, Help, and Search. The toolbar below has icons for Undo, Redo, Cut, Copy, Paste, Find, Replace, and others. The main window shows the Server Explorer on the left with a tree view of Data Connections (kevin-comba.master.dbo, kevin-comba.student\_db.dbo), Servers, and a selected item under kevin-comba.student\_db.dbo. The center pane displays an SQL query named 'SQLQuery1.sql' with the following code:

```
-- Create the required Student table
CREATE TABLE Student
(
 ID INT PRIMARY KEY,
 FirstName VARCHAR(50),
 LastName VARCHAR(50),
 Gender VARCHAR(50)
)
GO

-- Insert the required test data
INSERT INTO Student VALUES (101, 'Steve', 'Smith', 'Male')
INSERT INTO Student VALUES (102, 'Sara', 'Pound', 'Female')
INSERT INTO Student VALUES (103, 'Ben', 'Stokes', 'Male')
INSERT INTO Student VALUES (104, 'Jos', 'Butler', 'Male')
INSERT INTO Student VALUES (105, 'Pam', 'Semi', 'Female')
GO
```

The bottom status bar indicates the query was executed successfully at 8:24... on localhost (17.0 RC1) | AzureAD\KevinComba (95) | student\_db | 00:00:00 | 0 rows.

image\_212.png

Expand on tables to see your newly created table

A screenshot of the Visual Studio IDE interface, similar to the previous one but with the table structure expanded. The Server Explorer shows the expanded 'Student' table under 'Tables' with columns ID, FirstName, LastName, and Gender. The central pane shows the same SQL script as before, but the output window now displays the results of the insert statements:

```
(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)
```

The bottom status bar indicates the query was executed successfully at 8:24... on localhost (17.0 RC1) | AzureAD\KevinComba (95) | student\_db | 00:00:00 | 0 rows.

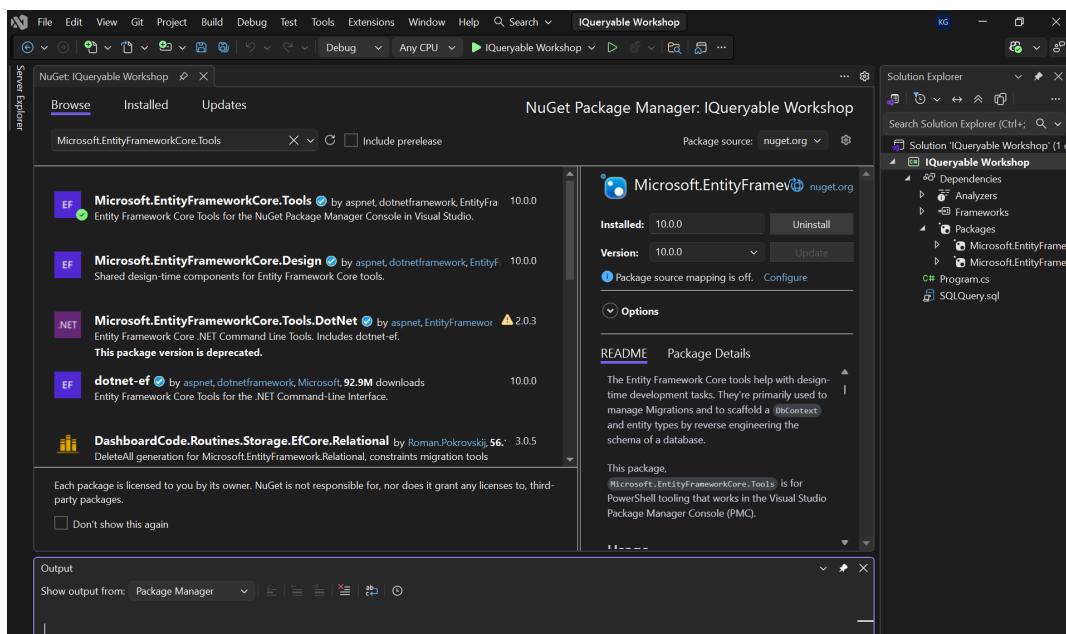
image\_213.png

# Create a New Console Application

- [DONE] Open Visual Studio and create a new Console Application project. Name it `IQueryableWorkshop`.
- After the project is created, you'll set up the Entity Framework to interact with your SQL Server database.

## Add Entity Data Model

- For EF Core (.NET 6/7/... Console App):
  - Install NuGet packages:
    - `Microsoft.EntityFrameworkCore.SqlServer`
    - `Microsoft.EntityFrameworkCore.Tools` These enable EF Core to work with SQL Server and design-time tooling.



image\_214.png

## Define your entity class(es)

For example, you have a Student table with columns ID, FirstName, LastName, Gender.  
You would create:

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

namespace IQueryable_Workshop
{
 internal class Student
 {
 public int ID { get; set; }
 public string FirstName { get; set; } = null!;
 public string LastName { get; set; } = null!;
 public string Gender { get; set; } = null!;
 }
}
```

## Create the DbContext class

This is the class through which you query and save data. Example:

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Text;

namespace IQueryable_Workshop
{
 internal class AppDbContext: DbContext
 {
 public DbSet<Student> Students { get; set; } = null!;
 protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
 {
 // Provide your connection string here
 optionsBuilder.UseSqlServer(
 @"Server=localhost;Database=YourDatabaseName;User
```

```

 Id=SA;Password=pass;");
 }

 protected override void OnModelCreating(ModelBuilder
modelBuilder)
{
 // Fluent API configurations (optional/customizations)
 modelBuilder.Entity<Student>(entity =>
 {
 entity.HasKey(e => e.ID);
 entity.Property(e =>
e.FirstName).HasMaxLength(50).IsRequired();
 entity.Property(e =>
e.LastName).HasMaxLength(50).IsRequired();
 entity.Property(e =>
e.Gender).HasMaxLength(50).IsRequired();
 entity.ToTable("Student");
 });

 // Similarly you can set up configuration for other
entities
}

}

```

### Explanation:

- `OnConfiguring` sets up the DB provider (SQL Server) and the connection string.
- `OnModelCreating` uses the Fluent API to configure mappings, constraints, table names etc.
- `DbSet<Student>` allows you to query the `Student` table.

### Use the context in your console app

In `Program.cs`:

```

using IQueryable_Workshop;

using (AppDbContext context = new AppDbContext())
{
 // Example 1: Filtering and limiting results
 IQueryable<Student> topMaleStudents = context.Students
 .Where(x => x.Gender == "Male")
 .Take(2);

 Console.WriteLine("Top 2 Students Where Gender = Male");
 Console.WriteLine("=====");
 foreach (var s in topMaleStudents)
 {
 Console.WriteLine($"ID: {s.ID}, Name: {s.FirstName}
{s.LastName}");
 }

 // Example 2: Sorting with multiple criteria
 IQueryable<Student> sortedStudents = context.Students
 .OrderByDescending(x =>
x.FirstName)
 .ThenBy(x => x.LastName)
 .Take(5);

 Console.WriteLine("\nTop 5 Students Sorted by FirstName Descending
and LastName Ascending");

 Console.WriteLine("=====");
 foreach (var s in sortedStudents)
 {
 Console.WriteLine($"ID: {s.ID}, Name: {s.FirstName}
{s.LastName}");
 }

 // Example 3: Count aggregation
 int totalStudents = context.Students.Count();
 Console.WriteLine($"\\nTotal Number of Students: {totalStudents}");
}

```

```

// Example 4: Count with filtering
int maleCount = context.Students.Count(s => s.Gender == "Male");
int femaleCount = context.Students.Count(s => s.Gender == "Female");
Console.WriteLine($"Male Students: {maleCount}");
Console.WriteLine($"Female Students: {femaleCount}");

// Example 5: Group by with count
var genderGroups = context.Students
 .GroupBy(s => s.Gender)
 .Select(g => new
 {
 Gender = g.Key,
 Count = g.Count()
 })
 .ToList();

Console.WriteLine("\nStudents Grouped by Gender:");
Console.WriteLine("=====");
foreach (var group in genderGroups)
{
 Console.WriteLine($"Gender: {group.Gender}, Count: {group.Count}");
}

// Example 6: Max and Min operations
var maxId = context.Students.Max(s => s.ID);
var minId = context.Students.Min(s => s.ID);
Console.WriteLine($"Highest Student ID: {maxId}");
Console.WriteLine($"Lowest Student ID: {minId}");

// Example 7: First and FirstOrDefault
var firstStudent = context.Students.FirstOrDefault();
if (firstStudent != null)
{
 Console.WriteLine($"First Student: {firstStudent.FirstName}
{firstStudent.LastName}");
}

// Example 8: Any and All operators

```

```

 bool hasMaleStudents = context.Students.Any(s => s.Gender ==
"Male");
 bool allHaveFirstName = context.Students.All(s =>
!string.IsNullOrEmpty(s.FirstName));
 Console.WriteLine($"\\nHas Male Students: {hasMaleStudents}");
 Console.WriteLine($"All Students Have First Name:
{allHaveFirstName}");

// Example 9: Complex query with multiple operations
var complexQuery = context.Students
 .Where(s => s.Gender == "Male")
 .OrderBy(s => s.LastName)
 .ThenBy(s => s.FirstName)
 .Skip(1)
 .Take(2)
 .Select(s => new
{
 FullName = $"{s.FirstName}"
{s.LastName}",
 s.Gender
})
 .ToList();
 Console.WriteLine("Complex Query (Skip first male student, take
next 2):");

Console.WriteLine("=====");
foreach (var item in complexQuery)
{
 Console.WriteLine($"Name: {item.FullName}, Gender:
{item.Gender}");
}

Console.WriteLine("\\nPress any key to exit...");
Console.ReadKey();
}

```

## What is IQueryables<Student>

- The type `IQueryable<Student>` represents a query that hasn't yet been executed; it builds an expression tree that the EF Core provider will translate into SQL and send to the database.
- Use of `IQueryable` (instead of something like `IEnumerable`) means that filtering, sorting, limiting, etc., will be executed in the database rather than in memory — which is more efficient.

## Walking through the above Code

### 1. Filtering + Limiting results

```
IQueryable<Student> topMaleStudents = context.Students
 .Where(x => x.Gender == "Male")
 .Take(2);
```

- `Where(x => x.Gender == "Male")`: builds a condition to only include students whose Gender is “Male”.
- `Take(2)`: limits the result to the first two of those matched rows.
- Because this is `IQueryable`, the WHERE + TOP(2) (or equivalent) will be included in the SQL sent to the database — only two male student rows will be fetched.

### 2. Sorting with multiple criteria

```
IQueryable<Student> sortedStudents = context.Students
 .OrderByDescending(x => x.FirstName)
 .ThenBy(x => x.LastName)
 .Take(5);
```

- `OrderByDescending(x => x.FirstName)`: sort students so those with later (alphabetically) first names come first.
- `.ThenBy(x => x.LastName)`: for students with the same first name, sort by last name (ascending).

- `.Take(5)`: pick only the top 5 after sorting.
- This entire sorting + limiting is executed in the database side.

### 3. Count aggregation

```
int totalStudents = context.Students.Count();
```

- `Count()` is a terminal method (on `IQueryable`) which causes the query to execute and only retrieves the count (not the full entities).
- Efficient: the database counts rows and returns just the number.

### 4. Count with filtering

```
int maleCount = context.Students.Count(s => s.Gender == "Male");
int femaleCount = context.Students.Count(s => s.Gender == "Female");
```

- This uses overload of `Count(predicate)` to count only those rows matching the predicate.
- Recognized by EF Core and translated into something like `SELECT COUNT(*) FROM Students WHERE Gender = 'Male'`.

### 5. Group by with count

```
var genderGroups = context.Students
 .GroupBy(s => s.Gender)
 .Select(g => new { Gender = g.Key, Count = g.Count() })
 .ToList();
```

- `GroupBy(s => s.Gender)`: group students by their gender value.
- The `Select(...)` then projects each group into an anonymous object with the gender key and the number of students in that group (`g.Count()`).
- `.ToList()` triggers execution — fetching the grouped results from the database.

### 6. Max and Min operations

```
var maxId = context.Students.Max(s => s.ID);
var minId = context.Students.Min(s => s.ID);
```

- `Max` & `Min` are also terminal methods. They ask the database to compute the maximum / minimum `ID` value, rather than retrieving all students then computing in memory.

## 7. First and FirstOrDefault

```
var firstStudent = context.Students.FirstOrDefault();
```

- `FirstOrDefault()` will return the first entity from the result (or `null` if none). Because it's `IQueryable`, the translation ensures only one row is retrieved from DB.

## 8. Any and All operators

```
bool hasMaleStudents = context.Students.Any(s => s.Gender == "Male");
bool allHaveFirstName = context.Students.All(s =>
 !string.IsNullOrEmpty(s.FirstName));
```

- `Any(predicate)`: returns true if **any** row matches the condition (efficient — stops once found in DB).
- `All(predicate)`: returns true if **all** rows satisfy the condition.
- Both are translated to optimized SQL.

## 9. Complex query with multiple operations

```
var complexQuery = context.Students
 .Where(s => s.Gender == "Male")
 .OrderBy(s => s.LastName)
 .ThenBy(s => s.FirstName)
 .Skip(1)
 .Take(2)
 .Select(s => new { FullName = $"{s.FirstName} {s.LastName}", s.Gender })
 .ToList();
```

- Step by step:
  - Filter male students.
  - Sort by LastName, then by FirstName.
  - Skip(1): skip the first result after sorting.
  - Take(2): then take the next two.
  - Select(...): project into an anonymous type with FullName and Gender.
  - ToList(): triggers execution.
- All of this is combined into a single SQL query that does filtering, ordering, skipping, limiting and projection on the database side — minimizing data transferred.

## Key Points to Keep in Mind

- **Deferred execution:** The query isn't executed when you declare it (e.g., var query = context.Students.Where(...)), but when you iterate it (foreach) or call a materializing operation (ToList(), Count(), etc.). ([DEV Community][3])
- **Query composition:** Because the query remains an IQueryable, you can keep chaining filters/sorts etc, and EF will build a single query expression tree and then translate efficiently.
- **Pushing work to the database:** Using IQueryable ensures that operations like filtering, sorting, grouping happen in the database rather than fetching everything into memory and doing them in the app — which is better for performance with large datasets. ([Stack Overflow][4])
- **Materializing vs non-materializing methods:** Methods like Where, OrderBy, Take, Skip, GroupBy, Select build up the query. Methods like ToList, Count, FirstOrDefault, Any, etc. execute the query.
- **Be aware of what queries are generated:** Because you're using IQueryable, the LINQ provider (EF Core) will translate your expression tree into SQL — not all .NET methods are translatable, and adding extra operations after materialization can degrade performance.



# Entity Framework

Before .NET 3.5 as a developer, we often used to write ADO.NET code to perform CRUD operations with the underlying database.

For this, we need to create a Connection Object with the database, then Open the Connection, Create the Command Object and execute the Command using Data Reader or Data Adapter. And then we create DataSet or DataTables to store the data in memory to perform different types of Operations on the Data as per the business requirements. Actually, this is a Time-Consuming, and Error-Prone Process.

Microsoft has provided a framework called “Entity Framework” to automate all these database-related activities for our application and for this to work, we just need to provide the necessary details to the Entity Framework.

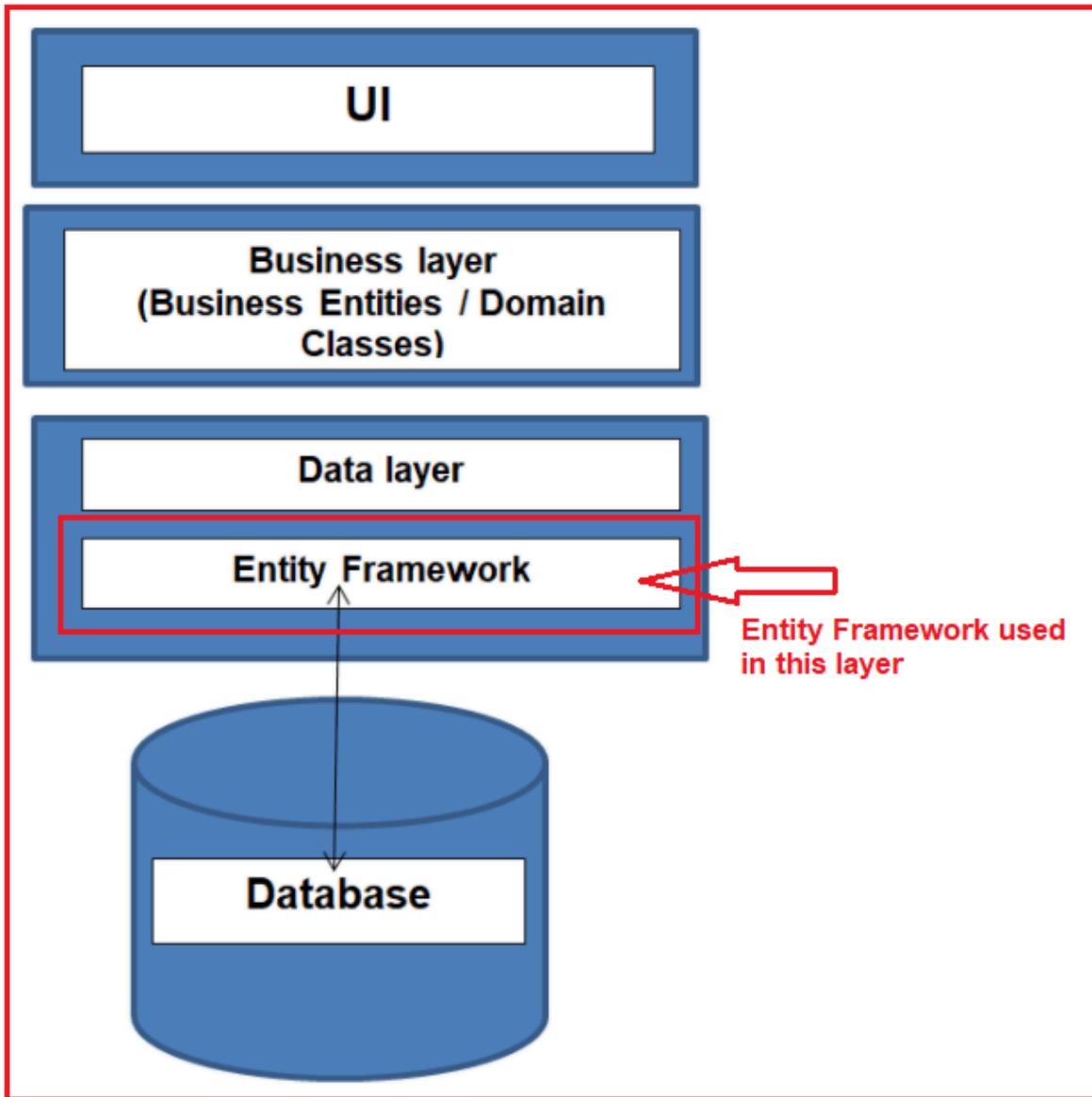
## What is the Entity Framework?

Entity Framework is an **Open-Source Object-Relational Mapping (ORM) Framework** for .NET applications that enables .NET developers to work with relational data using domain-specific objects without focusing on the underlying database tables and columns where actually the data is stored.

**⚠ Official Definition:** Entity Framework is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write.

## What is an Object-Relational Mapping Framework?

Object Relational Mapping framework automatically creates classes based on database tables, and vice versa is also true, that is, it can also automatically generate the necessary SQL to create database tables based on classes.



image\_215.png

The above diagram shows that the Entity Framework fits between the Data Layer and the database. It saves the data in the database which are stored in the properties of the business entities (domain classes) and also retrieves the data from the database and converts it to business entities objects automatically.

## Why do we need to use Entity Framework in .NET Applications?

Entity Framework is an ORM Tool and ORMs Tools are basically used to increase the developer's productivity by reducing the redundant task of doing CRUD operation

against a database in a .NET Application.

1. Entity Framework can generate the necessary database commands for doing the database CRUD Operation i.e. can generate **SELECT, INSERT, UPDATE** and **DELETE** commands for us.
2. While working with Entity Framework, we can perform different types of operations on the domain objects (basically classes representing database tables) using LINQ to entities.
3. Entity Framework will generate and execute the SQL Command in the database and then store the results in the instances of your domain objects so that you can do different types of operations on the data.

## Entity Framework Features:

1. **Cross-platform:** EF Core is a cross-platform framework which means it can run on Windows, Linux, and Mac operating systems
2. **Modeling:** EF (Entity Framework) creates an EDM (Entity Data Model) based on POCO (Plain Old CLR Object) entities with get/set properties of different data types. It uses this model also when querying or saving entity data to the underlying database.
3. **Querying:** EF allows us to use LINQ queries (C#/VB.NET) to retrieve data from the underlying database. The database provider will translate these LINQ queries to the database-specific query language (e.g. SQL for a relational database). EF also allows us to execute raw SQL queries directly to the database.
4. **Change Tracking:** EF keeps track of changes that occurred to instances of our entities (Property values) that need to be submitted to the database.
5. **Saving:** EF executes INSERT, UPDATE, and DELETE commands to the database based on the changes that occurred to our entities when we call the SaveChanges() method. EF also provides the asynchronous SaveChangesAsync() method
6. **Concurrency:** EF uses Optimistic Concurrency by default to protect against overwriting changes made by another user since data was fetched from the database.

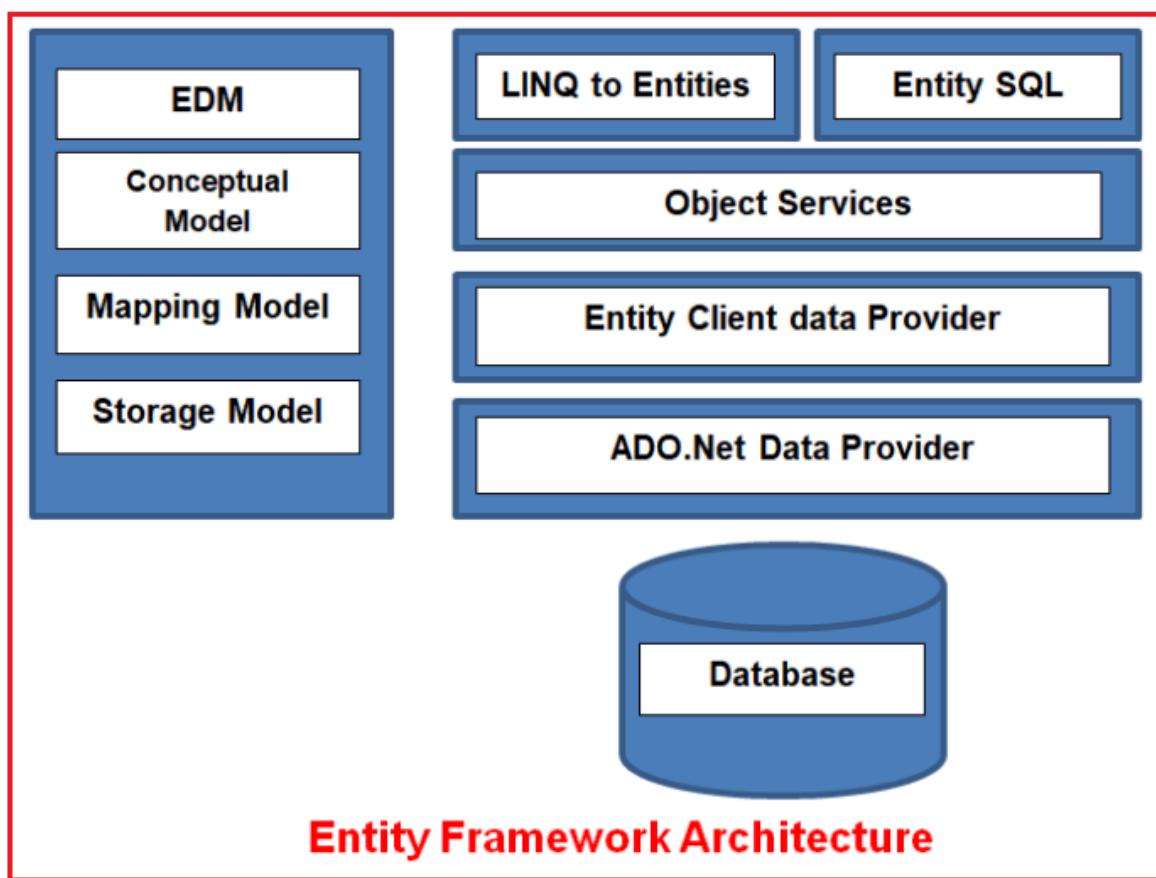
- 7. Transactions:** EF performs automatic transaction management while querying or saving data. It also provides options to customize transaction management.
- 8. Caching:** EF includes the first level of caching out of the box. So, repeated querying will return data from the cache instead of hitting the database.
- 9. Built-in Conventions:** EF follows conventions over the configuration programming pattern, and includes a set of default rules which automatically configure the EF model.
- 10. Configurations:** EF allows us to configure the EF model by using data annotation attributes or Fluent API to override default conventions.

# Entity Framework Architecture

The Architecture of the Entity Framework is composed of the following components

1. The Entity Data Model
2. LINQ to Entities
3. Entity SQL
4. The Object Services Layer
5. Entity Client Data Provider
6. ADO.NET Data Provider

The following diagram shows the overall architecture of the Entity Framework.



image\_216.png

## **EDM (Entity Data Model):**

The Entity Data Model (EDM) abstracts the logical or the relational schema and exposes the conceptual schema of the data using a three-layered approach i.e.

1. The Conceptual Model (C- Space) : The conceptual model contains the model classes (i.e. entities) and their relationships. This will be independent of your database table design. It defines your business objects and their relationships in XML files.
2. Mapping model (C-S Space): A Mapping Model consists of information about how the conceptual model is mapped to the storage model. The Mapping model is responsible for mapping the conceptual and logical layers (Storage layer). It maps the business objects and the relationships defined in the conceptual layer with the tables and relationships defined in the logical layer.
3. Storage model (S – Space): The storage model represents the schema of the underlying database. That means the storage model is the database design model which includes tables, views, Keys, stored procedures, and their relationships. The Entity Data Model uses the following three types of XML files to represent the C-Space, C-S Space, and the S-Space respectively.

## **LINQ to Entities:**

LINQ-to-Entities (L2E) is a query language used to write queries against the object model. It returns entities, which are defined in the conceptual model. You can use your LINQ skills here.

## **Entity SQL:**

Entity SQL is another query language (For EF 6 only) just like LINQ to Entities. However, it is a little more difficult than LINQ-to-Entities (L2E) and the developer will have to learn it separately. These E-SQL queries are internally translated to data store-dependent SQL queries

## **Object Service:**

The Object Services layer is the Object Context, which represents the session of interaction between the applications and the data source.

- The main use of the Object Context is to perform different operations like add, update and delete instances of entities and to save the changed state back to the database with the help of queries.
- It is the ORM layer of Entity Framework, which represents the data result to the object instances of entities.
- This service allows the developer to use some of the rich ORM features like primary key mapping, change tracking, etc. by writing queries using LINQ and Entity SQL.

Apart from this, the Object Services Layer provides the following additional services:

1. Change tracking
2. Lazy loading
3. Inheritance
4. Optimistic concurrency
5. Merging data
6. Identity resolution
7. Support for querying data using Entity SQL and LINQ to Entities

## **Entity Client Data Provider:**

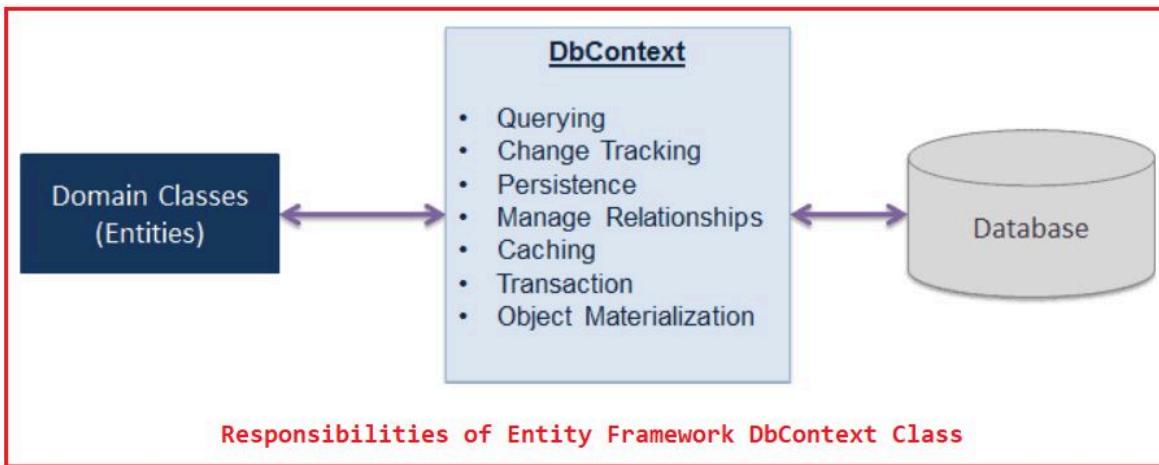
The main responsibility of this layer is to convert LINQ-to-Entities or Entity SQL queries into a SQL query that is understood by the underlying database. It communicates with the ADO.Net data provider which in turn sends or retrieves data from the database.

## **ADO.Net Data Provider:**

This layer communicates with the database using standard ADO.Net.

# DbContext in EF

The DbContext class is one of the important classes in Entity Framework. It is a bridge between your domain or entity classes and the underlying database. For a better understanding, please have a look at the below image.



image\_217.png

So, the following are the major responsibilities performed by the DbContext class:

1. **Querying:** Converts LINQ-to-Entities queries to SQL queries and then sends those queries to the database for execution.
2. **Change Tracking:** DbContext class also keeps track of the changes that occurred on the entities after querying from the database. For example, after fetching the data from the database, you have updated a few data, insert new data and delete some existing data. This information will be tracked by the DbContext class.
3. **Persisting Data:** The DbContext class is also responsible for performing the Insert, Update and Delete operations to the database, based on entity states.
4. **Caching:** DbContext class also provides first-level caching by default. In our upcoming articles, we will discuss this concept in detail with Examples.
5. **Transaction:** It also provides the support to implement Transactions using the Unit of Work and Repository Pattern.

6. **Manage Relationship:** This class is also responsible for managing the relationships between the entities using CSDL, MSL, and SSDL in Db-First or Model-First approach, and using fluent API in the Code-First approach. We will discuss CSDL, MSL, SSDL, and Fluent API in detail in our coming articles.
7. **Object Materialization:** It is also used to convert the raw data which is retrieved from the database into entity objects.

## DbContext Methods in Entity Framework:

Now, let us proceed further and try to understand the important methods provided by the DbContext class in Entity Framework.

- **Entry(object entity):** It is used to get a DbEntityEntry object for the given entity providing access to information about the entity and the ability to perform actions on the entity.
- **SaveChanges:** Saves all changes made in the context object to the underlying database. That means, it executes `INSERT`, `UPDATE`, and `DELETE` commands to the database for the entities with `Added`, `Modified`, and `Deleted` states.
- **SaveChangesAsync:** This method is used to asynchronously saves all changes made in this context to the underlying database. That means, it executes `INSERT`, `UPDATE`, and `DELETE` commands to the database for the entities with `Added`, `Modified`, and `Deleted` states asynchronously.
- **Set(Type entityType):** This method returns a non-generic DbSet instance for access to entities of the given type in the context and the underlying database. Here, the parameter `entityType` specifies the type of entity for which a set should be returned. That means it creates a `DbSet< TEntity >` that can be used to query and save instances of `TEntity` into the database.
- **OnModelCreating:** This method is called when the model for a derived context has been initialized, but before the model has been locked down and used to initialize the context. The default implementation of this method does nothing, but it can be overridden in a derived class such that the model can be further configured before it is locked down.

## Entity Framework DbContext Class Properties

1. **ChangeTracker { get; }**: This property provides access to features of the context that deal with change tracking of entities. This is a read-only property.
2. **Configuration { get; }**: This property provides access to configuration options for the context. This is a read-only property.
3. **Database { get; }**: This property creates a Database instance for the context object that allows for creation/deletion/existence checks for the underlying database. This is a read-only property.

# Entities in Entity Framework

At the end of this article, you will understand the following pointers in detail with Examples.

- What is an Entity in the Entity Framework?
- Understanding Scalar Property and Navigation Property

## What is an Entity in the Entity Framework?

```
public partial class EF_Demo_DBEntities : DbContext
{
 public EF_Demo_DBEntities()
 : base("name=EF_Demo_DBEntities")
 {
 }

 protected override void OnModelCreating(DbModelBuilder modelBuilder)
 {
 throw new UnintentionalCodeFirstException();
 }

 public virtual DbSet<Department> Departments { get; set; }
 public virtual DbSet<Employee> Employees { get; set; }
}
```

↑ Entities

image\_218.png

As you can see in the above image, Departments property type is `DbSet<Department>` and Employees property type is `DbSet<Employee>`. So, here, Department and Employee are nothing but entities. An Entity in Entity Framework is a class that is included as a `DbSet< TEntity >` type property in the derived context class.

Entity Framework maps each entity to a database table and each property of an entity is mapped to a column in the database table. In our example, the Department entity is mapped with the Departments database table and the Employee entity is mapped with the Employees database table.

## Scalar Property in C#:

The Primitive Type Properties defined inside a class are called Scalar Properties. Scalar property stores the actual data. A scalar property maps to a single column in the database table. For example, in the Employee class below are the scalar properties

```
public int ID { get; set; }
public string Name { get; set; }
public string Email { get; set; }
public string Gender { get; set; }
public Nullable<int> Salary { get; set; }
public Nullable<int> DepartmentId { get; set; }
```

Similarly, in the Department Class Below are the Scalar properties

```
public int ID { get; set; }
public string Name { get; set; }
public string Location { get; set; }
```

## Navigation Property in C#:

The Navigation Property represents a relationship with another Entity. There are two types of navigation properties. They are as follows:

1. Reference Navigation Property
2. Collection Navigation Property

### Reference Navigation Property in C#:

If an entity includes a property of another entity type, then it is called a Reference Navigation Property in C#. It represents the multiplicity of one (1) i.e. it represents the one-to-one relationship between the entities. For example, in Employee Class, the following property is a Reference Navigation property. This indicates one Employee belongs to one Department.

```
public virtual Department Department { get; set; }
```

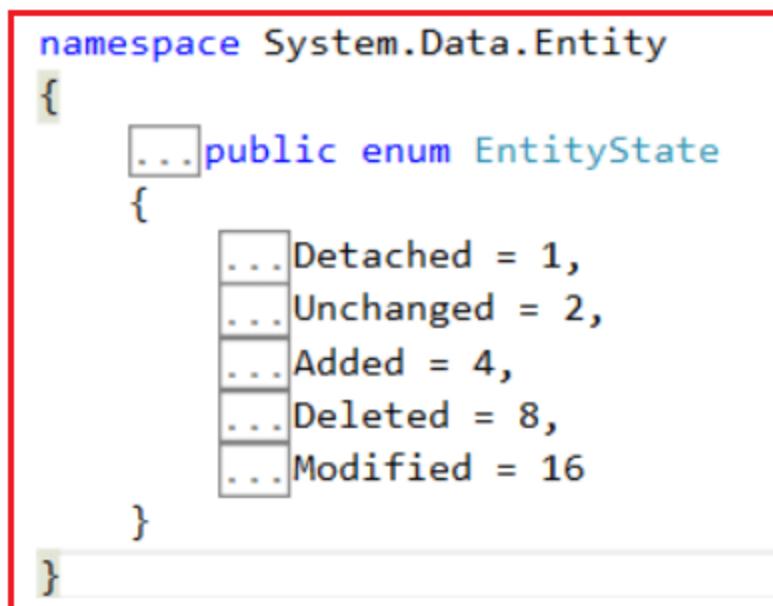
## **Collection Navigation Property in C#:**

If an entity includes a property of collection type, it is called a Collection Navigation Property in C#. It represents the multiplicity of many (\*) i.e. represents one-to-many relationships. For example, in the Department class following property is a collection navigation property. This indicates that one Department has many employees.

```
public virtual ICollection<Employee> Employees { get; set; }
```

# Entity States in Entity Framework

The Entity Lifecycle in Entity Framework describes the process in which an Entity is created, added, modified, deleted, etc. Entities have many states during their lifetime. Entity Framework maintains the state of each entity during its lifetime. Each entity has a state based on the operation performed on it via the context class (the class which is derived from DbContext class).



image\_219.png

That means the Entity State represents the state of an entity. An entity is always in any one of the following states.

- **Added:** The entity is marked as added. The entity is being tracked by the context but does not yet exist in the database.
- **Deleted:** The entity is marked as deleted. The entity is being tracked by the context and exists in the database, but has been marked for deletion from the database the next time SaveChanges is called.
- **Modified:** The entity has been modified. The entity is being tracked by the context and exists in the database, and some or all of its property values have been modified.
- **Unchanged:** The entity hasn't been modified. The entity is being tracked by the context and exists in the database, and its property values have not changed from the

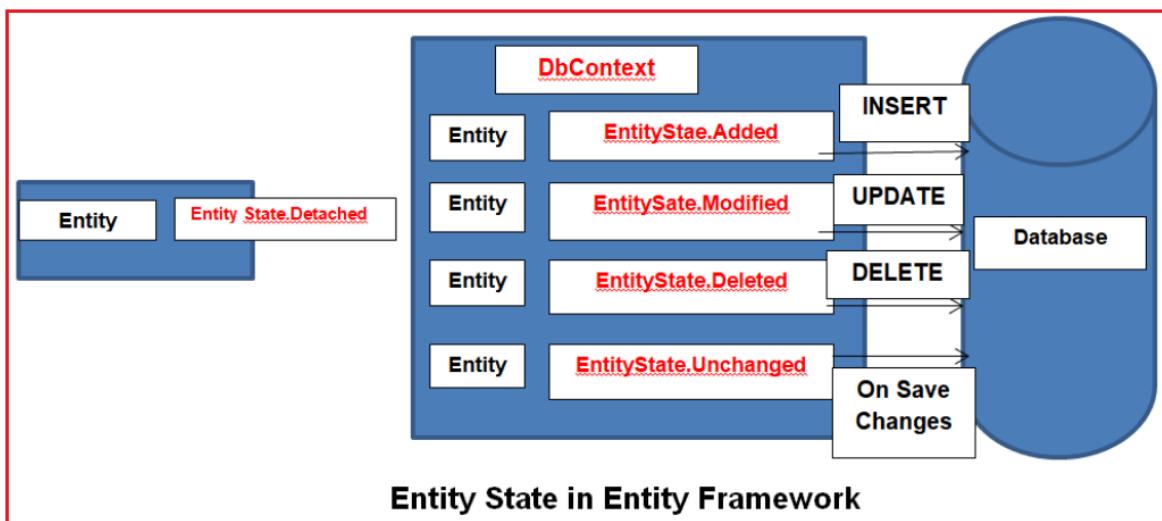
values in the database.

- **Detached:** The entity is not being tracked by the context. An entity is in this state immediately after it has been created with the new operator or with one of the System.Data.Entity.DbSet Create methods.

The Context object not only holds the reference to all the entity objects as soon as retrieved from the database but also keeps track of entity states and maintains modifications made to the properties of the entity. This feature is known as Change Tracking.

## Entity Lifecycle in Entity Framework

The change in the entity state from the Unchanged to the Modified state is the only state that's automatically handled by the context class. All other changes must be made explicitly by using the proper methods of DbContext class. The following diagram shows the different states of an entity in the Entity Framework.



image\_220.png

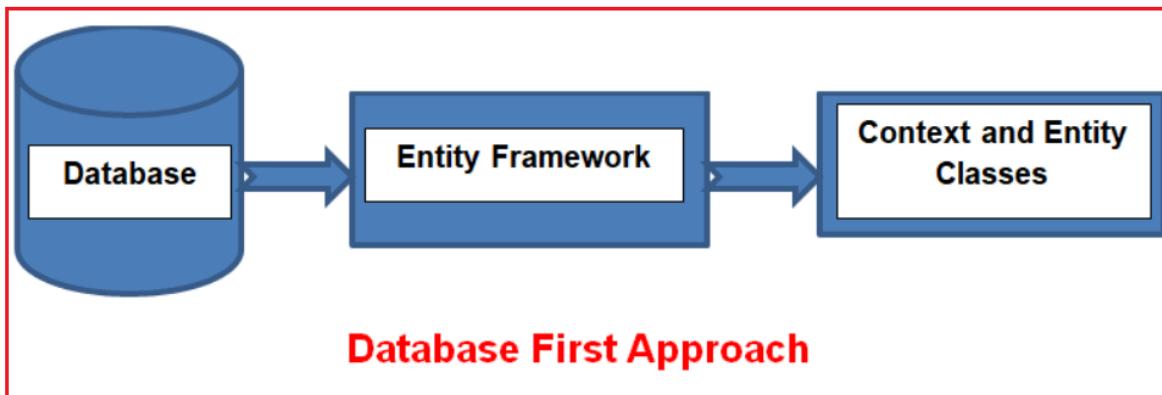
# Development Approach with Entity Framework

The entity framework provides three different approaches when working with the database and data access layer in your application as per your project requirement. These are

1. Database-First
2. Code-First
3. Model-First

## Database-First Approach of Entity Framework:

We can use the **Database First Approach** if the database schema already exists. In this approach, we generate the context and entities for the existing database using the EDM wizard. This approach is best suited for applications that use an already existing database.



image\_221.png

Database First approach is useful:

- When we are working with a legacy database
- If we are working in a scenario where the database design is being done by another team and the application development starts only when the database is ready

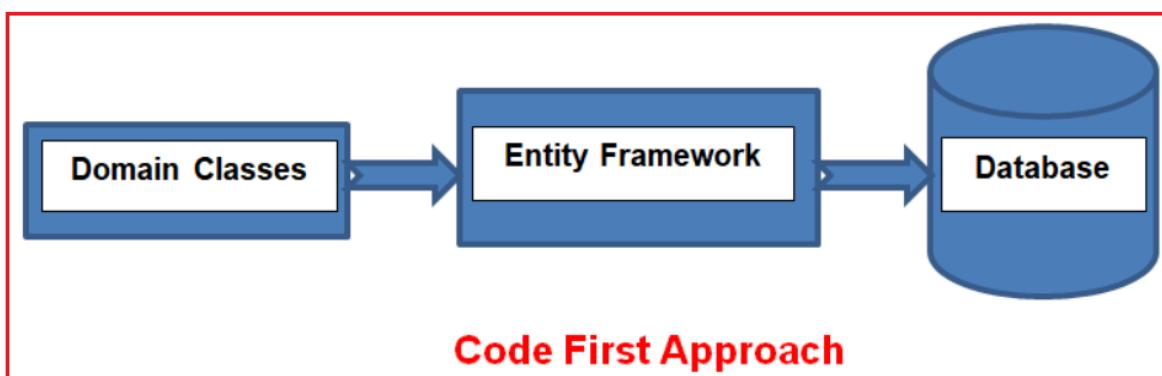
- When we are working on a data-centric application

## **Code-First Approach of Entity Framework:**

You can use the Code First approach when you do not have an existing database for your application. In the code-first approach, you start writing your entities (domain classes) and context class first and then create the database from these classes using migration commands.

This approach is best suited for applications that are highly domain-centric and will have the domain model classes created first. The database here is needed only as a persistence mechanism for these domain models.

That means Developers who follow the Domain-Driven Design (DDD) principles, prefer to begin coding with their domain classes first and then generate the database required to persist their data.



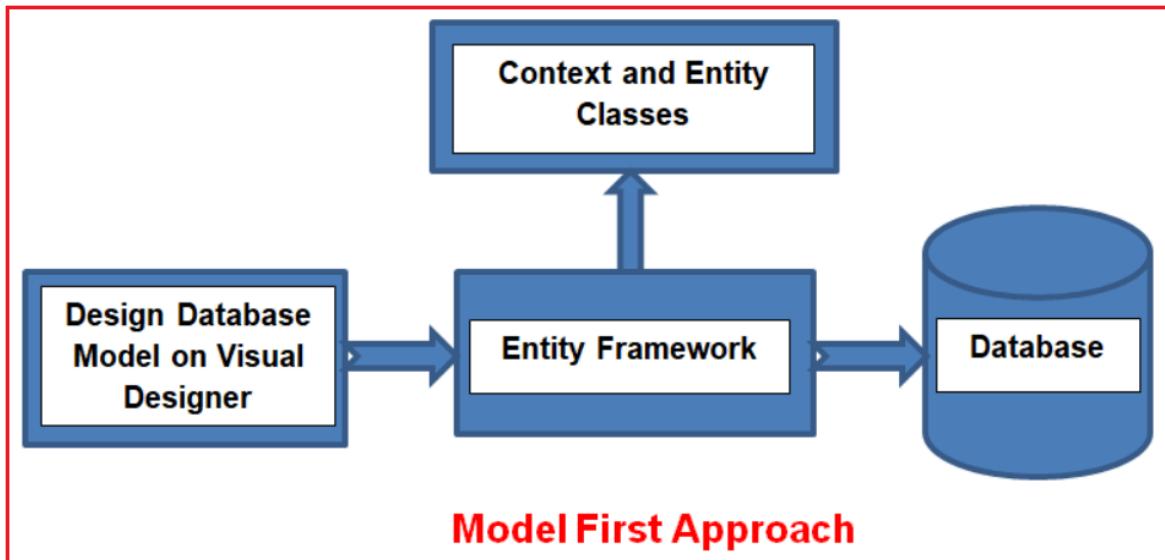
image\_222.png

Code First approach is useful:

- If there is no logic in the database
- When full control over the code, that is, there is no auto-generated model and context code
- If the database will not be changed manually

## **Model-First Approach of Entity Framework:**

This approach is very much similar to the Code First approach, but in this case, we use a visual EDMX designer to design our models. So in this approach, we create the entities, relationships, and inheritance hierarchies directly on the visual designer and then generate entities, the context class, and the database script from your visual model.



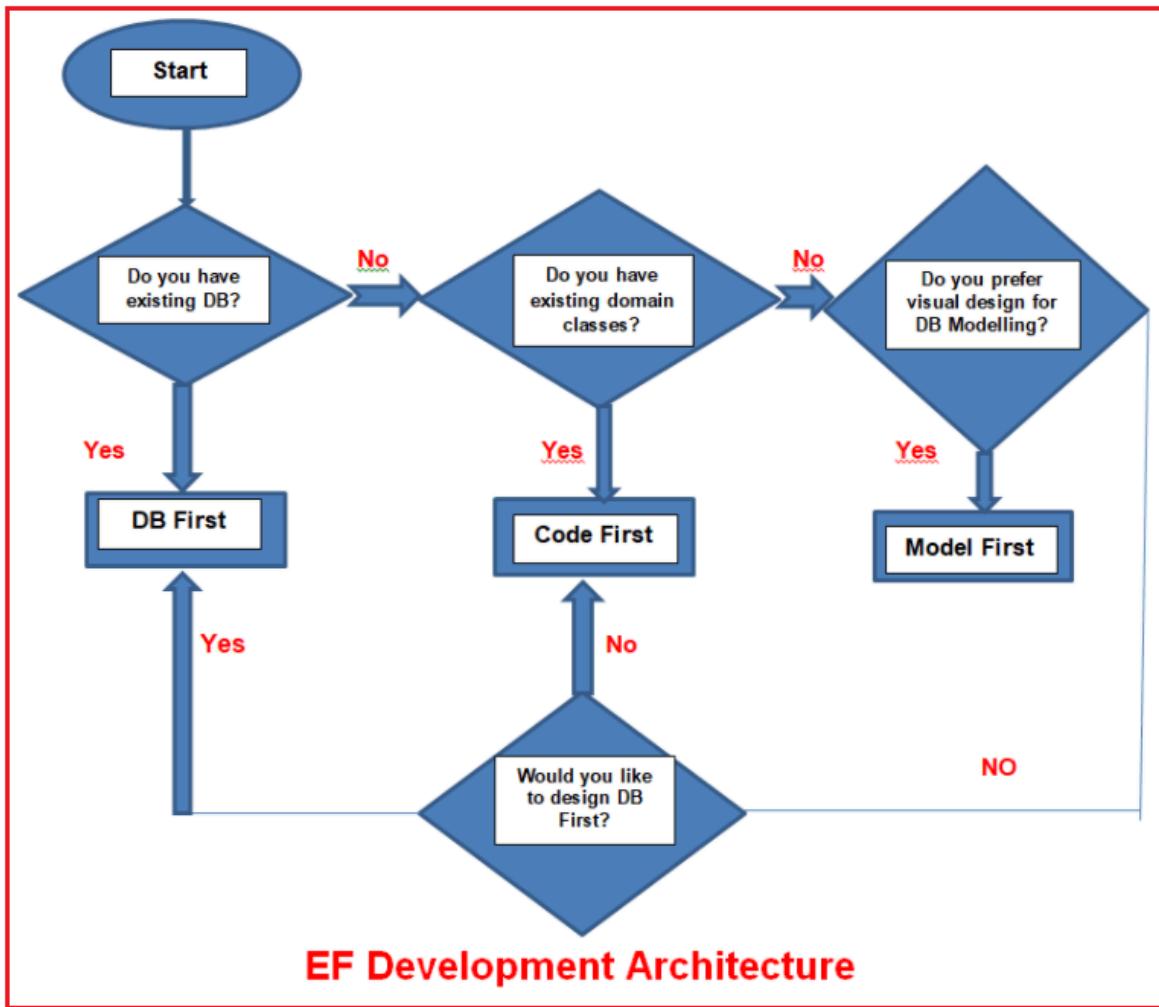
image\_223.png

- ⚠** The visual model will give us the SQL statements needed to create the database, and we can use them to create our database and connect it with our application.

The Model First approach is useful: When we really want to use the Visual Entity Designer

## Choosing the Development Approach for Your Application:

The below flowchart explains which is the right approach to develop your application using Entity Framework.

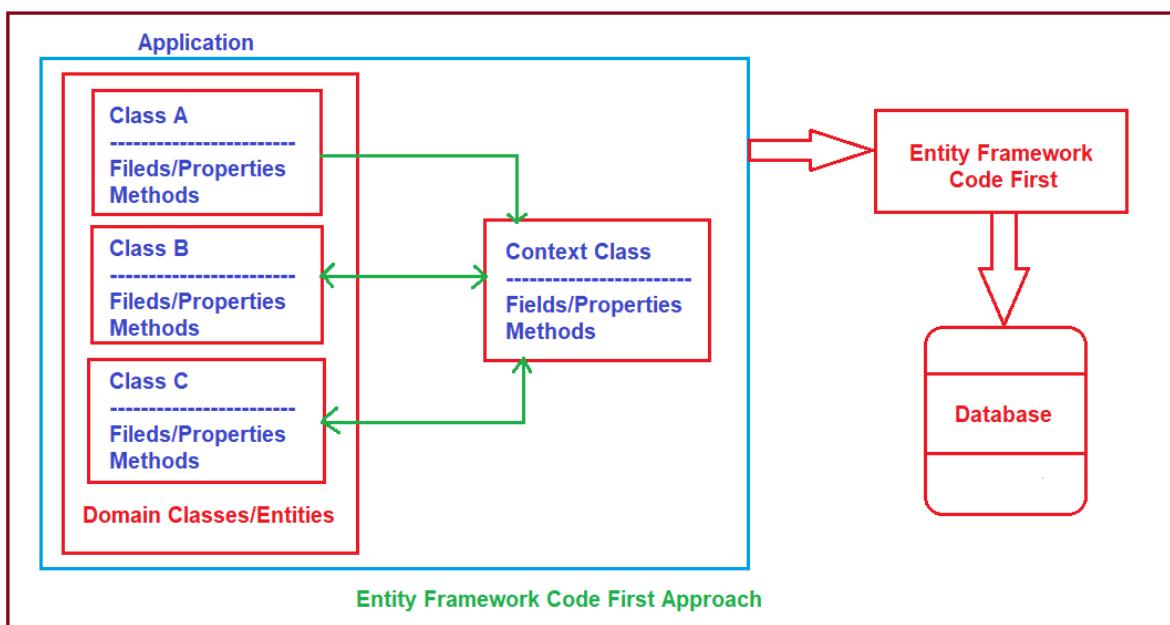


image\_224.png

As per the above diagram, if you have an existing database, then you can go with Database First approach because you can create an EDM from your existing database. But if you don't have an existing database but you have an existing application with domain classes then you can go with the code first approach because you can create a database from your existing classes. But if you don't have either an existing database or domain model classes then you can go with the model-first approach.

# Entity Framework Code First Approach

As already discussed, we need to use the Entity Framework Code First Approach when we do not have an existing database for our application. In this approach, we start developing our domain entities (domain classes) and context class first rather than designing the database first, and then based on the domain classes and context class, the entity framework will create the database. The following image shows the working style of Entity Framework Code First Approach. As you can see in the below image, the Entity Framework creates the database based on the domain classes and the context class.



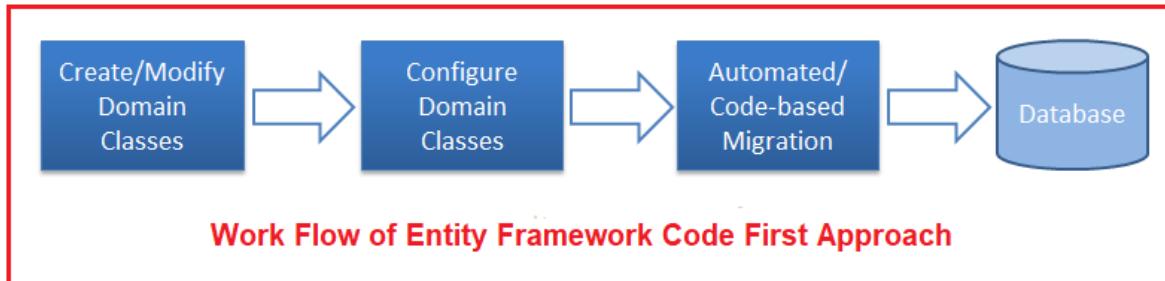
image\_225.png

This approach is best suited for applications that are highly domain-centric and will have the domain model classes created first. The database here is needed only as a persistence mechanism for these domain models.

## Entity Framework Code-First Approach Workflow

The following diagram shows the workflow of the Entity Framework Code-First Approach. First, Create or modify domain classes, then configure these domain classes

using Fluent-API or data annotation attributes, then create or update the database schema using automated migration or code-based migration



image\_226.png

# Default Code-First Conventions in Entity Framework

In Entity Framework Code-First Approach, the conventions are nothing but a set of default rules which automatically configure a Conceptual Model based on your domain classes

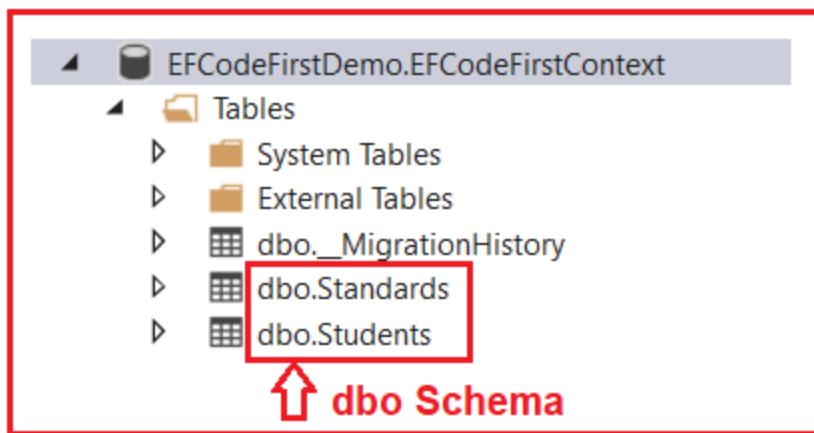
And, this is possible because of the Entity Framework Code-First Conventions. That means the database schema will be configured based on the conventions by Entity Framework API Automatically. It is also possible to change these default conventions which are followed by Entity Framework.

## What are the Default Entity Framework Code-First Conventions:

Let us understand the Default Entity Framework Code-First Conventions. They are as follows:

### Schema:

By default, the Entity Framework API creates all the database objects (tables, stored procedures, etc) with the dbo schema. If you verify that the two database tables are created with the dbo schema as shown in the below image.



image\_227.png

### Table Name:

The Entity Framework API will create the Database table with the entity class name suffixed by s i.e. <Entity Class Name> + 's'. For example, the \*\*Student\*\* domain class (entity) would map to the Students database table and the Standard domain class would map to the Standards database table and you can verify the same in the database as shown in the below image

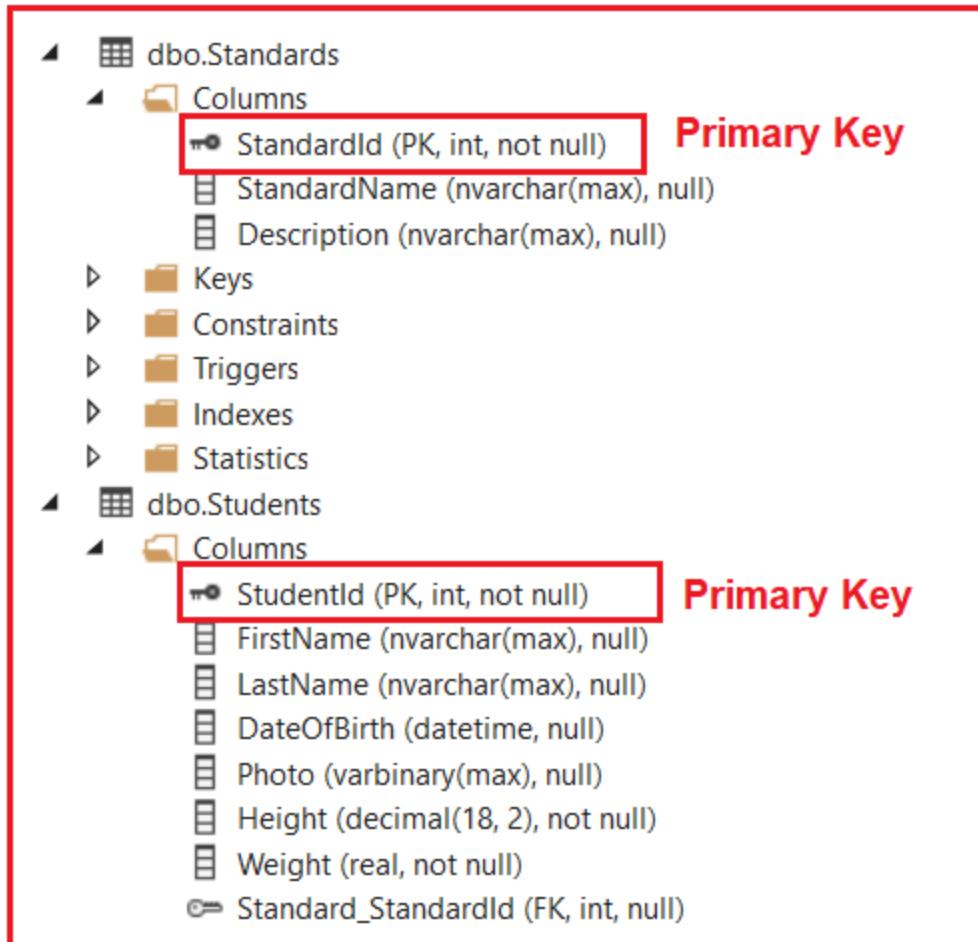
```
public class Standard
{
 public int StandardId { get; set; }
 public string StandardName { get; set; }
 public string Description { get; set; }
 public ICollection<Student> Students { get; set; }
}

public class Student
{
 public int StudentId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public DateTime? DateOfBirth { get; set; }
 public byte[] Photo { get; set; }
 public decimal Height { get; set; }
 public float Weight { get; set; }
 public virtual Standard Standard { get; set; }
}
```

image\_228.png

## Primary Key Name:

The Entity Framework API will create a primary key column for the property named Id or <Entity Class Name> + “Id” (case insensitive). For example, we have created StudentId and StandardId properties in the Student and Standard domain classes. So, these two properties will be created as Primary Key columns in the corresponding database tables as shown in the below image.

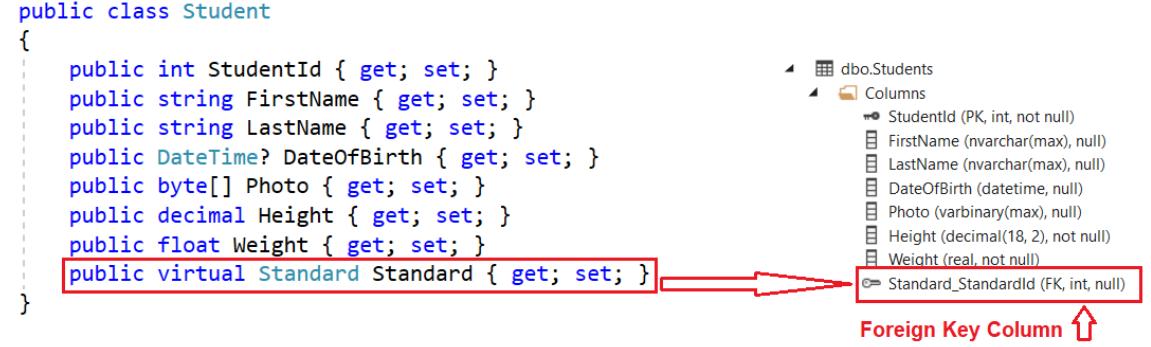


image\_229.png

**⚠** If both Id and <Entity Class Name> + “Id” properties are present in a single model class then it will create the Primary key based on the Id column only

### Foreign Key Column Name:

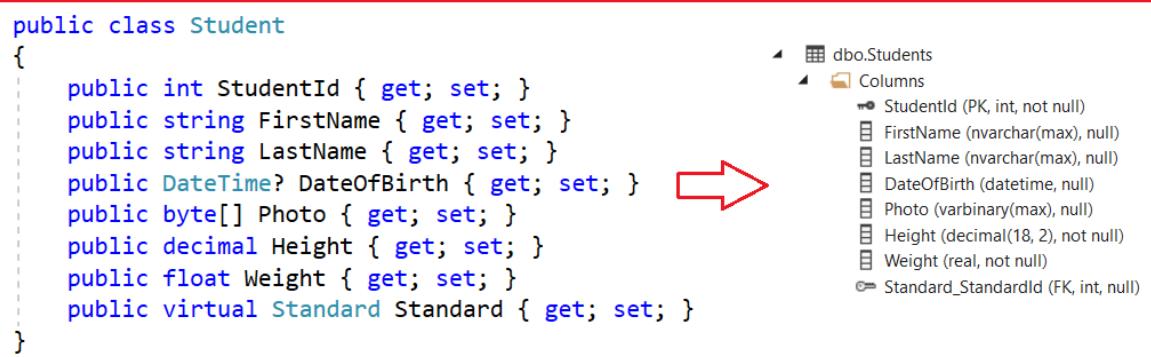
By default, the Entity Framework will look for the Foreign Key Property with the same name as the Principal Entity Primary key name in the Dependent Entity. If the foreign key property **does not exist** in the Dependent Entity class, then Entity Framework will create a Foreign Key column in the Database table with <Dependent Navigation Property Name> + “\_” + <Principal Entity Primary Key Property Name>. For example, as the Student Entity has the Standard Dependent Navigation Property, so the Entity Framework API will create Standard\_StandardId as the foreign key column in the Students table as the Student entity does not contain a foreign key property for Standard as shown in the below image.



image\_230.png

## Null and Not Null Columns:

By default, the Entity Framework API will create a null column for all reference type properties and nullable primitive properties, for example, string, Nullable<int>, Student, and Standard (all class type properties). And, the Entity Framework will create Not Null columns for Primary Key properties and non-nullable value type properties, for example, int, float, decimal, DateTime, etc.



image\_231.png

## DB Columns Order:

The Entity Framework API will create the Database table columns in the same order as the properties are added in the entity class. However, the primary key columns would be moved to the first position in the table.

## Properties Mapping to DB:

By default, all properties will map to the database table columns. If you do not want to map any property, then you need to use the [NotMapped] attribute to exclude the

property from column mapping

C# Data Type	Mapping to SQL Server Data Type
int	int
string	nvarchar(Max)
decimal	decimal(18,2)
float	real
byte[]	varbinary(Max)
datetime	datetime
bool	bit
byte	tinyint
short	smallint
long	bigint
double	float
char	No mapping
sbyte	No mapping
object	No mapping

image\_232.png

# Configure Domain Classes in Entity Framework

We have already discussed the default Entity Framework Code-First Conventions in the previous article. The EF Code-First builds the conceptual model from your domain classes using the default conventions. The Entity Framework Code-First follows a programming pattern referred to as Convention Over Configuration. However, it is also possible to override these default conventions and, in this case, we need to provide the required configuration information to the Entity Framework API. And, you can configure your domain classes or you can provide the configuration information to the Entity Framework in two different ways. They are as follows:

- Data Annotation Attributes
- Fluent API

## Data Annotations Attributes in Entity Framework Code First Approach:

Data Annotations are nothing but Attribute Based Configurations, which we can apply to our domain classes and their properties. The point that you need to remember is that these attributes are not only for Entity Framework but also used in ASP.NET MVC or ASP.NET Web API, etc. In .NET Framework, the **Data Annotations Attributes** are included in separate namespaces called `System.ComponentModel.DataAnnotations` and `System.ComponentModel.DataAnnotations.Schema`.

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace EFCodeFirstDemo
{
 [Table("StudentInfo")]
 public class Student
 {
```

```

[Key]
public int StudentId { get; set; }

[Column("FName", TypeName = "ntext")]
[MaxLength(50)]
public string FirstName { get; set; }

[Column("LName", TypeName = "ntext")]
[MaxLength(50)]
public string LastName { get; set; }

[Column("DOB", TypeName = "DateTime2")]
public DateTime DateOfBirth { get; set; }

[NotMapped]
public int Age { get; set; }

[Required]
public string Branch { get; set; }

[Index]
public int RegistrationNumber { get; set; }

public int StandardId { get; set; }

[ForeignKey("StandardId")]
public virtual Standard Standard { get; set; }

}

}

```

**A** The Problem with Data Annotation Attribute is that it does not support all the configuration options that are required for Entity Framework Code First Approach. So, in that case, we can go for **Fluent API**, which provides all the configuration options required for Entity Framework.

## Fluent API in Entity Framework Code First Approach:

Another approach to configuring the domain classes in Entity Framework Code First Approach is by using Fluent API. Entity Framework Fluent API is based on a Fluent API design pattern (Fluent Interface) where the result is formulated by method chaining. So, before understanding Entity Framework Fluent API, *first we need to understand Fluent Interface Design Pattern.*



The Fluent Interfaces and Method chaining are related to each other. Or we can say that one is a concept and the other one is its implementation.

### What is the Fluent Interface Design Pattern?

The main objective of the Fluent Interface Design Pattern is that we can apply multiple properties (or methods) to an object by connecting them with dots (.) without having to re-specify the object name each time. Something like the following.

```
FluentEmployee obj = new FluentEmployee();

obj.NameOfTheEmployee("Anurag Mohanty")
 .Born("10/10/1992")
 .WorkingOn("IT")
 .StaysAt("Mumbai-India");
```

image\_233.png

### What is Method Chaining?

Method Chaining is a common technique where each method returns an object and all these methods can be chained together to form a single statement. For a better understanding, please create a new console application and copy and paste the following code.

```
using System;
namespace FluentInterfaceDesignPattern
{
 class Program
```

```

{
 static void Main(string[] args)
 {
 FluentEmployee obj = new FluentEmployee();

 obj.NameOfTheEmployee("Anurag Mohanty")
 .Born("10/10/1992")
 .WorkingOn("IT")
 .StaysAt("Mumbai-India");

 Console.Read();
 }
}

public class Employee
{
 public string FullName { get; set; }
 public DateTime DateOfBirth { get; set; }
 public string Department { get; set; }
 public string Address { get; set; }
}

public class FluentEmployee
{
 private Employee employee = new Employee();

 public FluentEmployee NameOfTheEmployee(string FullName)
 {
 employee.FullName = FullName;
 return this;
 }

 public FluentEmployee Born(string DateOfBirth)
 {
 employee.DateOfBirth = Convert.ToDateTime(DateOfBirth);
 return this;
 }
}

```

```
public FluentEmployee WorkingOn(string Department)
{
 employee.Department = Department;
 return this;
}

public FluentEmployee StaysAt(string Address)
{
 employee.Address = Address;
 return this;
}
}
```

# Data Annotation Attributes in Entity Framework Code-First

Data Annotations are nothing but .NET Attributes or .NET Classes that can be applied to our domain classes and their properties to override the default conventions in Entity Framework.

**⚠** Data Annotation Attributes give you only a subset of configuration options in Entity Framework Code-First. For the full set of configuration options in Code-First, you need to use Fluent API.

## **System.ComponentModel.DataAnnotations Attributes:**

The following Data Annotation Attributes impact the nullability or size of the column in a database and these Attributes belong to `System.ComponentModel.DataAnnotations` namespace.

- 1. Key:** The Key Attribute can be applied to a property of a domain entity to specify a key property and make the corresponding database column a PrimaryKey column. That means it denotes one or more properties that uniquely identify an entity.
- 2. Timestamp:** The Timestamp Attribute can be applied to a property of a domain entity to specify the data type of the corresponding database column as a row version.
- 3. ConcurrencyCheck:** The ConcurrencyCheck Attribute can be applied to a property of a domain entity to specify that the corresponding column should be included in the optimistic concurrency check.
- 4. Required:** The Required Attribute can be applied to a property of a domain entity to specify that the corresponding database column is a NotNull column. That means it specifies that a data field value is required.
- 5. MinLength:** The MinLength Attribute can be applied to a property of a domain entity to specify the minimum string length allowed in the corresponding database column. That means it specifies the minimum length of array or string data allowed in a property.

6. **MaxLength:** The MaxLength Attribute can be applied to a property of a domain entity to specify the maximum string length allowed in the corresponding database column. That means it specifies the maximum length of array or string data allowed in a property.
7. **StringLength:** The StringLength Attribute can be applied to a property of a domain entity to specify the minimum and maximum string length allowed in the corresponding database column. That means it specifies the minimum and maximum length of characters that are allowed in a data field.

## **System.ComponentModel.DataAnnotations.Schema Attributes**

The following Data Annotation Attributes impact the Schema of a database and these Attributes belong to `System.ComponentModel.DataAnnotations.Schema` namespace.

1. **Table:** The Table Attribute can be applied to a domain entity to configure the corresponding table name and schema in the database. That means it specifies the database table that a class is mapped to. It has two properties i.e. Name and Schema which are used to specify the corresponding database table name and schema.
2. **Column:** The Column Attribute can be applied to a property of a domain entity to configure the corresponding database column name, order, and data type. That means it represents the database column that a property is mapped to. It has three properties i.e. Name, Order, and TypeName to specify the column name, order, and data type in the database.
3. **Index:** The Index Attribute can be applied to a property of a domain entity to configure that the corresponding database column should have an Index in the database. That means when this attribute is placed on a property it indicates that the database column to which the property is mapped has an index. This attribute is used by Entity Framework Migrations to create indexes on mapped database columns. Multi-column indexes are created by using the same index name in multiple attributes. The information in these attributes is then merged together to specify the actual database index. It is available from Entity Framework 6.1.
4. **ForeignKey:** The ForeignKey Attribute can be applied to a property of a domain entity to mark it as a foreign key column in the database. That means it denotes a property

used as a foreign key in a relationship.

5. **NotMapped:** The NotMapped Attribute can be applied to a property or entity class that should be excluded from the model and should not generate a corresponding column or table in the database. That means it denotes that a property or class should be excluded from database mapping.
6. **InverseProperty:** The InverseProperty Attribute can be applied to a property to specify the inverse of a navigation property that represents the other end of the same relationship.

# Table Data Annotation Attribute in Entity Framework

The Table Data Annotation Attribute in Entity Framework Code First Approach can be applied to a domain class to configure the corresponding database table name and schema. It overrides the default convention in Entity Framework. As per the default conventions, Entity Framework API will create a database table whose name is matching with + 's' (or 'es') in a context class.

All the Data Annotation Attributes are classes that are inherited from the Attribute abstract class. Now, if you go to the definition of Table Attribute class, then you will see the following.

```
public class TableAttribute : Attribute
{
 ...
 public TableAttribute(string name);

 ...
 public string Name { get; }
 ...
 public string Schema { get; set; }
}
```

image\_234.png

As you can see in the above TableAttribute class, it is having two properties and one constructor. The constructor is taking one string parameter which is nothing but the database table name and this is mandatory. The Schema property is optional and the use of the Name and Schema properties are as follows:

- Name: Gets the name of the table the class is mapped to. It is a read-only property.
- Schema: Gets or sets the schema of the table the class is mapped to. This is optional. It is a read-write property.



Using square bracket [], we need to specify the attributes in .NET.

Syntax to use **Table Attribute**: [Table(string name, Properties:[Schema = string]) Example to use **Table Attribute**: [Table("StudentInfo", Schema="Admin")]

## Examples to understand Table Data Annotation Attribute in Entity Framework:

Let us understand Table Data Annotation Attribute in Entity Framework Code First Approach with an example. Let us modify the Student Entity class as follows. As you can see, we have specified the table name as StudentInfo.

```
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 [Table("StudentInfo")]
 public class Student
 {
 public int StudentId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
 }
}
```

In the above example, the Table Attribute is applied to the Student Entity class. So, the Entity Framework will override the default conventions and create the StudentInfo database table instead of the Students table in the database which is going to be mapped with the above Student Entity class.

# Column Data Annotation Attribute in Entity Framework

The Column Data Annotation Attribute can be applied to one or more properties of a domain entity to configure the corresponding database column name, column order, and column data type. That means it represents the database column that a property is mapped to.

Now, if you go to the definition of Column Attribute, then you will see the following.

```
namespace System.ComponentModel.DataAnnotations.Schema
{
 ...
 public class ColumnAttribute : Attribute
 {
 ...
 public ColumnAttribute();
 public ColumnAttribute(string name);

 ...
 public string Name { get; }
 public int Order { get; set; }
 public string TypeName { get; set; }
 }
}
```

image\_235.png

As you can see in the above image, this class has two constructors. The 0-Argument constructor will create the database column with the same name as the property name while the other overloaded constructor which takes the string name as a parameter will create the database table column with the passed name. Again, this Column attribute has three properties. The meaning of the properties are as follows:

- Name: Gets the name of the column the property is mapped to. This is a read-only property.
- Order: Gets or sets the zero-based order of the column the property is mapped to. It returns the order of the column. This is a read-write property.

- **TypeName:** Gets or sets the database provider-specific data type of the column the property is mapped to. It returns the database provider-specific data type of the column the property is mapped to. This is a read-write property.

Syntax to use Column Attribute: [Column (string name, Properties: [Order = int], [TypeName = string])] Example to use Column Attribute: [Column("DOB", Order = 2, TypeName = "DateTime2")]

## Examples to understand Column Data Annotation Attribute in Entity Framework:

With the FirstName property, we have not provided the string name i.e. using the 0-Argument constructor, so in this case, it will create the database table column with the same name as the Property name. With the LastName property, we have specified the name as LName i.e. using the constructor which takes one string parameter, so in this case, it will create the database table column with the name LName which is mapped to the LastName property. The Column attribute belongs to System.ComponentModel.DataAnnotations.Schema namespace.

```
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 public class Student
 {
 public int StudentId { get; set; }
 [Column]
 public string FirstName { get; set; }
 [Column("LName")]
 public string LastName { get; set; }
 }
}
```

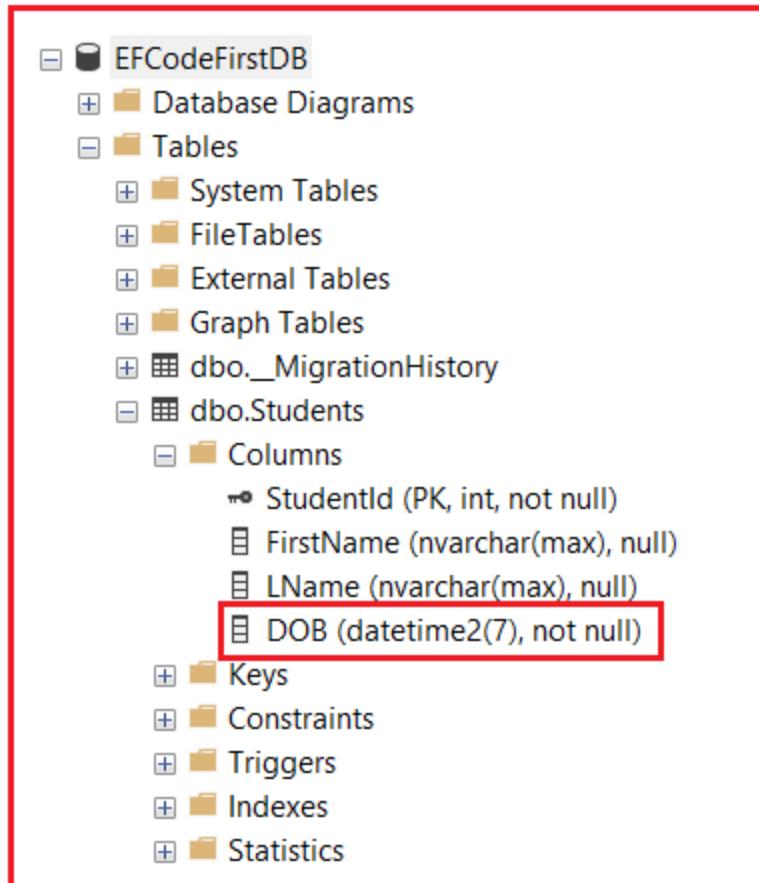
## Column Data Type in Entity Framework:

As we see, the Column Attribute class is having a property called TypeName, and that TypeName property is used to get or set the data type of a database column. That means this property gets or sets the database provider-specific data type of the column the

property is mapped to. For a better understanding, please modify the Student Entity as follows. Here, you can see, we have set the DateOfBirth column name as DOB and Data type as DateTime2 using TypeName Property.

```
using System;
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 public class Student
 {
 public int StudentId { get; set; }
 [Column]
 public string FirstName { get; set; }
 [Column("LName")]
 public string LastName { get; set; }
 [Column("DOB", TypeName = "DateTime2")]
 public DateTime DateOfBirth { get; set; }
 }
}
```

With the above changes in place, now run the application code. It should create a column with the name as DOB with the data type as DateTime2 instead of DateTime. You can verify the same in the database as shown in the below image.



image\_236.png

## Column Order in Entity Framework Code First Approach:

Another property called Order is provided by the Column Attribute class which is used to set or get the order of the columns. It is 0 Based orders i.e. it is going to start from 0. As per the default convention, the Primary Key columns will come first, and then the rest of the columns based on the order that we specified in the Column Attribute Order Property.

```
using System;
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 public class Student
 {
 //Primary Key: Order Must be 0
 [Column(Order = 0)]
```

```
public int StudentId { get; set; }

[Column(Order = 2)]
public string FirstName { get; set; }

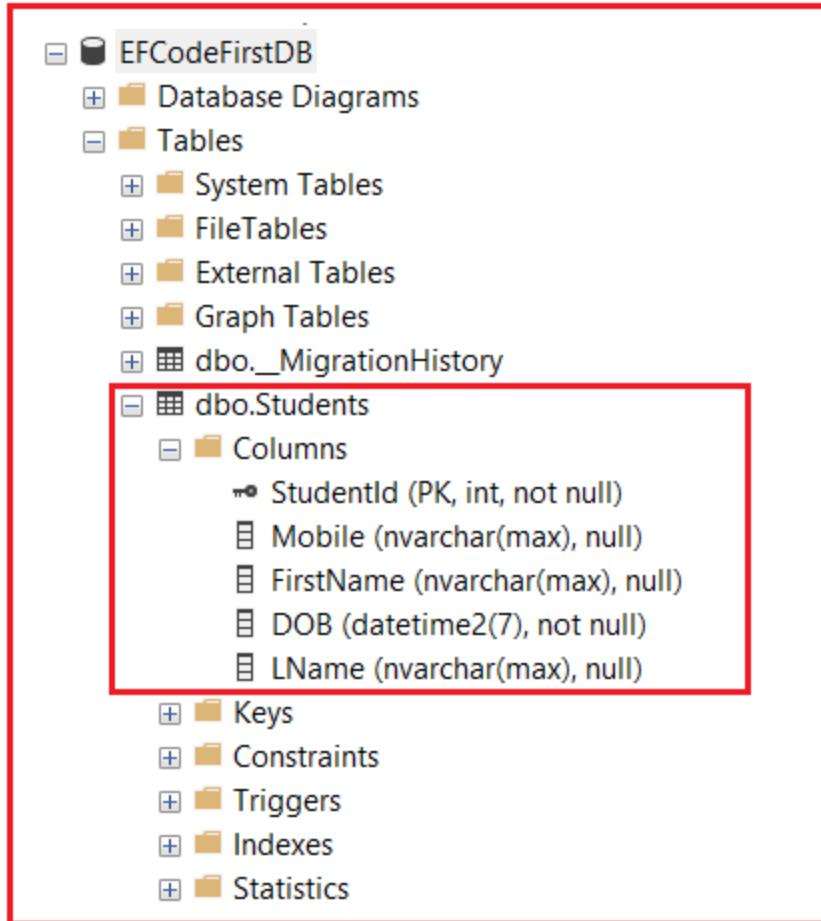
[Column("LName", Order =4)]
public string LastName { get; set; }

[Column("DOB", Order =3, TypeName = "DateTime2")]
public DateTime DateOfBirth { get; set; }

[Column(Order = 1)]
public string Mobile { get; set; }

}
```

Now, run the application and verify the database table columns order and it should be in proper order as per the Order parameter as shown in the below image.



image\_237.png

# Key Attribute in Entity Framework

Before understanding Key Attributes, let us first understand what is a Primary Key in a database.

## What is Primary Key in the Database?

The Primary Key is the combination of Unique and Not Null Constraints. That means it will not allow either NULL or Duplicate values into a column or columns on which the primary key constraint is applied. Using the primary key, we can enforce entity integrity i.e. using the primary key value we should uniquely identify a record.

- ⚠ A table should contain only 1 Primary Key which can be either on single or multiple columns i.e. the composite primary key. A table should have a primary key to uniquely identify each record.

## Key Attribute in Entity Framework

As per the default convention, the Entity Framework will create the primary key column for the property whose name is Id or + “Id” (case insensitive). For example, let us modify the Student entity class as follows. Here, we have created the StudentId property in the Student class. So, this property will be created as a Primary Key column in the corresponding database table.

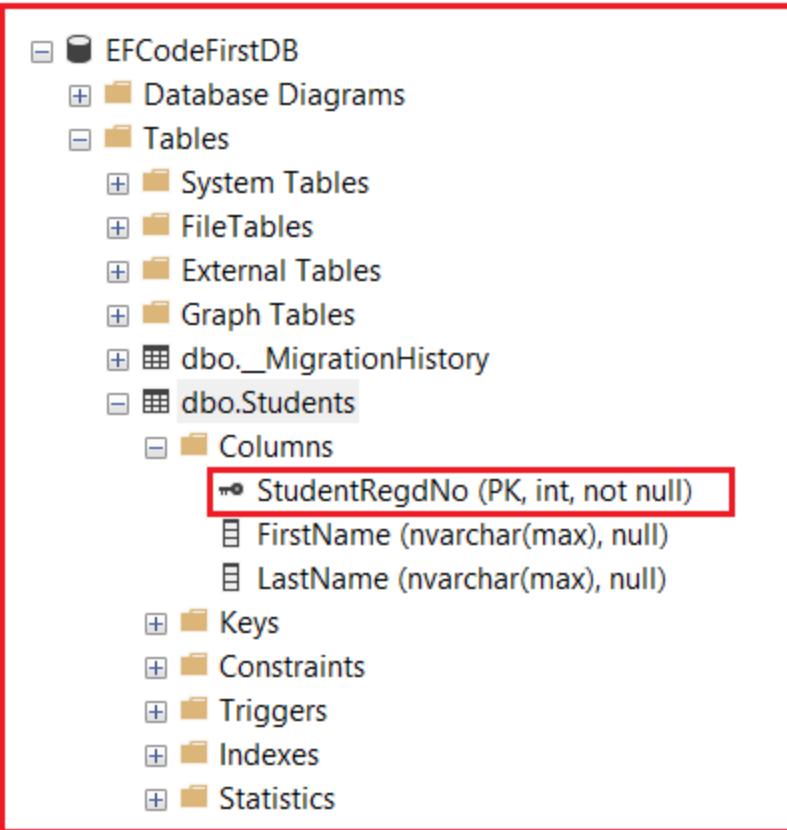
```
using System;
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 public class Student
 {
 public int StudentId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
 }
}
```

Now, we do not want to make StudentId as the Primary key column, instead, we want a different column let us say StudentRegdNo as the primary key.

So, modify the Student class as follows. As you can see in the below code, we just decorate the StudentRegdNo property with the Key Attribute.

```
using System.ComponentModel.DataAnnotations;
namespace EFCodeFirstDemo
{
 public class Student
 {
 [Key]
 public int StudentRegdNo { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }
 }
}
```

With the above changes in place, now run the application code again, and this time you will not get any exceptions and the database should be updated as expected. You can also verify the database as shown in the below image.



image\_238.png

# ForeignKey Attribute in Entity Framework

One of the most important concepts in a database is creating the relationship between the database tables. This relationship provides a mechanism for linking the data stores in multiple database tables and retrieving them in an efficient manner.

In order to create a link between two database tables, we must specify a Foreign Key in one table that references a unique column (Primary Key or Foreign Key) in another table. That means the Foreign Key constraint is used for binding two database tables with each other and then verifying the existence of one table's data in other tables. A foreign key in one TABLE points to a primary key or unique key in another table.

## ForeignKey Attribute in Entity Framework:

The ForeignKey Attribute in Entity Framework is used to configure a Foreign Key Relationship between the two entities. It overrides the default Foreign Key convention which is followed by Entity Framework. As per the default convention, the Entity Framework API will look for the Foreign Key Property with the same name as the Principal Entity Primary Key Property name in the Dependent Entity.

A Relationship in Entity Framework always has two endpoints. Each endpoint must return a navigational property that maps to the other end of the relationship. Let us understand this with an example. The following Standard Entity is going to be our Principal Entity and StandardId is the Primary Key Property. Here, you can see, we have one collection navigational property i.e. students and this is mandatory in order to implement Foreign Key Relationships using Entity Framework Code First Approach.

```
using System;
using System.Collections.Generic;
namespace EFCodeFirstDemo
{
 public class Standard
 {
 public int StandardId { get; set; }
 public string StandardName { get; set; }
 public string Description { get; set; }
```

```
 public ICollection<Student> Students { get; set; }
}
}
```

Now, modify the Student Entity as follows. Here, you can see, we have added the StandardId property whose name is the same as the Primary Key Property of the Standard table. We have also added the Standard Reference Navigational Property and this is mandatory.

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 public class Student
 {
 public int StudentId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }

 //The Following Property Exists in the Standard Entity
 public int StandardId { get; set; }

 //To Create a Foreign Key it should have the Standard
 //Navigational Property
 public Standard Standard { get; set; }
 }
}
```

- A** The point that you need to remember, both Entities should and must have Navigational Properties to implement Foreign Key Relationships. In our example, Student Entity has the Standard Reference Navigational Property, and Standard Entity has the Students collection Navigation Property which will establish the one-to-many relationship between these two entities. That means one student has one standard and one standard can have multiple students.



# Index Attribute in Entity Framework

Entity Framework 6 provides the [Index] attribute to create an index on a particular column in the database. If you are using an earlier version of Entity Framework, then the Index Attribute will not work. It is also possible to create an index on one or more columns using the Index Attribute. Adding the Index Attribute to one or more properties of an Entity will cause Entity Framework to create the corresponding index in the database when it creates the database.

Before creating an Index using Entity Framework Code First Approach, let us first understand some basic concepts of an Index like what is Index, why we need an Index, what different types of Indexes and how the index impact DML Operations.

## What is an Index and why Index in a Database?

Indexes in SQL Server are nothing but database object in a database which is used to improve the performance of search operations. When we create an index on any column or columns of a table, the SQL Server Database internally maintains a separate table called the Index Table. And when we are trying to retrieve the data from the original table, depending on the index table, SQL Server directly goes to the original table and retrieves the data very quickly.

## When SQL Server uses Indexes?

The SQL Server Database uses indexes of a table provided that the select or update or delete statement contained the “WHERE” clause and moreover the where clause condition column must be an indexed column. If the select statement contains an “ORDER BY” clause then also the indexes can be used.

## Types of indexes in SQL Server

SQL Server Indexes are divided into two types. They are as follows:

1. **Clustered Index:** The Clustered Index in SQL Server defines the order in which the data is physically stored in a table.

2. Non-Clustered Index: In SQL Server Non-Clustered Index, the arrangement of data in the index table will be different from the arrangement of data in the actual table.

**⚠** When we create an Index by using the Unique option then it is called Unique Index. Then the column(s) on which the unique index is created will not allow duplicate values

## How does Index Affect the DML Operations?

The most important point that you need to keep in mind is that indexes make the retrieval of data faster and more efficient, in most cases. However, creating a lot of Indexes in a table or view could affect the performance of other operations such as inserts, delete, or updates.

Let us modify the Student Entity Class as follows to use the Index Attribute. Here, you can see, we have applied the Index Attribute on the RegistrationNumber property.

```
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 public class Student
 {
 public int StudentId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }

 [Index]
 public int RegistrationNumber { get; set; }
 }
}
```

Now, if you want to give a different name to your Index name rather than the auto-generated index name, then you need to use the other overloaded version of the Constructor which takes the name parameter.

```
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
```

```

{
 public class Student
 {
 public int StudentId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }

 [Index("Index_RegistrationNumber")]
 public int RegistrationNumber { get; set; }
 }
}

```

## Creating Index on Multiple Columns using Entity Framework Code First Approach

It is also possible to create an Index on Multiple Columns. For this, we need to use the overloaded version which takes the index name and order parameter. In this case, we need to specify the same name for both the Index Attribute. Let us understand this with an example. Please modify the Student class as follows. As you can see, here, we have applied the Index Attribute with the same name with RegistrationNumber and RollNumber properties and with the order values 2 and 1 respectively. In this case, the Entity Framework API will create one index based on the RegistrationNumber and RollNumber columns.

```

using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 public class Student
 {
 public int StudentId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }

 [Index("Index_Roll_Registration", 2)]
 public int RegistrationNumber { get; set; }

 [Index("Index_Roll_Registration", 1)]
 public int RollNumber { get; set; }
 }
}

```

```
 }
}
```

## How to Create Clustered and Unique Index using Entity Framework?

By default, *Entity Framework API creates Non-Clustered and Non-Unique Index*. If you want to create a Clustered and Unique Index, then you need to use the following two properties and you need to set the values to True.

- **IsClustered**: Set this property to true to define a clustered index. Set this property to false to define a non-clustered index.
- **IsUnicode**: Set this property to true to define a unique index. Set this property to false to define a non-unique index.

Now, modify the Student class as shown below to create a clustered and unique index on the RegistrationNumber property.

```
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCodeFirstDemo
{
 public class Student
 {
 public int StudentId { get; set; }
 public string FirstName { get; set; }
 public string LastName { get; set; }

 [Index("Index_RegistrationNumber", IsClustered =true, IsUnique
=true)]
 public int RegistrationNumber { get; set; }
 }
}
```

# Relationships in EF

In EF Core you declare entity types — your domain classes — and relationships between them using navigation properties and optionally configuration (via attributes or Fluent API). The main kinds of relationships supported are:

- One-to-One (1:1) — one entity is associated with exactly one of another.
- One-to-Many / Many-to-One (1:n / n:1) — one entity has many related entities, each of which refers to one parent.
- Many-to-Many (n:n) — many entities on one side relate to many on the other. EF Core supports this — either with an explicit join entity or via “skip-navigation” so that EF manages the join table under the hood.

## Blogging API Entity Design

This document describes a recommended data model for a blogging API using TypeORM. It includes entity definitions and relationship mappings for the following domain objects:

- Users
- Profiles
- Authors
- Blogs
- Comments
- Categories

The design includes one-to-one (1-1), one-to-many (1-n), and many-to-many (n-n) relationships and example TypeORM decorator usage for each entity.

### High-level relationship summary

- User - Profile: One-to-One (1-1) : Each User has exactly one Profile that contains profile metadata.
- User - Author: One-to-One (1-1) : A User can be an Author. Author contains publishing-related data.
- Author - Blog: One-to-Many (1-n) : An Author can write many Blog posts.
- Blog - Comment: One-to-Many (1-n) : A Blog can have many Comments.
- Blog - Category: Many-to-Many (n-n) : Blogs can belong to multiple Category entries and vice-versa.
- User - Comment: One-to-Many (1-n) : A User can write many Comments.

## Translating Your Blog Model to EF Core (Code-First with .NET 10)

Create a new .NET 10 console app:

```
dotnet new console -n BloggingApp
cd BloggingApp
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer # or Sqlite
etc.
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet add package Microsoft.EntityFrameworkCore.Design # Migrations
```

Then install the EF CLI tools (if not already):

```
dotnet tool install --global dotnet-ef
```

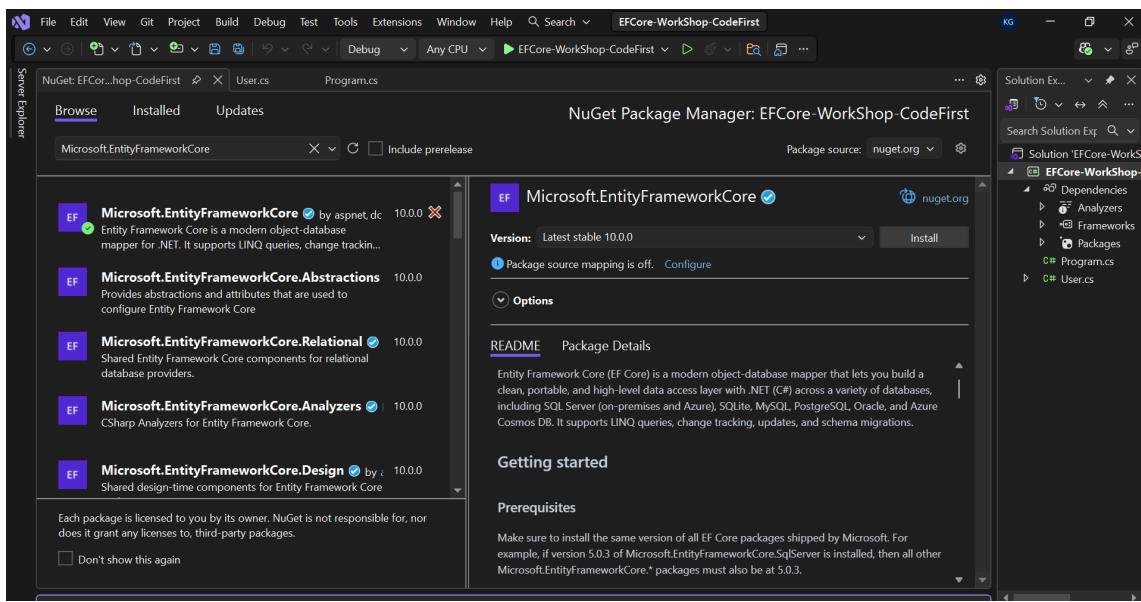
You can check installation with:

```
dotnet ef --version
```:contentReference[oaicite:3]{index=3}
```

Ensure your project references include a `DbContext` (we'll add below), so the CLI tools know where to operate.

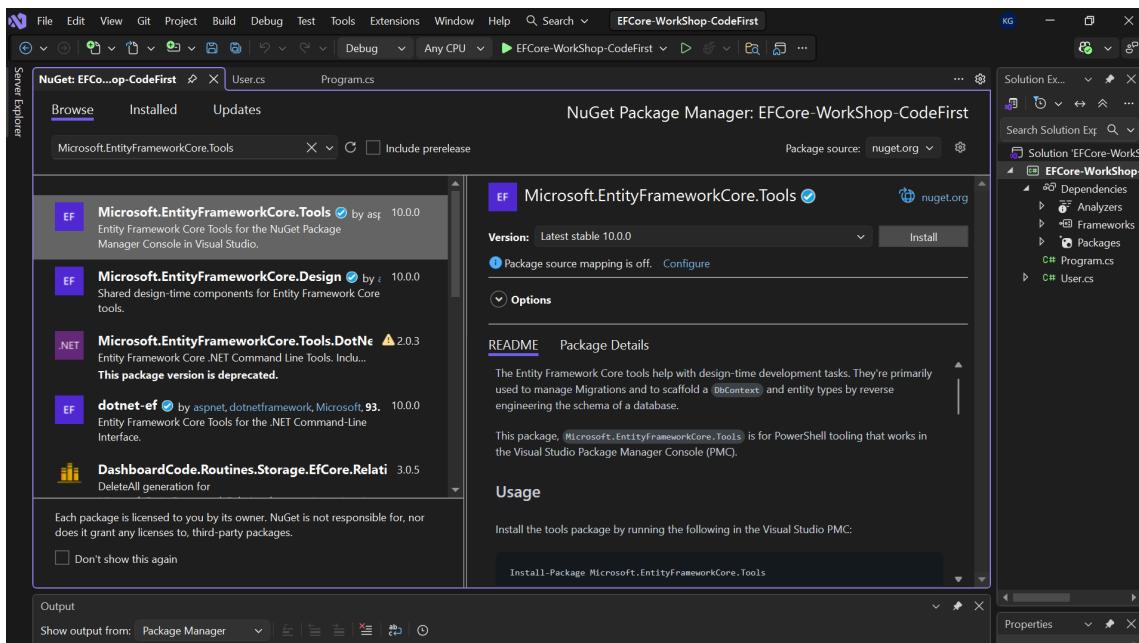
📂 Project Structure (Suggested)

- Microsoft.EntityFrameworkCore



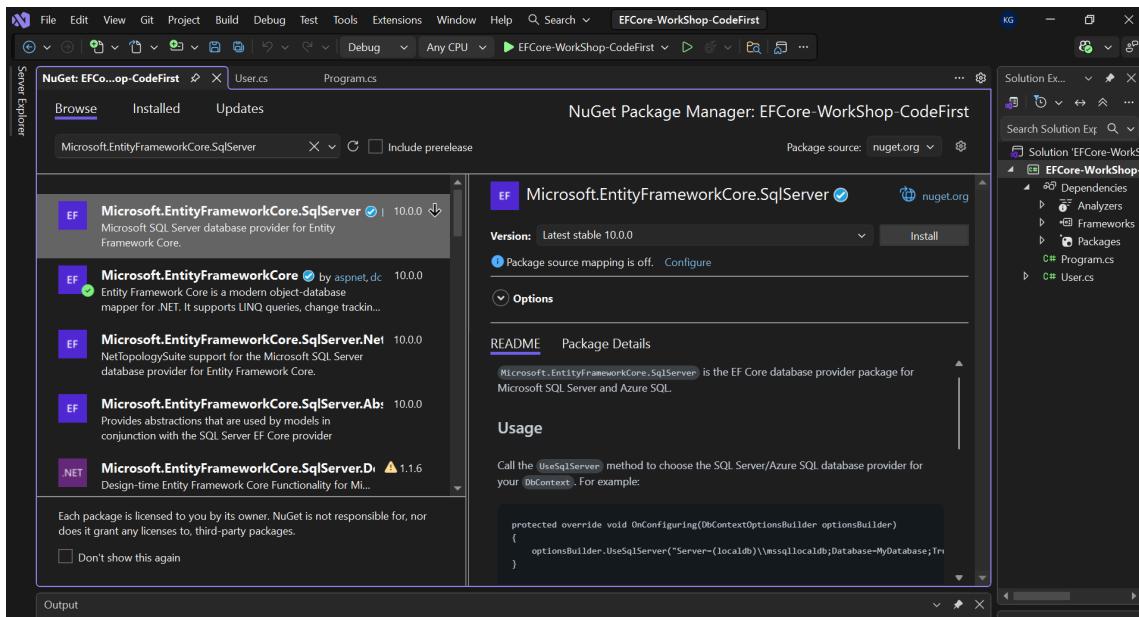
image_239.png

- Microsoft.EntityFrameworkCore.Tools



image_240.png

- Microsoft.EntityFrameworkCore.SqlServer



image_241.png

Recommended Folder Structure for Your Console + EF Core App

Here's one layout that works well — starting simple, but structured enough for growth:

```

/YourSolutionRoot
|
|   -- BloggingApp.csproj           <-- your .NET project file
|   -- Program.cs                  <-- entry point (or minimal Host setup)
|
|   -- Entities/                  <-- domain entity classes (each in its
|       own file)
|       |   -- User.cs
|       |   -- Profile.cs
|       |   -- Author.cs
|       |   -- Blog.cs
|       |   -- Comment.cs
|       |   -- Category.cs
|
|   -- Data/                      <-- data-access related: DbContext,
|       migrations, config
|       |   -- BloggingContext.cs
|       |   -- (optionally) Migrations/    <-- if you plan to use EF Core
|           migrations
|
|   -- Services/ (optional)      <-- if you have business-logic or
|       services / helpers
|       |   ...
|
|   -- Infrastructure/ (optional) <-- for infra-specific code (e.g.
|       external integrations)
|       |   ...
|
|   -- README.md                 <-- description, how to run/build, etc.

```

- `Entities/` — contains plain C# classes representing domain entities (User, Blog, Comment, etc.). Each entity in its own .cs file. This helps with clarity: you immediately know where to find a given type; easier for large models.
- `Data/` — contains the EF Core DbContext class (e.g. BloggingContext.cs), plus migrations folder if you generate migrations. All DB-access and persistence-related

code lives here.

- `Program.cs` — minimal application entry point. Keep it focused (e.g. configuring `DbContext`, running seed logic, example operations) — avoid cluttering with domain or business logic.
- Optional folders — as your app grows, you can add `Services/` (for business logic, domain services), or `Infrastructure/` (for external dependencies: email providers, file storage, etc.). Even if not needed now, having them ready helps scale cleanly.
- `README.md / docs` — always good for onboarding or future reference.

Add the entity classes (`User`, `Profile`, `Author`, `Blog`, `Comment`, `Category`). They currently have simple scalar properties but no relationships / navigation properties / FK fields.

Example:

`User.cs:`

```
using System.ComponentModel.DataAnnotations;

namespace EFCore_WorkShop_CodeFirst.Entities
{
    internal class User
    {
        public Guid Id { get; set; }

        [Required]
        [MaxLength(100)]
        public string Username { get; set; } = null!;

        [Required]
        [MaxLength(200)]
        public string Email { get; set; } = null!;

        [Required]
        public string PasswordHash { get; set; } = null!;

        public bool IsActive { get; set; } = true;
    }
}
```

```
        public DateTime CreatedAt { get; set; }
        public DateTime UpdatedAt { get; set; }
        // <-- no navigation or relationships yet
    }
}
```

Profile.cs:

```
using System.ComponentModel.DataAnnotations;

namespace EFCore_WorkShop_CodeFirst.Entities
{
    internal class Profile
    {
        public Guid Id { get; set; }

        [MaxLength(200)]
        public string? FullName { get; set; }

        public string? Bio { get; set; }
        public string? AvatarUrl { get; set; }

        public DateTime CreatedAt { get; set; }
        public DateTime UpdatedAt { get; set; }
        // <-- no navigation or relationships yet
    }
}
```

Author.cs:

```
using System.ComponentModel.DataAnnotations;

namespace EFCore_WorkShop_CodeFirst.Entities
{
    internal class Author
```

```
{  
    public Guid Id { get; set; }  
  
    [MaxLength(200)]  
    public string? PenName { get; set; }  
  
    public string? Biography { get; set; }  
  
    public DateTime CreatedAt { get; set; }  
    public DateTime UpdatedAt { get; set; }  
    // <-- no navigation or relationships yet  
}  
}
```

Blog.cs:

```
using System.ComponentModel.DataAnnotations;  
  
namespace EFCore_WorkShop_CodeFirst.Entities  
{  
    internal class Blog  
    {  
        public Guid Id { get; set; }  
  
        [Required]  
        [MaxLength(200)]  
        public string Title { get; set; } = null!;  
  
        [Required]  
        public string Content { get; set; } = null!;  
  
        public bool Published { get; set; } = false;  
        public string? Excerpt { get; set; }  
  
        public DateTime CreatedAt { get; set; }  
        public DateTime UpdatedAt { get; set; }  
        // <-- no navigation or relationships yet
```

```
    }  
}
```

Comment.cs:

```
using System.ComponentModel.DataAnnotations;  
  
namespace EFCore_WorkShop_CodeFirst.Entities  
{  
    internal class Comment  
    {  
        [Key]  
        public Guid Id { get; set; }  
  
        [Required]  
        public string Content { get; set; } = null!;  
  
        public bool IsApproved { get; set; } = false;  
  
        public DateTime CreatedAt { get; set; }  
        public DateTime UpdatedAt { get; set; }  
        // <-- no navigation or relationships yet  
    }  
}
```

Category.cs:

```
using System.ComponentModel.DataAnnotations;  
  
namespace EFCore_WorkShop_CodeFirst.Entities  
{  
    internal class Category  
    {  
        public Guid Id { get; set; }  
  
        [Required]  
        [MaxLength(100)]
```

```

        public string Name { get; set; } = null!;

        public string? Description { get; set; }

        public DateTime CreatedAt { get; set; }
        public DateTime UpdatedAt { get; set; }

        public List<Blog> Blogs { get; set; } = new();
        // <-- no navigation or relationships yet
    }
}

```

Create the DbContext (e.g. BloggingContext inside Data folder) as shown below

BloggingContext.cs

```

using EFCore_WorkShop_CodeFirst.Entities;
using Microsoft.EntityFrameworkCore;

namespace EFCore_WorkShop_CodeFirst.Data
{
    internal class BloggingContext : DbContext
    {
        public DbSet<User> Users => Set<User>();
        public DbSet<Profile> Profiles => Set<Profile>();
        public DbSet<Author> Authors => Set<Author>();
        public DbSet<Blog> Blogs => Set<Blog>();
        public DbSet<Comment> Comments => Set<Comment>();
        public DbSet<Category> Categories => Set<Category>();

        protected override void OnConfiguring(DbContextOptionsBuilder
options)
        {
            // Example connection string – change as needed
            options.UseSqlServer(@"Server=localhost;Initial
Catalog=blogging_db;User ID=SA;Password=pass;Trust Server
Certificate=True");
        }
    }
}

```

```
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // no relationships yet
    }
}
```

Entity Framework Migrations

in the console run this command `dotnet add package`

`Microsoft.EntityFrameworkCore.Design`

It is used to add the `Microsoft.EntityFrameworkCore.Design` NuGet package to a .NET project. This package provides tools that are needed to manage Entity Framework (EF) Core migrations and design-time operations. It is a development-time-only dependency and is not included in the production build of the application.

Here are the key purposes of the `Microsoft.EntityFrameworkCore.Design` package:

- **Migrations Support:** It provides tools that are required to create and manage EF Core migrations. Migrations are used to update the database schema based on changes in the application's data model. The package enables commands like `dotnet ef migrations add`, `dotnet ef migrations update`, and `dotnet ef database update` to be executed from the command line.
- **Scaffolding Support:** It allows scaffolding of models from an existing database. This can be done using commands like `dotnet ef dbcontext scaffold`, which generates entity classes based on the schema of an existing database.
- **Design-Time Services:** It also includes design-time services required by EF Core to generate a context and model. These services are typically used by tools like the Entity Framework Core CLI or Visual Studio for tasks such as code generation and migrations.

Common Workflow

Development Cycle:

```
# 1. Make model changes  
# 2. Create migration  
dotnet ef migrations add AddNewFeature  
  
# 3. Review migration files  
# 4. Apply to database  
dotnet ef database update  
  
# 5. If needed, rollback  
dotnet ef database update PreviousMigration
```

Fresh Start:

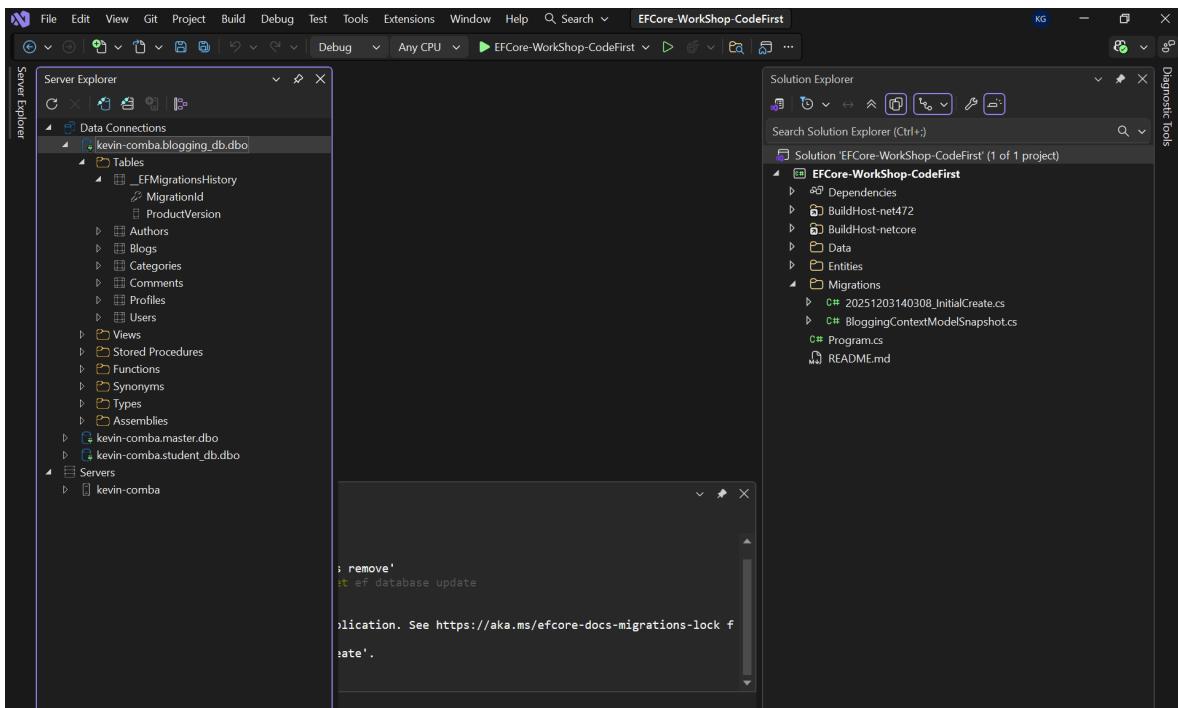
```
dotnet ef database drop --force  
dotnet ef migrations remove  
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

Then Lets continue:

Run below commands

```
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

You can verify the migration. This creates the database with tables for each entity — useful baseline.



image_242.png

One-to-One (1:1) relationships

User - Profile: One-to-One (1-1) : Each User has exactly one Profile that contains profile metadata.

Modify entities to add navigation and FK:

```
// Entities/User.cs – add:
public Profile? Profile { get; set; }
public Author? Author { get; set; }
```

```
// Entities/Profile.cs – add:
public Guid UserId { get; set; }
[ForeignKey(nameof(UserId))]
public User User { get; set; } = null!;
```

```
// Entities/Author.cs – add:
public Guid UserId { get; set; }
```

```
[ForeignKey(nameof(UserId))]
public User User { get; set; } = null!;
```

Then update `DbContext.OnModelCreating (...):`

```
// One-to-One relationship (User - Profile)
modelBuilder.Entity<User>()
    .HasOne(u => u.Profile)
    .WithOne(p => p.User)
    .HasForeignKey<Profile>(p => p.UserId)
    .OnDelete(DeleteBehavior.Cascade);

// One-to-One relationship (User - Author)
modelBuilder.Entity<User>()
    .HasOne(u => u.Author)
    .WithOne(a => a.User)
    .HasForeignKey<Author>(a => a.UserId)
    .OnDelete(DeleteBehavior.Cascade);
```

Now create a new migration:

```
dotnet ef migrations add AddUserProfileAuthor_OneToOne
dotnet ef database update
```

This will alter the database — adding the necessary FK columns and constraints for 1:1 relationships.

One-to-Many (1:n) relationships

Extend entities:

```
// Author.cs
public List<Blog> Blogs { get; set; } = new();

// Blog.cs
public Guid AuthorId { get; set; }
[ForeignKey(nameof(AuthorId))]
```

```

public Author Author { get; set; } = null!;
public List<Comment> Comments { get; set; } = new();

// Comment.cs
public Guid UserId { get; set; }
[ForeignKey(nameof(UserId))]
public User User { get; set; } = null!;
public Guid BlogId { get; set; }
[ForeignKey(nameof(BlogId))]
public Blog Blog { get; set; } = null!;

// Also in User.cs
public List<Comment> Comments { get; set; } = new();

```

Fluent API mappings:

```

// One-to-Many relationship (Author - Blogs)
modelBuilder.Entity<Author>()
    .HasMany(a => a.Blogs)
    .WithOne(b => b.Author)
    .HasForeignKey(b => b.AuthorId)
    .OnDelete(DeleteBehavior.Cascade);

// One-to-Many relationship (Blog - Comments)
modelBuilder.Entity<Blog>()
    .HasMany(b => b.Comments)
    .WithOne(c => c.Blog)
    .HasForeignKey(c => c.BlogId)
    .OnDelete(DeleteBehavior.Restrict);

// One-to-Many relationship (User - Comments)
modelBuilder.Entity<User>()
    .HasMany(u => u.Comments)
    .WithOne(c => c.User)
    .HasForeignKey(c => c.UserId)
    .OnDelete(DeleteBehavior.Restrict);

```

Then run a new migration:

```
dotnet ef migrations add AddBlogAndComment_Relations_OneToMany  
dotnet ef database update
```

The database schema will now include foreign-key constraints mapping authors → blogs, blogs → comments, user → comments.

Many-to-Many (n:n) between Blog and Category

Modify entities:

```
// Blog.cs  
// many-to-many relationship with Category  
public List<Category> Categories { get; set; } = new();  
  
// Category.cs  
// many-to-many relationship with Blog  
public List<Blog> Blogs { get; set; } = new();
```

Fluent API mapping:

```
modelBuilder.Entity<Blog>()  
    .HasMany(b => b.Categories)  
    .WithMany(c => c.Blogs)  
    .UsingEntity(j => j.ToTable("BlogCategories"));
```

Then run:

```
dotnet ef migrations add AddBlogCategory_ManyToMany  
dotnet ef database update
```

EF Core will create a **join table** (e.g. BlogCategories) linking Blog and Category.

Create & Query Data

In Program.cs file add

```
using EFCore_WorkShop_CodeFirst.Data;
using EFCore_WorkShop_CodeFirst.Entities;
using Microsoft.EntityFrameworkCore;

class Program
{
    static void Main()
    {
        using (var context = new BloggingContext())
        {
            // Apply migrations to ensure database is up to date
            context.Database.Migrate();

            // Create a new User with related entities
            var user = new User
            {
                Id = Guid.NewGuid(),
                Username = "john_doe",
                Email = "john.doe1@example.com",
                PasswordHash = "hashedpassword",
                IsActive = true,
                CreatedAt = DateTime.Now,
                UpdatedAt = DateTime.Now
            };

            var profile = new Profile
            {
                Id = Guid.NewGuid(),
                FullName = "John Doe",
                Bio = "Tech Enthusiast",
                AvatarUrl = "https://example.com/avatar.jpg",
                UserId = user.Id,
                CreatedAt = DateTime.Now,
                UpdatedAt = DateTime.Now
            };

            var author = new Author
```

```

    {
        Id = Guid.NewGuid(),
        PenName = "TechGuru",
        Biography = "Author of several tech blogs.",
        UserId = user.Id,
        CreatedAt = DateTime.Now,
        UpdatedAt = DateTime.Now
    };

    context.Users.Add(user);
    context.Profiles.Add(profile);
    context.Authors.Add(author);
    context.SaveChanges();

    Console.WriteLine("Data saved successfully!");

    // Query and display added entities
    var savedUser = context.Users
        .Include(u => u.Profile)
        .Include(u => u.Author)
        .FirstOrDefault(u => u.Username == "john_doe");

    Console.WriteLine($"User: {savedUser?.Username}, Profile: {savedUser?.Profile?.FullName}, Author: {savedUser?.Author?.PenName}");

    // Create categories
    var techCategory = new Category
    {
        Id = Guid.NewGuid(),
        Name = "Technology",
        Description = "All about technology and innovation",
        CreatedAt = DateTime.Now,
        UpdatedAt = DateTime.Now
    };

    var programmingCategory = new Category
    {
        Id = Guid.NewGuid(),

```

```

        Name = "Programming",
        Description = "Software development and programming
tips",
        CreatedAt = DateTime.Now,
        UpdatedAt = DateTime.Now
    };

    context.Categories.Add(techCategory);
    context.Categories.Add(programmingCategory);
    context.SaveChanges();

    Console.WriteLine("Categories created successfully!");

    // Create a blog associated with the author and categories
    var blog = new Blog
    {
        Id = Guid.NewGuid(),
        Title = "Introduction to Entity Framework Core",
        Content = "Entity Framework Core is a modern object-
database mapper for .NET. It supports LINQ queries, change tracking,
updates, and schema migrations.",
        Excerpt = "Learn the basics of EF Core",
        Published = true,
        AuthorId = author.Id,
        CreatedAt = DateTime.Now,
        UpdatedAt = DateTime.Now,
        Categories = new List<Category> { techCategory,
programmingCategory }
    };

    context.Blogs.Add(blog);
    context.SaveChanges();

    Console.WriteLine("Blog created successfully!");

    // Create comments on the blog
    var comment1 = new Comment
    {

```

```

        Id = Guid.NewGuid(),
        Content = "Great article! Very informative.",
        IsApproved = true,
        BlogId = blog.Id,
        UserId = user.Id,
        CreatedAt = DateTime.Now,
        UpdatedAt = DateTime.Now
    };

    var comment2 = new Comment
    {
        Id = Guid.NewGuid(),
        Content = "Looking forward to more posts on this
topic!",
        IsApproved = true,
        BlogId = blog.Id,
        UserId = user.Id,
        CreatedAt = DateTime.Now,
        UpdatedAt = DateTime.Now
    };

    context.Comments.Add(comment1);
    context.Comments.Add(comment2);
    context.SaveChanges();

    Console.WriteLine("Comments created successfully!");

    // Query and display the complete blog with all
relationships
    var blogWithDetails = context.Blogs
        .Include(b => b.Author)
            .ThenInclude(a => a.User)
        .Include(b => b.Comments)
            .ThenInclude(c => c.User)
        .Include(b => b.Categories)
        .FirstOrDefault(b => b.Id == blog.Id);

    if (blogWithDetails != null)

```

```

    {
        Console.WriteLine("\n--- Blog Details ---");
        Console.WriteLine($"Title: {blogWithDetails.Title}");
        Console.WriteLine($"Author:
{blogWithDetails.Author.PenName}
({blogWithDetails.Author.User.Username})");
        Console.WriteLine($"Published:
{blogWithDetails.Published}");
        Console.WriteLine($"Categories: {string.Join(", ", blogWithDetails.Categories.Select(c => c.Name))}");
        Console.WriteLine($"Comments
({blogWithDetails.Comments.Count}):");

        foreach (var comment in blogWithDetails.Comments)
        {
            Console.WriteLine($" - {comment.User.Username}:
{comment.Content}");
        }
    }

    Console.WriteLine("\nAll operations completed
successfully!");
}
}

```

EF Core CLI Database Commands

View Database Information

```
dotnet ef database list
```

Lists all databases for the configured context.

Create/Update Database

```
dotnet ef database update
```

Applies all pending migrations to the database.

```
dotnet ef database update <MigrationName>
```

Updates database to a specific migration (can rollback or forward).

```
dotnet ef database update 0
```

Reverts all migrations (removes all tables).

Drop Database

```
dotnet ef database drop
```

Deletes the database (prompts for confirmation).

```
dotnet ef database drop --force
```

Deletes the database without confirmation.

Migration Commands

Create Migration

```
dotnet ef migrations add <MigrationName>
```

Creates a new migration based on model changes.

Remove Migration

```
dotnet ef migrations remove
```

Removes the last migration (only if not applied).

List Migrations

```
dotnet ef migrations list
```

shows all migrations and their status.

Generate SQL Script

```
dotnet ef migrations script
```

Generates SQL script for all migrations.

```
dotnet ef migrations script <FromMigration> <ToMigration>
```

Generates SQL for specific migration range.

DbContext Commands

```
dotnet ef dbcontext info
```

Shows information about the DbContext.

```
dotnet ef dbcontext list
```

Lists all available DbContext types.

```
dotnet ef dbcontext scaffold <ConnectionString> <Provider>
```

Scaffolds a DbContext from an existing database (Database-First).

Common Workflow

Development Cycle:

```
# 1. Make model changes  
# 2. Create migration  
dotnet ef migrations add AddNewFeature  
  
# 3. Review migration files  
# 4. Apply to database  
dotnet ef database update  
  
# 5. If needed, rollback  
dotnet ef database update PreviousMigration
```

Fresh Start:

```
dotnet ef database drop --force  
dotnet ef migrations remove  
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

ASP.NET Core

Start typing here...

Introduction & Env Setup

Start typing here...

ASP.NET Web API

ASP.NET Core Web API is Microsoft's modern framework for building **HTTP-based services (RESTful APIs)** on top of the .NET Core platform. It allows developers to create robust and flexible APIs that various clients can consume, such as web applications, mobile apps, desktop applications, and third-party services. It enables developers to create lightweight, scalable, and secure APIs that multiple types of clients can consume.

Prerequisites to Learn ASP.NET Core Web API

Learning to develop with ASP.NET Core Web API involves understanding general development concepts and specific technologies related to Web API development. Here are the prerequisites that are good if you know before learning ASP.NET Core Web API:

- **Basic Knowledge of C#:** ASP.NET Core is built on C#, so a solid understanding of C# Programming is essential. You should be comfortable with C# syntax, basic programming constructs like loops and conditionals, classes and objects, and more advanced concepts such as LINQ, async/await, and exception handling. This is Mandatory.
- **Understanding .NET Core Basics:** Familiarity with the .NET Core framework is important. This includes understanding the .NET Core CLI, the structure of .NET Core applications, basic concepts like dependency injection, and how to use NuGet packages. We have already discussed this in our ASP.NET Core Basic course, and it is mandatory.
- **Familiarity with Entity Framework Core:** Entity Framework Core (EF Core) is the recommended ORM for data access in ASP.NET Core applications. Understanding EF Core for performing CRUD operations with databases is highly beneficial. This is mandatory.
- **Basic Database Knowledge:** Basic knowledge of databases, especially relational databases like SQL Server, MySQL, or Oracle, is important. You should know how to design databases, write basic SQL queries, and understand concepts like tables, keys, and relationships. This is mandatory.

Why is Web API

Web APIs (Application Programming Interfaces) are essential for allowing different software applications to communicate and exchange data with each other over the Internet. They enable data exchange between various platforms, applications, and devices, regardless of their underlying technologies.

- Web APIs enable systems developed on different technologies (like different programming languages or frameworks) to communicate effectively. For example, a mobile app built with Swift (iOS) can interact with a backend service written in ASP.NET Core (C#) using HTTP-based APIs.
- They facilitate integration between different services and applications. For instance, a weather forecasting service may expose a Web API that other applications can query to get weather data, which they can then integrate into their own functionality.
- Web APIs provide a flexible way to access and manipulate data. They allow developers to create custom client applications that can consume these APIs based on their specific needs.

Key Characteristics of Web APIs:

- **HTTP-Based Communication:** Web APIs are designed to work over HTTP, the same protocol used for Web Browsing. This means APIs can be accessed using standard HTTP methods like GET, POST, PUT, DELETE, etc. The API endpoints are typically represented as URLs (Uniform Resource Locators).
- **Data Exchange Formats:** Web APIs use standardized data exchange formats such as JSON (JavaScript Object Notation) and XML (Extensible Markup Language) to structure and transmit data between the client and server. JSON has become the most popular format due to its simplicity and ease of use.
- **RESTful Architecture:** Web APIs are designed to follow Representational State Transfer (REST) principles. A RESTful API is stateless, uses standard HTTP methods, and organizes resources into a hierarchy with unique URLs for each resource.

- **Authentication and Authorization:** Web APIs implement security mechanisms for authentication and authorization to ensure that only authorized clients can access resources or perform specific actions. Common authentication methods include API keys, OAuth, and JWT (JSON Web Tokens).

Basics of Web APIs

Why do we need Web APIs?

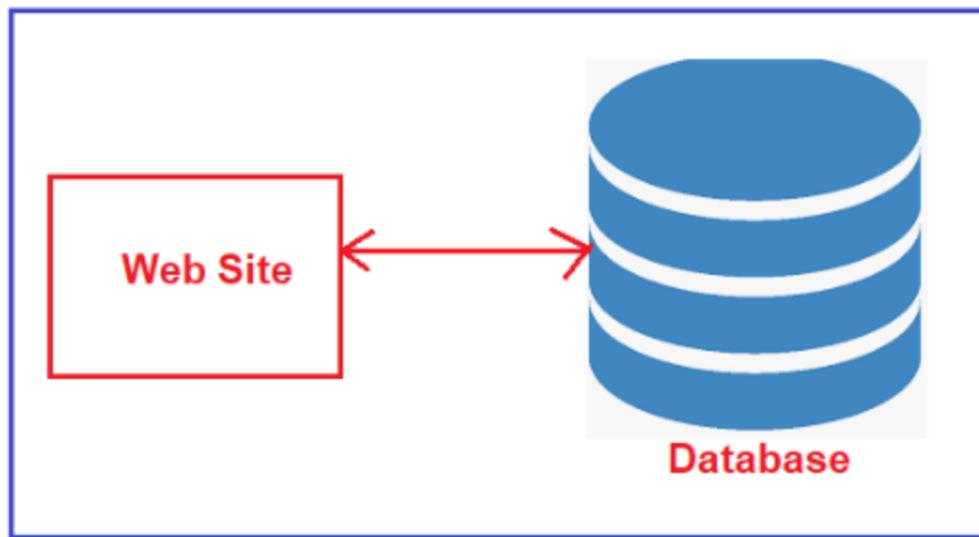
In today's multi-device world, Web APIs are the backbone of modern applications, powering websites, mobile apps, and even smart devices. Let us understand the need for Web APIs with some realistic examples.

Step 1 – Initial Setup (Only Website + Database)

Suppose you have an idea to develop and launch a product. For this, you need to develop a website and launch this product. Then what will you do? The simplest setup would be:

- A **Website** built using **ASP.NET MVC**, **ASP.NET Core**, **PHP**, **JSP**, or any other **server-side technology** that is available in the market.
- Of course, a **Database** such as **SQL Server**, **MySQL**, or **Oracle** to store all your product's business data.

With this setup, your website becomes a dynamic, fully functional system that can create, read, update, and delete data from the database.



image_243.png

Step 2 – Business Grows (Website + Android + iOS Apps)

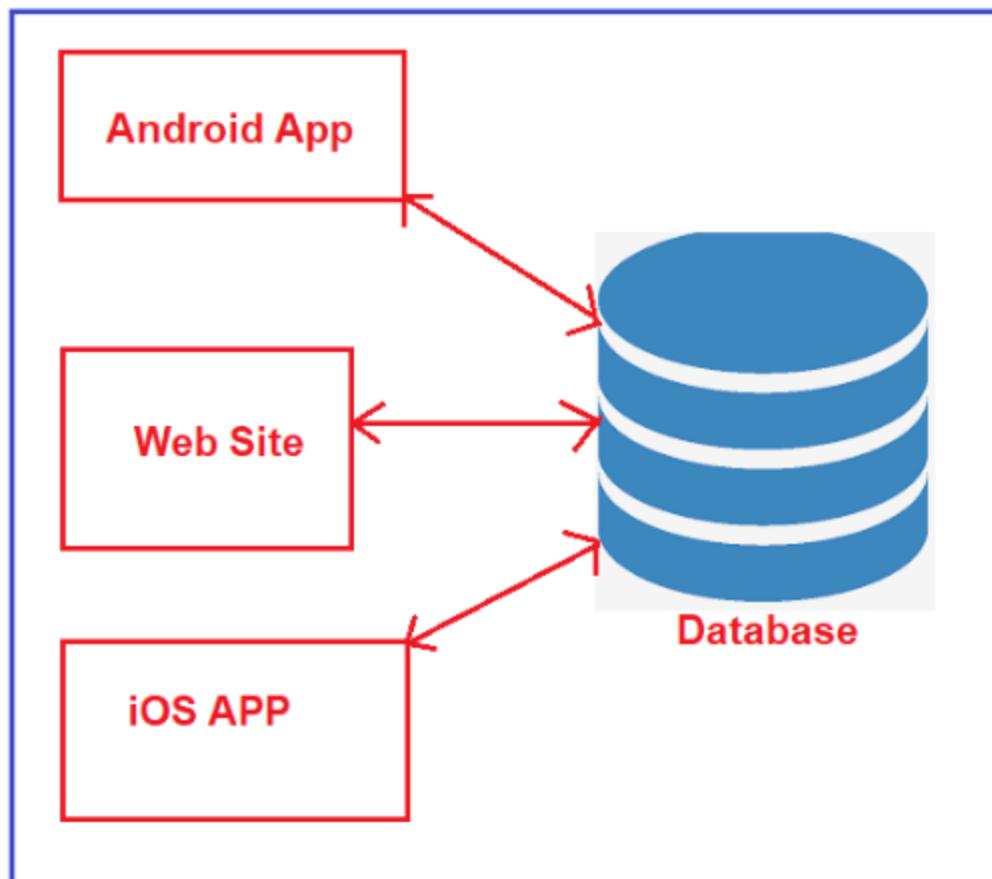
Over time, your business grows. Now, besides the website, you also want:

- An **Android Mobile App**
- An **iOS Mobile App**

This means:

- You now have **three different applications** (Website, Android, iOS).
- But you still have **only one database** that stores all your data.
- So, we have three different applications and one database.

Now, all these three applications have to interact with the database, as shown in the below image.



image_244.png

Step 3 – Problems With Direct Database Access

If all these three applications directly interact with the database, we have some problems. Let us understand the problems first, and then we will see how to overcome the above problems:

- **Duplicate Logic Across Applications:** Each application must implement the same business logic independently. This causes Code Duplication.
- **Time-Consuming and Error-Prone:** Writing the same logic in three places increases the chances of mistakes. If a piece of logic changes, you need to update it everywhere. This will add more errors to your application
- **Front-End Framework Limitations:** Some frameworks (like Angular or React) cannot directly talk to a database, because they are client-side technologies. They need a middle layer.
- **Hard to Maintain:** If you need to update or fix business logic, you must do so in multiple applications, which is a complex and inefficient process.

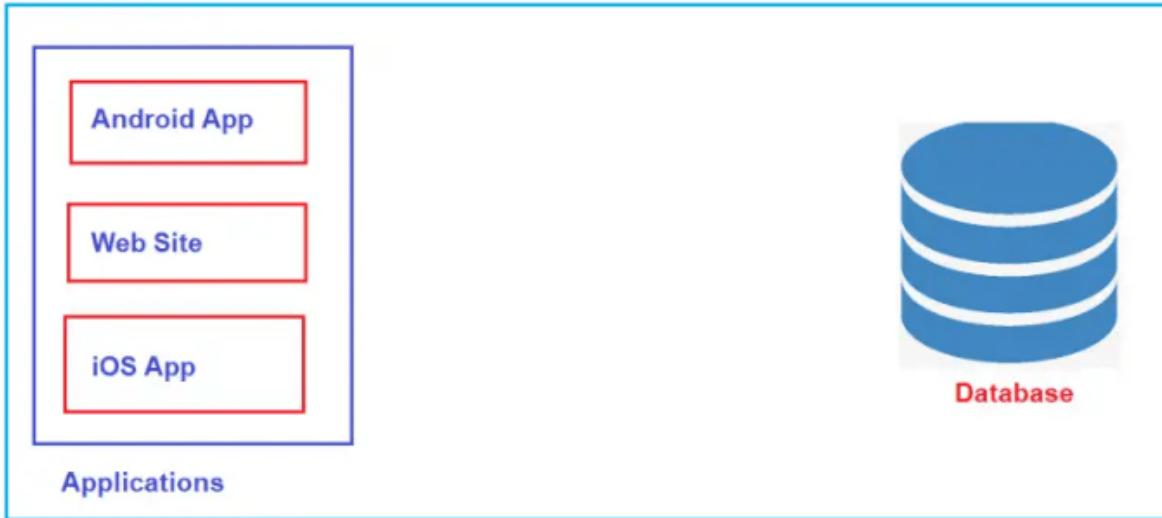


In short, **direct database communication is messy, unsafe, and hard to scale.**

There are also other problems that we face in this structure. Let us see how to overcome the above problems, or, in other words, why we need Web APIs.

The Need for Web APIs

As you can see in the image below, we have three applications on the left-hand side, and on the right-hand side, we have the database.

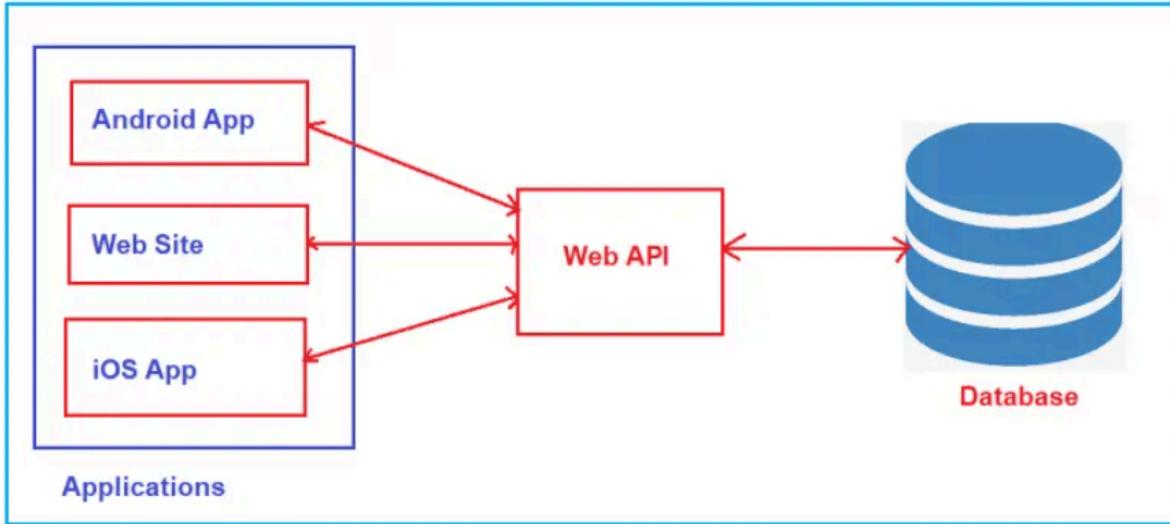


image_245.png

We aim to establish communication between all three applications and the database to manage the data. So, what will we do? We will add a **Web API layer** between the **frontend applications (website, Android, iOS)** and the **backend database**.

- Instead of each application directly accessing the database, they all communicate with the **Web API**.
- The **Web API** contains all the **business logic** and manages all communication with the database.
- This makes the **Web API act as a mediator** between the frontend and the backend.

As a result, the database becomes secure, logic is centralized, and applications become easier to maintain and extend.



image_246.png

⚠ Note: The Website, Android, and iOS applications do not have direct access to the database. They only need to communicate with the Web API Project, and it is the Web API project's responsibility to interact with the database. The entire business logic will be written in the Web API project only. So, Web API acts as a mediator between the Front-End and Back-End.

What is Rest?

REST (Representational State Transfer) is not a technology or a framework, but an architectural style for designing **Distributed Systems**. Its primary purpose is to define how **clients** (like browsers, mobile apps, or other applications) and **servers** (back-end services) should communicate over a network.

The beauty of REST lies in its **Platform Independence**:

- The client could be written in Java, .NET, Angular, React, or any other technology.
- The server could be built using Node.js, ASP.NET Core, PHP, Python, or Java.
- As long as they both follow REST principles and communicate via **HTTP**, they can exchange data seamlessly

REST treats **Everything as a resource**: books, users, orders, products, etc. Each resource is uniquely identified by a **URI (Uniform Resource Identifier)**, and standard **HTTP**

methods (GET, POST, PUT, DELETE, PATCH, etc.) are used to perform operations (CRUD – Create, Read, Update, Delete) on these resources.

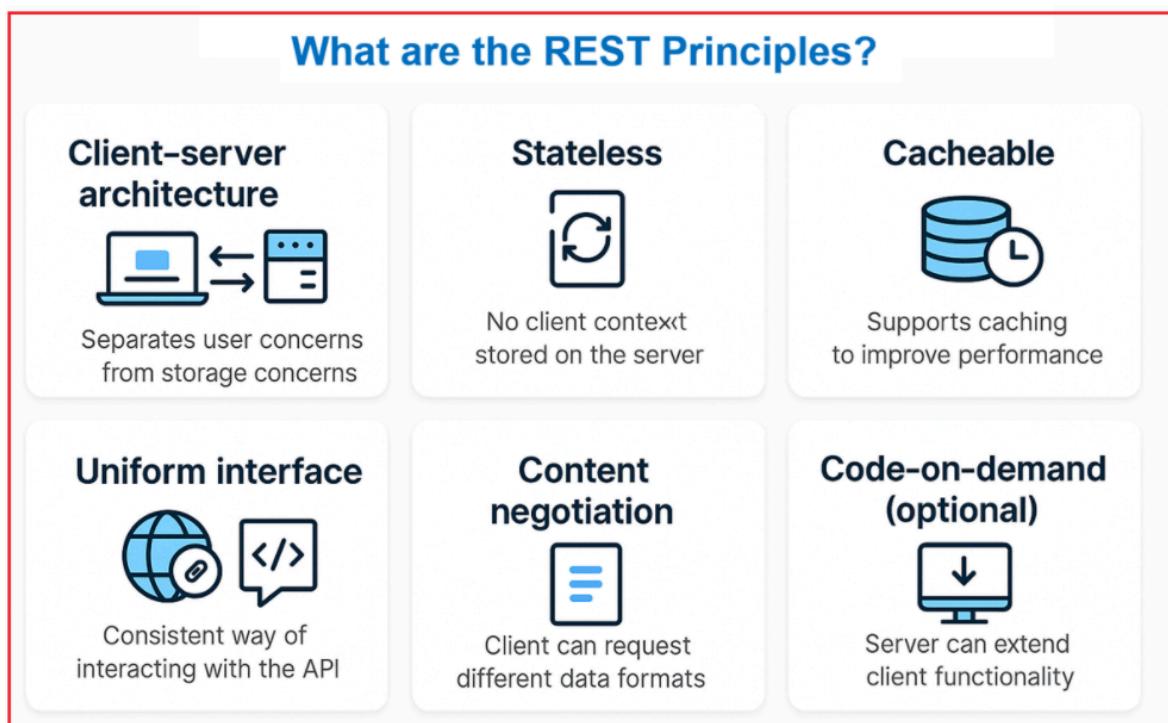
Example – Book Management System API:

- GET /books → Retrieve all books.
- GET /books/ → Retrieve details of a specific book.
- POST /books → Add a new book.
- PUT /books/ → Update details of a book.
- DELETE /books/ → Delete a book.

This clear and uniform way of accessing resources is what makes REST widely adopted in modern web APIs.

What are the REST Principles?

REST defines **six key principles (constraints)** that make a system truly RESTful. Let us proceed and understand the REST constraints or principles.



image_247.png

Client-Server Architecture:

REST enforces a clear separation of responsibilities:

- **Client** → Handles user interface, presentation, and experience.
- **Server** → Handles business logic, processing, and database/storage.

They communicate using a **standard interface (HTTP)**.

Why it matters:

- Client and server can be developed, deployed, and scaled **independently**.
- UI changes (say, redesigning a mobile app) don't affect the back-end logic.
- Server upgrades (such as database migrations) don't break clients.



Example: In Instagram, your mobile app (client) fetches profile data via the GET /users/ endpoint. The server retrieves the data and returns it, regardless of whether the request originated from iOS, Android, or a web browser.

Stateless:

The **stateless constraint** specifies that client-server communication must be **stateless between requests**. Each request from the client-server must be **independent**. The server does **not store any session data** between requests. Every request from the client should carry **all necessary information** (authentication, parameters, etc.) for the server to process it.

Why it matters:

- Any server in a cluster can handle any request (better scalability).
- Simplifies server design (no need to track client state).
- Improves reliability (no “session lost” issues).
- **Each request from the clients can be treated independently by the server.**

⚠ Example: When a client calls GET /orders/789, it sends an Authorization token in headers. The server validates the token, fetches the order, and responds. The server doesn't need to remember past interactions.

Cacheable:

In real-time applications, some data provided by the server is not changed frequently, such as the list of Countries, States, Cities, and products. REST allows clients (and intermediate proxies) to cache responses, improving performance and reducing server load. The server informs the client whether a response is cacheable and for how long, using headers such as Cache-Control or Expires.

Why it matters:

- Faster responses for frequently accessed data.
- Reduces redundant server calls.
- Optimizes bandwidth usage.
- Frequently requested resources can be served from a cache instead of re-fetching from the server.
- Reduces the number of round-trip requests to the server.

Example:

- A request GET /countries may rarely change. The server responds with:
- Cache-Control: max-age=86400
- This means the client can reuse the response for 24 hours without needing to request it again from the server.

Uniform Interface:

The most critical principle: all resources are accessed in a **consistent and standardized way**. This makes APIs predictable and easier to use.

It has four sub-constraints:

- **Resource Identification** → A URI uniquely identifies each resource. Example: /users/1001 identifies a specific user.
- **Manipulation via Representations** → Clients modify resources by sending their representation (JSON/XML). Example: PUT /users/1001 with JSON body updates a user.
- **Self-Descriptive Messages** → Requests and responses carry enough information (HTTP method, headers, content type) so clients/servers understand how to process them.
- **HATEOAS (Hypermedia as the Engine of Application State)** → Responses guide clients with links to available next actions.

Example: A response for a book might include links like:

```
{
  "Id": 101,
  "Title": "REST Basics",
  "Links": [
    {"rel": "update", "href": "/books/101", "method": "PUT"},
    {"rel": "delete", "href": "/books/101", "method": "DELETE"}
  ]
}
```

Why it matters:

- Clients know exactly how to interact with the API.
- Reduces coupling between client and server.
- Easier for new clients to consume APIs.

Examples:

- GET <https://example.com/books> retrieves a list of books.
- GET <https://example.com/books/123> retrieves a book with ID 123.

- POST `https://example.com/books` creates a new book.
- PUT `https://example.com/books/123` updates the book with ID 123.
- DELETE `https://example.com/books/123` deletes the book with the specified ID.

Content Negotiation:

Clients and servers can negotiate the media type used for representing a resource. The client typically specifies its preferred format via the Accept header, and the server responds in one of the requested formats if supported.

Why It Matters:

- Allows different clients (mobile apps, browsers, etc.) to request the format they can best handle (e.g., JSON, XML, HTML).
- Servers can support multiple formats without changing the resource identifier.

Example: If a client requests an article by calling `GET /articles/456` and includes `Accept: application/json`, the server can respond with JSON (e.g., `Content-Type: application/json`). If the client had requested `Accept: application/xml`, the server could respond with XML if it supports it.

Layered System:

REST allows APIs to be built as a **layered architecture**. Each layer has its own responsibility (e.g., security, caching, load balancing), and clients are unaware of the number of layers that exist.

Why it matters:

- Scalability → Easy to add caching servers, gateways, or load balancers.
- Security → Middle layers can handle authentication or rate limiting.
- Flexibility → Layers can develop independently.

Example: In an e-commerce app:

1. **Layer 1:** Web Server handles client requests.

2. Layer 2: Application Layer applies business logic.

3. Layer 3: Database stores data.

Clients don't know about internal layering; they just see a single API endpoint.

Code on Demand (Optional)

A REST API may optionally send **executable code** to clients (e.g., JavaScript). This allows extending client functionality without additional requests. **This is an optional constraint used to add flexibility for the client.**

Why it matters:

- Reduces server load.
- Allows clients to perform operations locally.

Example: A server sends a JavaScript snippet with sorting logic for a product list. The client runs it locally, avoiding repeated server requests for sorting.

HTTP (HyperText Transport Protocol)

The Hypertext Transfer Protocol (HTTP) is the foundation of communication on the World Wide Web, enabling interaction between clients and servers. It defines how requests and responses are exchanged, ensuring that resources such as web pages, images, stylesheets, and APIs are delivered correctly and efficiently.

What is HTTP?

HTTP (Hypertext Transfer Protocol) is the backbone of data communication on the web. It defines the rules for how clients (like browsers or apps) send requests to servers and how servers respond with data. This protocol ensures that when you access a website, the content (HTML, images, CSS, scripts) is transmitted properly between your device and the server hosting the site. HTTP works as a request-response protocol where the client initiates communication and the server fulfills it.

Important Points

- HTTP stands for **Hypertext Transfer Protocol**.

- It is the foundation of communication on the World Wide Web.
- Defines how messages are formatted and transmitted.
- Enables browsers, apps, and tools to send **requests** and receive **responses**.
- Operates in a **client–server model**.

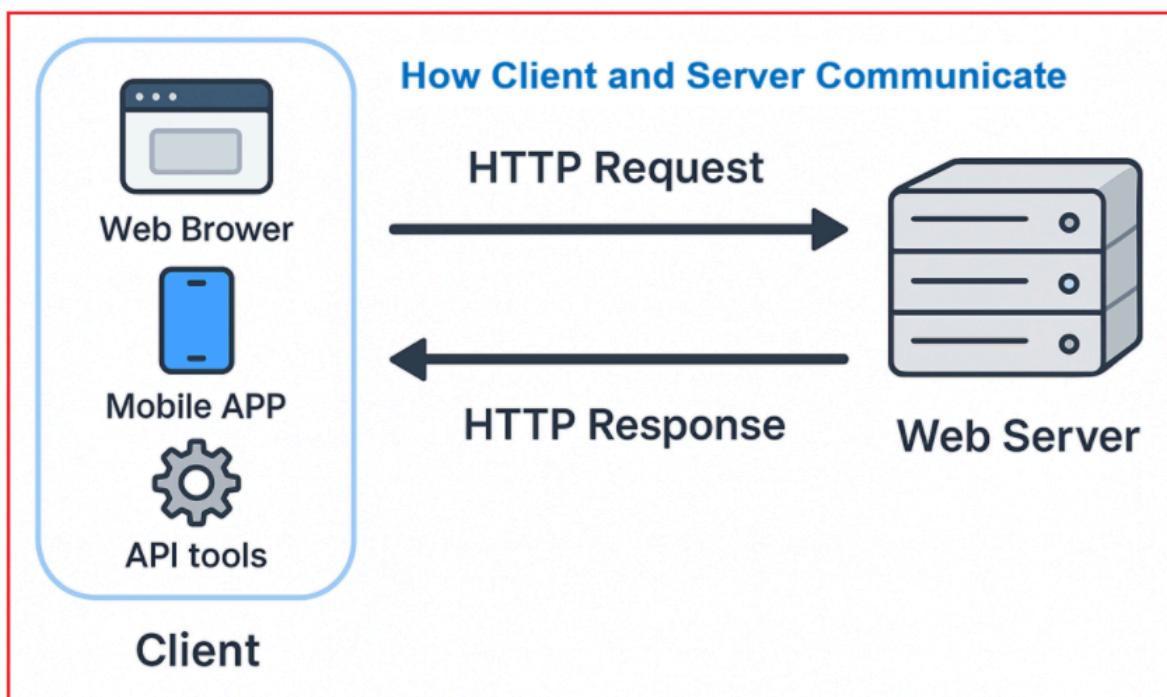
Example

When you type <https://www.example.com> into your browser and press Enter:

- The browser sends an **HTTP Request** to the server hosting `example.com`.
- The server replies with an **HTTP Response** containing the **HTML page**.
- The browser then displays that page to you.

How Client and Server Communicate

Communication between a browser (client) and a web server occurs through HTTP, in the form of requests and responses, within a **client–server architecture**.



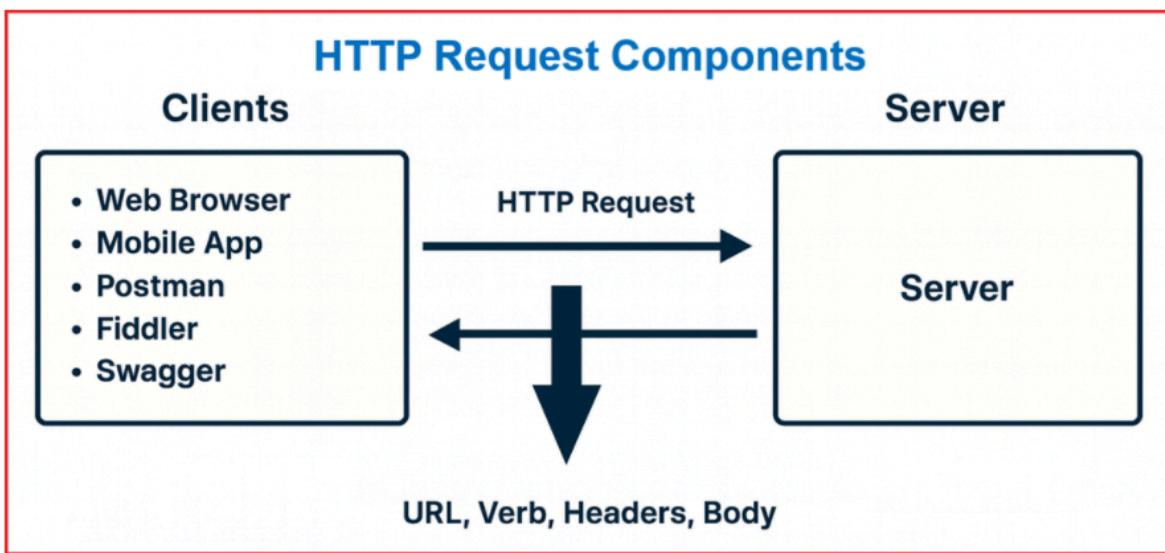
image_248.png

Important Points

- **Client:** Browser, mobile app, or tool (like Postman, Swagger, Fiddler).
- **Server:** Hosts the web application or API.
- **Request:** Data sent from client to server (e.g., asking for a webpage or JSON data).
- **Response:** Data sent from the server to the client (e.g., HTML page, JSON).
- **Protocol:** Communication is possible only through HTTP/HTTPS.

HTTP Request Components

An **HTTP Request** is the message sent by the client (browser, mobile app, Postman, Swagger, etc.) to the server asking for a resource or action. Every request must follow a specific structure so that the server can understand what the client needs. It consists of a **request line**, **headers**, an optional **body**, and sometimes **query parameters** in the URL. Together, these components define **what resource is being requested, how it should be processed, and any additional data or context needed**.



image_249.png

Request Line:

The **Request Line** is the first line of an HTTP request and specifies to the server the exact action the client wants to perform, the resource it is targeting, and the HTTP version it supports. It always contains three parts:

HTTP Method: This represents the type of operation the client wants to perform on the server. For example:

- **GET** → Retrieve a resource without making changes (e.g., viewing a webpage).
- **POST** → Send new data to the server for processing (e.g., submitting a login form).
- **PUT** → Update an existing resource completely.
- **DELETE** → Remove a resource from the server.
- Other methods include **PATCH** (Partial Update), **HEAD** (request headers only), and **OPTIONS** (check supported methods).

Request Headers:

After the request line, the client usually sends **one or more headers**. These headers serve as additional instructions, providing more detailed descriptions of the request. They are written in **key-value** format (e.g., Host: www.example.com) and each appears on a separate line. Common request headers include:

- **Host:** Specifies the domain name of the target server (Host: www.example.com).
- **User-Agent:** Identifies the client software making the request (e.g., browser, mobile app, Postman, etc.).
- **Accept:** Lists the content types the client is willing to accept from the server (e.g., HTML, JSON, XML).
- **Content-Type:** When the request includes a body (like a POST, PUT, or PATCH request), this header indicates the format of the request body. Example: application/json, application/xml, etc.
- **Authorization:** Carries credentials, such as API keys or tokens, for authentication.
- **Cookie:** Includes any cookies previously stored by the browser to maintain sessions. This is used for state management.
- **Cache-Control:** Provides caching rules for both client and server.

Example:

- Host: www.example.com

- User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36
- Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8

Request Body (Optional):

The request body is the part of an HTTP request that carries data from the client to the server.

- The body of an HTTP request is **optional** and is generally used in methods such as **POST**, **PUT**, and **PATCH**, where the client needs to send data to the server for processing.
- The data can also be:
 - Form submissions (e.g., login details).
 - File uploads (images, PDFs).
 - JSON or XML payloads for APIs.

Example Body (JSON in a POST Request):

For instance, in a POST request, the body would contain the JSON data as follows.

```
{  
  "Username": "ExampleUser",  
  "Password": "ExamplePass"  
}
```

Query Parameters:

Query parameters are part of the URL itself. They are used to send extra details to the server in the form of key-value pairs.

- They start with a **question mark (?)**.
- They are sent in the form of a key-value pair.
- Multiple parameters are separated by an **ampersand (&)**.

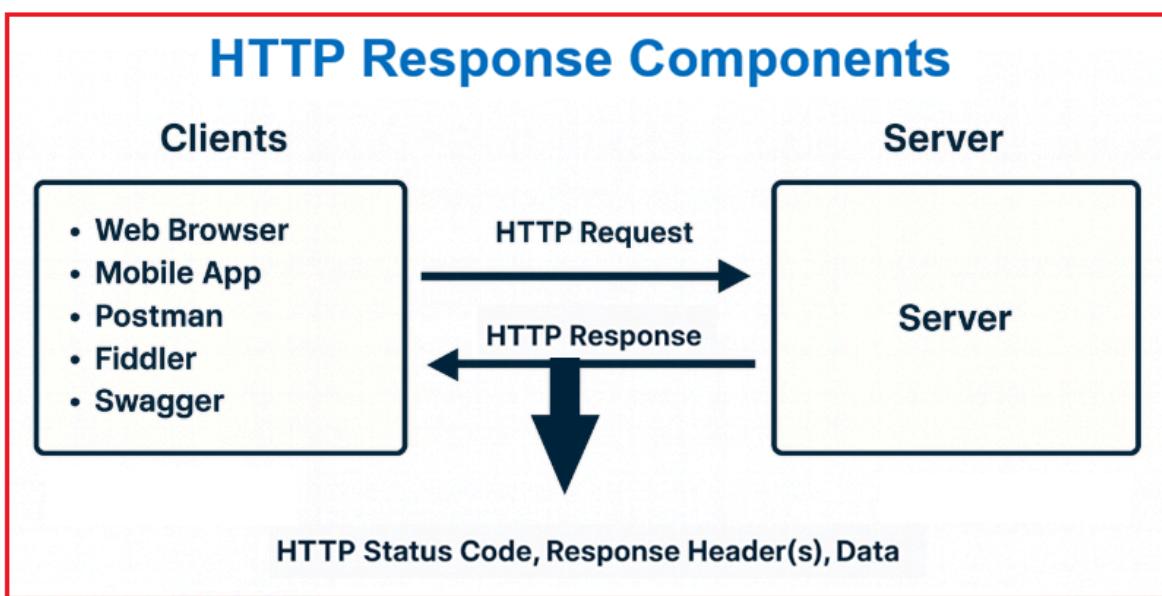
- These parameters are beneficial for **search**, **filtering**, **sorting**, or **pagination** operations on the server.

Example URL with Query Parameters: <https://example.com/search?q=example&sort=asc>

q=example: The query parameter q has the value example. **sort=asc:** The query parameter sort has the value asc.

HTTP Response Components

An HTTP response is the message sent from the **Server** back to the **Client** after processing a request. The response informs the client whether the request was successful or not and may also include the requested content or an error message. An HTTP response is composed of a **Status Line**, **Response Headers**, and optionally, a **Response Body**. This response structure is crucial for the client to understand what action the server took and what data is being returned. For a better understanding, please have a look at the following diagram:



image_250.png

Response Line:

The **Response Line** (also known as the **Status Line**) is the first line of an HTTP response. It provides the client with essential information about the response that the server is sending back. It is always made up of three parts:

HTTP Version: Tells the client which version of HTTP the server is using. Common versions include **HTTP/1.1** (still widely used), **HTTP/2** (faster with multiplexing), and

HTTP/3 (the latest, offering improved performance and security).

Status Code: A **three-digit number** that represents the outcome of the client's request. The Status codes are grouped into categories:

1. 1xx (Informational): Request received, processing continues.
2. 2xx (Success): Request was successful (e.g., 200 OK, 201 Created).
3. 3xx (Redirection): Client must take additional steps (e.g., 301 Moved Permanently).
4. 4xx (Client Errors): The request has an error (e.g., 400 Bad Request, 404 Not Found).
5. 5xx (Server Errors): The server failed to process the request (e.g., 500 Internal Server Error).

Status Text: A short, human-readable description of the status code (e.g., OK for 200, Not Found for 404).

Example Response Line: *HTTP/1.1 200 OK* This means:

- The server used HTTP version 1.1,
- The status code is 200 (success),
- The status text is OK.

Response Headers:

Each HTTP Response can have one or more Response Headers. The **Response Headers** come immediately after the response line and are composed of key-value pairs. These headers provide **additional information** about the response and the server. Each header is on its own line. Common response headers include:

- **Content-Type:** Specifies the type of data in the response body (e.g., text/html for webpages, application/json for APIs).
- **Content-Length:** Specifies the size of the response body in bytes (helps the client know how much data to expect).
- **Server:** Provides information about the server software that handled the request (e.g.,

Apache, Nginx, IIS, Kestrel, etc.).

- **Set-Cookie:** Used to send cookies from the server to the client browser. This is useful for session management and personalization.
- **Cache-Control:** Tells the client or proxy how caching should be handled (e.g., store for a day, don't cache at all).
- **Date:** The timestamp when the server sent the response

Example Response Headers:

- Content-Type: text/html; charset=UTF-8
- Content-Length: 138
- Server: Apache/2.4.41 (Ubuntu)

Response Body:

The Response Body is the part of the HTTP response that carries the **actual data** being returned to the client. Whether or not a body is included depends on the request method and the status code. For example:

- A GET request typically includes a body containing the requested content.
- A 204 No Content response intentionally has no body.

The data in the response body can be in various formats, such as:

- HTML (for rendering web pages),
- JSON or XML (for APIs),
- Plain text,
- Images, files, or other media types.

The format and presence of the body depend on the Content-Type header sent by the server.

Example JSON Response Body:

```
{  
    "Id": 123,  
    "Name": "John Doe",  
    "Email": "john.doe@example.com"  
}
```

HTTP Verbs or HTTP Methods:

GET HTTP Method: The GET method is the most commonly used HTTP method. It is designed to retrieve information from the server without changing anything on the server.

POST HTTP Method: The POST method is used to send data to the server, usually resulting in the creation of a new resource or triggering a process. Unlike GET, it modifies the server state by adding or processing data.

Example:

POST /articles HTTP/1.1

Host: example.com

Content-Type: application/json

```
{  
    "Title": "New Article",  
    "Content": "This is the content of the new article."  
}
```

PUT HTTP Method: The PUT method is used to completely update or replace an existing resource with the new data provided in the request body. If the resource does not exist, some servers may create it; however, the primary purpose is typically to perform a complete update or replacement of the existing resource.

Example:

PUT /articles/101 HTTP/1.1

Host: example.com

Content-Type: application/json

```
{  
  "Id": 101,  
  "Title": "Updated Article",  
  "Content": "This is the updated content of the article."  
}
```

⚠ Note: Ensure the ID in the URL path (/articles/101) matches the resource identifier in the body ("Id": 101) if your API design requires it.

PATCH HTTP Method: The PATCH method is used for Partial Updates to an existing resource. Unlike PUT, which replaces the entire resource, PATCH updates only the fields provided in the request body. This makes PATCH more efficient when only a subset of data needs to be changed.

Example:

PATCH /articles/1 HTTP/1.1

Host: example.com

Content-Type: application/json

```
{  
  "Content": "This is the updated content of the article."  
}
```

⚠ Note: Commonly, you do not include the resource's ID in the body if the ID is already specified in the URL, unless your API design specifically requires it.

DELETE HTTP Method: The DELETE method is used to remove a resource from the server. Once deleted, attempting to access the same resource should ideally return a 404 Not Found error or indicate that it's inactive. Many applications use either soft delete (marking an item as inactive) or hard delete (permanently removing an item).

Example:

DELETE /articles/1 HTTP/1.1

Host: example.com

Deletion Strategies: In modern applications, we can implement Delete Functionality in two ways: Soft Delete and Hard Delete.

HTTP Status Code Categories:

HTTP status codes are three-digit numbers sent by the server in an HTTP response. They inform the client whether the request was successful, requires further action, or has failed due to an error.

They are as follows: Here, XX will represent the actual number.

1. 1xx (Informational Responses): Request received; server is working on it.
2. 2xx (Successful Responses): Request understood and successfully processed.
3. 3xx (Redirection Responses): Further action is needed to complete the request.
4. 4xx (Client Error Responses): There was an issue with the client's request.
5. 5xx (Server Error Responses): The server encountered an issue processing the request.

1XX: Informational Response Status Codes

The 1xx category represents informational responses.

Examples:

- **100 Continue** → Server acknowledges headers and asks the client to send the request body (useful before sending large data).
- **102 Processing** → Server is still working on the request, but hasn't finished (used in WebDAV and long tasks).

2XX: Successful Response Status Codes

The 2xx category indicates that the client's request was successfully received, understood, and processed.

Examples:

- **200 OK:** The standard response for successful HTTP requests. The requested resource (or data) is usually returned in the response body. For example, retrieving a web page

or fetching data from an API.

- **201 Created:** The request was successful, and a new resource was created as a result. It is commonly used for POST requests. For example, creating a new user account or posting a new article.
- **202 Accepted:** The request has been accepted for processing, but it has not yet been completed. This is often used for asynchronous tasks. For example, initiating a background job or asynchronous task.
- **204 No Content:** The server successfully processed the request and is not returning any content. Often used after a successful **DELETE** or **PUT** operation when there is no need to return data. For example, deleting a resource or updating a record without returning the updated data.

3XX: Redirection Response Status Codes

The **3xx category** indicates that the client must take further action to complete the request, typically by making another request to a different location. These codes are essential for **redirecting users** or handling resource movements, such as when websites change their domain names.

Examples:

- **301 Moved Permanently:** The resource has been permanently moved to a new URI provided in the Location header. The ideal use case is redirecting from an old domain to a new one. Future requests should use the new URI.
- **302 Found:** The resource is temporarily located at a different URI. The client should use the new URI for this request, but future requests can still use the original URI. An ideal use case is temporary redirection during maintenance.



Real-Life Example: If you type <http://example.com>, the server may respond with 301 Moved Permanently to redirect you to <https://example.com>.

4XX: Client Error Response Status Codes

The **4xx category** indicates that the client made an error in the request. These errors are caused by incorrect input, unauthorized access, or asking for resources that don't exist.

Examples: **400 Bad Request**: The server cannot process the request due to malformed syntax or invalid data. For example, sending malformed JSON in a POST request. **401 Unauthorized**: Authentication is required and has failed or has not been provided. For example, accessing a protected resource without valid credentials. **403 Forbidden**: The server understands the request but refuses to authorize it. For example, accessing a resource without sufficient permissions. **404 Not Found**: The requested resource could not be found on the server. For example, requesting a non-existent webpage or API endpoint. **405 Method Not Allowed**: The HTTP method used in the request is not supported for the requested resource. For example, sending a PUT request to an endpoint that only supports GET and POST.



Real-Life Example: If you mistype a page URL like /artcles instead of /articles, the server responds with 404 Not Found.

5XX: Server Error Response Status Codes

The 5xx category indicates the request was valid, but the server failed to fulfil it due to an internal problem.

Examples:

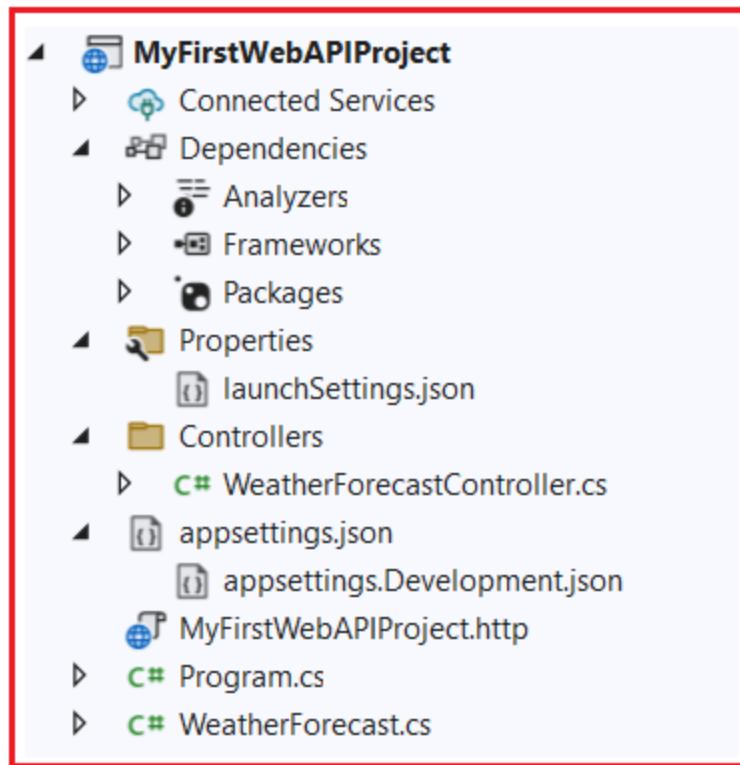
- **500 Internal Server Error**: A generic error message indicating an unexpected condition that prevented the server from fulfilling the request. For example, an unhandled exception in server-side code.
- **502 Bad Gateway**: The server, while acting as a gateway or proxy, received an invalid response from the upstream server. For example, failures in communication between proxy servers.
- **503 Service Unavailable**: The server is currently unable to handle the request due to temporary overload or maintenance. For example, scheduled maintenance periods or unexpected traffic spikes. The client should try the request again later.
- **504 Gateway Timeout**: The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server. For example, slow responses from a backend service can lead to a timeout. The client may need to retry the request later.

- ⚠** Real-Life Example: During a high-traffic sale, a website may return a 503 Service Unavailable error because its servers are overloaded.

ASP.NET Core Web API – Basics

Default ASP.NET Core Web API Files and Folders

When you create a new ASP.NET Core Web API project in Visual Studio 2022 Or 2026 with .NET 10.0, a predefined set of files and folders is automatically generated. This structure serves as the foundation of the application, ensuring developers have a well-organized starting point.



image_251.png

Connected Services

The Connected Services node in Solution Explorer allows developers to easily integrate external services into their applications without manually writing commonly used code for connectivity, authentication, or configuration.

It provides integration for services such as:

- **Azure Services** – Azure Storage, Azure App Services, Azure Key Vault, etc.

- **Third-party REST APIs** – For example, Google APIs, Stripe for payment processing, or Twilio for SMS.
- **Microsoft Office 365 APIs** – Accessing Office 365 resources like emails, calendars, or SharePoint.

Benefits of Connected Services:

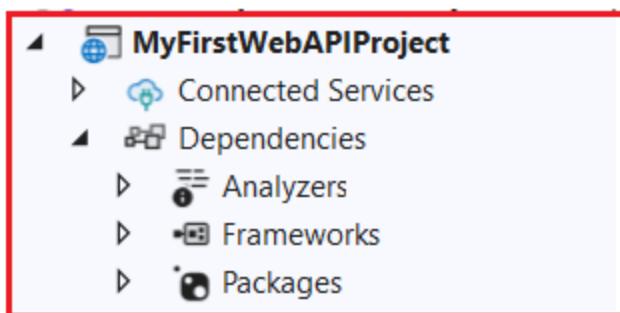
- **Automatic Configuration** – Adds the necessary configuration files and entries to appsettings.json.
- **Client Code Generation** – Creates client libraries that allow developers to call external APIs as strongly typed methods instead of crafting raw HTTP requests.
- **Dependency Management** – Installs required NuGet packages automatically.



Example: If you want to use Azure Key Vault for securely storing API keys or database connection strings, you can add it via Connected Services, and Visual Studio will handle package installation, configuration, and secret retrieval setup.

Dependencies:

The Dependencies folder shows all **Packages**, **Analyzers**, **SDKs**, and **Frameworks** your project depends on. It ensures that everything needed to build and run your Web API is properly referenced. This makes it easy to see exactly what external components are powering your application. When expanded, it typically contains three key categories: **Analyzers**, **Frameworks**, and **Packages**, as shown in the below image.



image_252.png

Analyzers

Analyzers are **Static Code Analysis Tools** that run inside Visual Studio and during build time, it is used to improve code quality. They help with:

- **Code Quality Checks:** Detecting potential bugs, unused variables, or unsafe patterns.
- **Style Enforcement:** Ensuring coding conventions (e.g., naming conventions, spacing rules, formatting, etc.) are followed.
- **Performance Suggestions:** Recommend better coding practices for speed and memory efficiency.

Frameworks

The Frameworks section lists the core libraries and runtime components that your project depends on. By default, two major framework groups are included. They are as follows:

Microsoft.NETCore.App:

This Includes the .NET runtime, Base Class Libraries (BCL), Garbage Collector, JIT compiler, and other core functionality required to run .NET applications.

- **Base Class Libraries (BCL):**

- System.IO → File and stream handling.
- System.Collections → Collections like lists and dictionaries.
- System.Threading → Multithreading and parallel programming.

- **Core Runtime Libraries:** Provides the **Garbage Collector (GC)**, **JIT Compiler**, and runtime services necessary for executing .NET Core applications.

Microsoft.AspNetCore.App:

Includes all the **ASP.NET Core libraries** required for web development. It provides:

- **Web Server Implementations** → Kestrel for cross-platform hosting and IIS integration for Windows hosting.

- **ASP.NET Core MVC Framework** → For building APIs and web apps using the MVC pattern.
- **Razor Engine** → Used for processing Razor pages and views in dynamic web content.
- **Routing and Model Binding** → Enables mapping of HTTP requests to controllers and automatic data binding.
- **Authentication and Authorization Libraries** → Provides support for securing APIs with JWT, OAuth, Identity, etc.

Packages

The **Packages Section** lists external NuGet packages that are added to extend the project's functionality. They are downloaded from NuGet.org and are automatically restored whenever you build the project on a new machine. Packages can be removed if no longer required, keeping the project lightweight.

Common examples include:

- **Swashbuckle.AspNetCore** – Integrates Swagger UI for interactive API documentation and testing.
- **Entity Framework Core (EF Core)** – Provides ORM (Object Relational Mapping) support for database access.
- **Serilog / NLog** – Advanced logging frameworks for structured and centralized logging.
- **IdentityServer** – A library for implementing authentication, authorization, and token management.

Properties:

The **Properties folder** in an ASP.NET Core project contains files that define how the application behaves during **development and debugging** inside Visual Studio or when running via the .NET CLI. By default, it includes only one file: launchSettings.json.

This **file is not used in production**. Instead, it is only relevant during local development, helping Visual Studio and the dotnet run command determine:

- Which URLs the application should listen on.

- Whether it runs on HTTP, HTTPS, or IIS Express.
- Which environment variables should be set (for example, ASPNETCORE_ENVIRONMENT=Development).
- Whether a browser should open automatically and which URL it should open.

launchSettings.json

The launchSettings.json file contains launch profiles that specify how the application should be started. Each profile is like a separate startup configuration, allowing you to run the same project in different modes (HTTP, HTTPS, or IIS Express).

- It's a configuration file used locally (on a developer's machine) by tools like Visual Studio, dotnet CLI (dotnet run) or other IDEs to decide how to launch the app.
- It is **not** used by the application itself once published or deployed. Production hosting ignores it; production configuration should come from other sources (e.g. appsettings.json, real environment variables).

```
{
  "profiles": {
    "http": {
      "commandName": "Project",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "dotnetRunMessages": true,
      "applicationUrl": "http://localhost:5021"
    },
    "https": {
      "commandName": "Project",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "dotnetRunMessages": true,
      "applicationUrl": "https://localhost:7081;http://localhost:5021"
    },
    "Container (Dockerfile)": {
    }
  }
}
```

```
"commandName": "Docker",
"launchUrl": "{Scheme}://{ServiceHost}:{ServicePort}",
"environmentVariables": {
    "ASPNETCORE_HTTPS_PORTS": "8081",
    "ASPNETCORE_HTTP_PORTS": "8080"
},
"publishAllPorts": true,
"useSSL": true
},
},
"$schema": "https://json.schemastore.org/launchsettings.json"
}
```

Structure of Your JSON & What It Means

Your JSON defines three “profiles” under “profiles” — each profile describes a different way to launch the application. These are useful when you want different behaviors or settings depending on context (e.g. plain HTTP, HTTPS, Docker container).

Here are the profiles:

Profile Name	Purpose / Meaning
"http"	Runs the app over plain HTTP (on localhost), using the project directly (likely via Kestrel server). <code>environmentVariables</code> sets <code>ASPNETCORE_ENVIRONMENT = "Development"</code> , meaning the app runs in “development” mode locally. <code>applicationUrl</code> says it will listen on <code>http://localhost:5021</code> .
"https"	Runs the app over HTTPS (and also HTTP fallback) in development mode. The <code>applicationUrl</code> is <code>https://localhost:7081;http://localhost:5021</code> , meaning the app will listen on both those addresses. This is for testing secure HTTPS connections locally.
"Container (Dockerfile)"	A profile used when running the app inside a container (e.g. Docker). <code>commandName: "Docker"</code> signals tooling to spin up a container rather than directly run the project. The profile defines environment variables (e.g. ports for HTTP and HTTPS inside container), and <code>publishAllPorts: true</code> indicates that ports should be exposed/published. <code>useSSL: true</code> suggests HTTPS should be used.

Other settings:

- `"commandName"` — defines how to start the app. For `"Project"` → run the compiled project directly (e.g. with Kestrel). For `"Docker"` → run via Docker container.
- `"environmentVariables"` — these are variables set for the process when launched via that profile. Commonly used to set the runtime environment (e.g. `"ASPNETCORE_ENVIRONMENT": "Development"`).
- `"applicationUrl"` — the URL(s) (and ports) the application will bind to when run under this profile. Multiple URLs (semicolon-separated) mean the app listens on multiple addresses (e.g. both HTTP and HTTPS).
- `$schema` — a reference to the JSON schema for `launchSettings.json` (for IDE validations). Doesn’t affect runtime logic.

What It's Good For

- Easily switch between different launch modes (http, https, container) depending on what you're testing.
- Automatically set environment variables (like "ASPNETCORE_ENVIRONMENT") so your app can behave differently (for example loading development settings, showing detailed errors, enabling debugging).
- Define which ports/URLs the app uses during development without hard-coding them in code — makes it easier for each developer on the team to run locally without conflicts.
- If using Docker/containers, define container-specific run settings (ports, SSL, publishing) for easy container runs.

Controllers Folder:

The Controllers folder is the core of any ASP.NET Core Web API application. It contains the controller classes, which are responsible for:

- Accepting **incoming HTTP requests** from clients (e.g., browser, Postman, mobile app).
- Executing the necessary **business logic** (either directly or via service/repository layers).
- Returning the appropriate **HTTP response** back to the client (e.g., JSON data, status codes).

Controllers act as the **bridge between clients (such as browsers, mobile apps, or API consumers)** and our application's business logic.

- Each **controller class** typically represents a **resource** or a **set of related endpoints**. For example, you might have a ProductsController to handle product-related APIs or a UsersController for user management.
- Controllers use **routing attributes** (like [Route], [HttpGet], [HttpPost]) to map specific HTTP requests to action methods.

- Each **action method** inside a controller corresponds to an operation that can be performed on the resource, such as:
 - **GET** → Retrieve data
 - **POST** → Insert new data
 - **PUT** → Update existing data
 - **DELETE** → Remove data

Example: WeatherForecastController

By default, when we create a new ASP.NET Core Web API project, Visual Studio adds a **sample controller** named WeatherForecastController. This controller demonstrates the basic structure and functionality of a Web API endpoint.

```
using Microsoft.AspNetCore.Mvc;
namespace MyFirstWebAPIProject.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class WeatherForecastController : ControllerBase
    {
        private static readonly string[] Summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm",
            "Balmy", "Hot", "Sweltering", "Scorching"
        };

        private readonly ILogger<WeatherForecastController> _logger;

        public WeatherForecastController(ILogger<WeatherForecastController> logger)
        {
            _logger = logger;
        }

        [HttpGet(Name = "GetWeatherForecast")]
        public IEnumerable<WeatherForecast> Get()
```

```

    {
        return Enumerable.Range(1, 5).Select(index => new
WeatherForecast
{
    Date =
DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
    TemperatureC = Random.Shared.Next(-20, 55),
    Summary =
Summaries[Random.Shared.Next(Summaries.Length)]
})
    .ToArray();
}
}

```

appsettings.json file:

The **appsettings.json** file acts as the **primary configuration source** for an ASP.NET Core application. It contains structured key-value pairs in **JSON format** and defines how the application behaves at runtime.

- **Database connection strings** (SQL Server, PostgreSQL, etc.).
- **API keys** for third-party services.
- **Logging configuration** (log levels for different parts of the app).
- **Custom configuration values** are defined by the developer.

ASP.NET Core also supports **hierarchical configuration** and **environment-specific overrides**. For example:

- **appsettings.json** → Base/default settings.
- **appsettings.Development.json** → Overrides for development environment.
- **appsettings.Production.json** → Overrides for production deployment.

ASP.NET Core automatically loads the correct configuration based on the **ASPNETCORE_ENVIRONMENT** variable. This layered approach enables developers to have distinct settings for each environment while maintaining a consistent codebase.

Program.cs Class File:

The **Program.cs** file is the **entry point** of every ASP.NET Core application. When the application starts, this is the file that the .NET runtime looks for and executes. It defines the **host configuration, service registration, and the HTTP request pipeline (also known as the middleware pipeline)**. Together, these determine how your API behaves when handling client requests and responses.

In .NET 6 and later, Microsoft introduced a **minimal hosting model**, which simplified the structure of Program.cs by merging Startup.cs into it. This is why you see all configurations (services + middleware) handled in one place.

Key Responsibilities of the Program.cs

The Program.cs file performs four major tasks:

- **Build and configure the ASP.NET Core host** → Sets up the hosting environment (Kestrel web server, configuration, logging, etc.).
- **Register services** → Adds services such as Controllers, Swagger, Authentication, EF Core, and custom services to the dependency injection (DI) container.
- **Define the middleware pipeline** → Configures request/response processing (e.g., HTTPS redirection, authentication, routing, error handling) and also the order in which the middleware components handle requests.
- **Start the application** → Runs the app (Calls app.Run()) so it can begin listening for HTTP requests.

Default Program.cs File

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
```

```
builder.Services.AddOpenApi();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Service Registration (Dependency Injection)

Service registration is the process of adding reusable components (services) to the **Dependency Injection** (DI) container. ASP.NET Core uses built-in DI by default, which makes it easier to manage dependencies. Services are registered inside the `builder.Services`.

- `AddControllers()`
 - Registers **MVC controllers** with the DI container.
 - Enables support for Web API endpoints defined in controller classes.
- `AddOpenApi()`
 - Registers services needed to generate openAPI docs.
 - Maps an endpoint for viewing the OpenAPI document in JSON format with the `MapOpenApi()` extension method on the app.

Middleware Configuration

The app object defines the **Middleware Pipeline**, a sequence of components that process every incoming HTTP request and outgoing response. Each piece of middleware can inspect, modify, or short-circuit the request/response before passing it to the next component. The order of middleware calls is very important.

- `if (app.Environment.IsDevelopment())`
 - Checks if the current environment is Development.
- `app.UseAuthorization()`
 - Adds authorization middleware to enforce security policies.
 - Works in conjunction with authentication middleware to restrict access to endpoints.
- `app.MapControllers()`
 - Maps controller routes to the request pipeline.
 - Without this line, controller endpoints will not be accessible.
- `app.Run()`
 - Starts the application and begins listening for incoming HTTP requests.

MyProjectName.http file

The **.http file** in an ASP.NET Core Web API project is a **built-in testing file** that allows you to send HTTP requests directly from within **Visual Studio** or **Visual Studio Code**. Instead of relying on external tools like **Postman** or **Fiddler**, or command-line tools such as **cURL**, you can quickly write, send, and debug API requests directly inside your IDE.

This file is part of the Visual Studio HTTP Client feature and is automatically generated when you create a new Web API project.

When you open the MyProjectName.http file for the first time, you will typically see

```
@MyFirstWebAPIProject_HostAddress = http://localhost:5021
```

```
GET {{@MyFirstWebAPIProject_HostAddress}}/weatherforecast/  
Accept: application/json
```

```
###
```

- **@Blogging_Web_API_HostAddress** → A variable holding the base URL of your API (localhost + port).
- **GET {}/weatherforecast/** → Defines a sample GET request to the WeatherForecast endpoint.
- **Accept: application/json** → Adds a request header that tells the server to return the response in JSON format.

How to Use To test an API request using the .http file:

- **Run the project** → Start your API using the HTTP launch profile in Visual Studio.
- **Ensure the port matches** → The port number in MyFirstWebAPIProject.http must match the one in launchSettings.json. If your app runs on http://localhost:5021, make sure the variable points to the same URL.
- **Send request inside IDE** → Open the .http file and hover over the request. Visual Studio will show a “Send Request” link above it.
- **View the response** → After clicking “Send Request,” the response (status code, headers, body) will appear in a new results pane or side window.

```
@MyFirstWebAPIProject_HostAddress = http://localhost:5222
    ↓ Click Here to Test this API
    × Send request | Debug
| GET {{MyFirstWebAPIProject_HostAddress}}/weatherforecast/
| Accept: application/json
|
###
```

image_253.png

Once you click on the Send Request link, the responses are typically shown in a pane next to the request or in a separate window, allowing you to view the status code, response body, and headers, as shown in the image below.

The screenshot shows the .NET HTTP Test Client interface. On the left, there's a code editor window with the following content:

```
@MyFirstWebAPIProject_HostAddress:  
✓ Send request | Debug  
]GET {{MyFirstWebAPIProject_HostAddress}}  
Accept: application/json  
|  
###
```

To the right, the status bar indicates "Status: 200 OK Time: 29.05 ms Size: 389 bytes". Below the status bar, the title "Formatted Raw Headers Request" is underlined in blue. Underneath it, the word "Body" is followed by "application/json; charset=utf-8, 389 bytes". The response body is a JSON array:

```
[  
  {  
    "date": "2024-02-01",  
    "temperatureC": -10,  
    "temperatureF": 15,  
    "summary": "Chilly"  
  },  
  {  
    "date": "2024-02-02",  
    "temperatureC": -9,  
    "temperatureF": 16,  
    "summary": "Scorching"  
  },  
  {  
    "date": "2024-02-03",  
    "temperatureC": 1,  
    "temperatureF": 33,  
    "summary": "Balmy"  
  }]
```

image_254.png

The **.http** file is a developer-friendly tool built into Visual Studio that allows you to **send and test API requests directly inside your IDE**. It simplifies debugging, supports multiple HTTP methods, and provides quick access to responses without relying on external tools.

Controllers in ASP.NET Core Web API

In **ASP.NET Core Web API**, controllers are **special classes** that handle HTTP requests sent by clients (such as browsers, mobile apps, Postman, or other services) and return HTTP responses. Think of them as the “ **Traffic Police**” or **Gatekeepers** of your API. They listen to requests at specific endpoints (URLs), process them with the aid of business logic, and return meaningful responses in standard web formats, such as JSON or XML.

In MVC terms:

- **Model** = Data (Entities, DTOs, Domain Models).
- **View** = Not used in Web APIs (unlike MVC web apps).
- **Controller** = The central unit that receives the request, coordinates with services and models, and sends back a response.

Key Roles of a Controller

Let us understand the key Roles and Responsibilities of a Controller in an ASP.NET Core Web API Application.

Request Handling

The controller listens at a specific URL route (e.g., /api/products) and decides which **Action Method** (a public method inside the controller) should handle the request. Each action method is decorated with attributes such as `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, `[HttpDelete]` that map to HTTP verbs.

Example: Here, `GET /api/products/5` will be handled by the `GetProduct` method.

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
```

```
{  
    // logic to fetch product  
    return Ok(new { Id = id, Name = "Laptop" });  
}  
}
```

⚠️ Analogy: Think of a call center where incoming calls (requests) are routed to the correct department (action method).

Data Processing

Controllers **shouldn't hold heavy business logic themselves**. Instead, they delegate:

- To **services** for applying business rules (e.g., price discounts, tax calculation).
- To **repositories** or EF Core for data persistence (CRUD on a database).
- To **validators** for input checking (e.g., ensuring an email format is valid).

They handle input validation, model binding, and error checking before passing data to the services. For example:

```
[HttpPost]  
public IActionResult CreateProduct(ProductDto dto)  
{  
    if (!ModelState.IsValid)  
        return BadRequest(ModelState);  
  
    var product = _productService.Create(dto);  
    return CreatedAtAction(nameof(GetProduct), new { id = product.Id },  
product);  
}
```

⚠️ Analogy: The controller is like a coordinator. It doesn't do the heavy work but ensures the right team (services, database) gets involved.

Response Generation

After the data is processed, the controller decides **what response to send back**.

Responses may include:

- **Response Payload** → Data in JSON/XML format (default is JSON in Web API).
- **Status codes** → To indicate success or failure (200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server Error, etc.).
- **Error messages** → Clear explanation if something went wrong.

Example:

```
[HttpDelete("{id}")]
public IActionResult DeleteProduct(int id)
{
    bool deleted = _productService.Delete(id);
    if (!deleted)
        return NotFound(new { Message = "Product not found" });

    return NoContent(); // 204
}
```



Analogy: Like a waiter in a restaurant, after checking with the chef (services/database), the waiter (controller) delivers the order (response) back to the customer (client).

Adding Controller Class in ASP.NET Core Web API

When creating a controller in an ASP.NET Core Web API project, adhere to the following conventions and best practices:

Naming Convention:

ASP.NET Core uses **Conventions** for recognizing controllers. By convention, ASP.NET Core recognizes controller classes whose names end with the “**Controller**” suffix.

- **Must end with Controller suffix** → This is how ASP.NET Core identifies them during routing.
- Examples:
 - EmployeeController → Manages employees.
 - StudentController → Manages students.
 - ProductController → Manages products.

If you name your class Employee instead of EmployeeController, ASP.NET Core won't recognize it as a controller.

⚠ Best Practice: Choose meaningful names based on the resource you're exposing, not on verbs.

- Correct Example: EmployeeController
- Wrong Example: ManageEmployeeController or DoEmployeeStuffController

Base Class

Controllers in Web API projects usually inherit from ControllerBase.

- **ControllerBase** → Provides features needed for APIs (No views, just JSON/XML responses).
- **Controller** → Used in **MVC applications** where you need both API and Razor Views (it inherits from ControllerBase and adds view-related features like View(), PartialView(), etc.).

In Web API, stick with ControllerBase unless you are mixing APIs with Razor views.

```
[ApiController]
[Route("api/[controller]")]
public class EmployeeController : ControllerBase
{
```

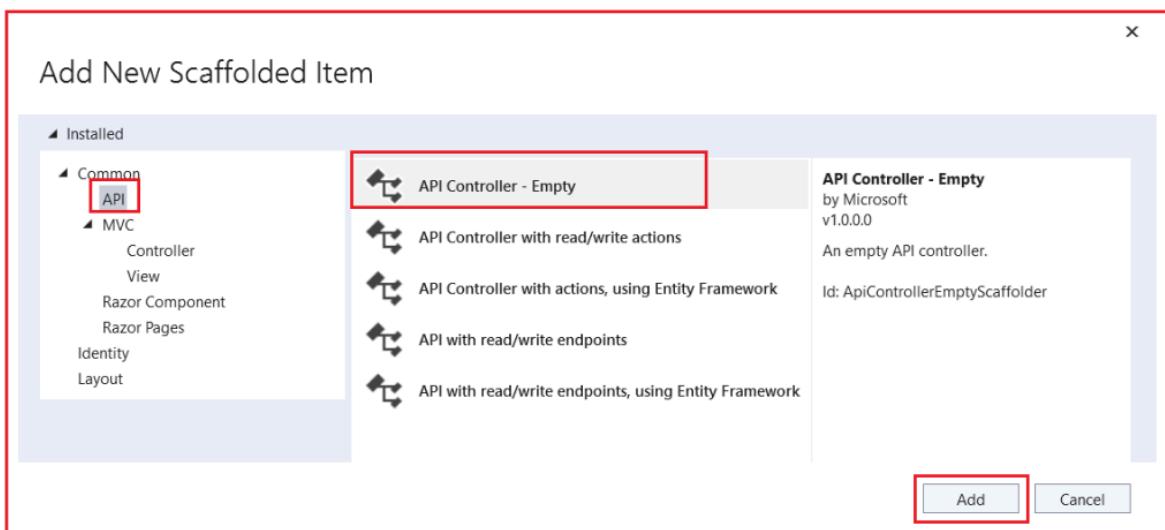
```

[HttpGet]
public IActionResult GetEmployees()
{
    return Ok(new[] { "John", "Jane", "Alice" });
}

```

Steps to Add a Controller:

Let's add a controller named Employee. To do so, right-click on the Controllers folder and then select Add => Controller from the context menu, which will open the following Add New Scaffolded Item window. From this window, please choose API, then API Controller – Empty, and click on the Add button, as shown in the image below.



image_255.png

Understanding the Different Options:

API Controller – Empty

This option provides a blank controller with no predefined actions, allowing you to build everything from scratch. It's ideal for learning, experimenting, and situations where you need maximum flexibility to design custom endpoints and logic without code.

- Contains only class declaration with [ApiController] and [Route].
- No predefined CRUD methods are available; you must add all endpoints manually.

- Best suited for beginners and projects that require full customization.
- Helps in understanding the fundamentals of Routing, HTTP Verbs, and Responses.



Analogy: Like moving into an empty room where you have complete freedom to decide the layout, furniture, and design.

API Controller with Read/Write Actions

This option scaffolds a controller with predefined CRUD (Create, Read, Update, Delete) methods, making it easy to set up a RESTful API quickly. It's ideal for projects where you want a head start and don't want to create standard CRUD endpoints manually.

- Automatically generates methods for Get, Get by ID, Post, Put, and Delete.
- Suitable for quick prototypes or standard APIs that do not require complex logic.
- Reduces repetitive coding and provides a working structure instantly.
- Can be extended later by adding business logic or custom validations.



Analogy: Like moving into a house with basic furniture already in place, you can start living immediately, but you can still customize it later.

API Controller with Actions Using Entity Framework

This option extends CRUD scaffolding by connecting directly to a database using Entity Framework Core. It's perfect when you already have models and a DbContext and want to generate database-ready API endpoints quickly.

- Prebuilt methods that query and update data directly from the database.
- Integrates Entity Framework Core for database interactions.
- Useful for rapid prototyping of database-backed applications.
- Reduces duplicate code for common database operations.

API with Read/Write Endpoints

This option is similar to the Read/Write Actions template but provides more control over endpoint naming and structure. It's meant for developers who need flexibility in designing endpoints while still benefiting from predefined CRUD scaffolding.

Generates standard CRUD endpoints but allows custom route names. Useful when API design requires adherence to specific naming conventions. Balances automation with customization for endpoint structure. Suitable for teams following unique API standards.

API Controller with Read/Write Endpoints Using Entity Framework

This option combines the benefits of Entity Framework integration with customizable endpoints, providing a flexible solution. It gives you ready-to-use database CRUD operations while allowing you to rename or extend endpoints beyond the defaults.

- Direct database integration using Entity Framework Core.
- Allows renaming and customizing routes for more flexibility.
- Ideal for production-ready apps requiring both EF and non-standard endpoint names.
- Saves time by scaffolding EF code while supporting customization.

Models in ASP.NET Core Web API

In ASP.NET Core Web API, **Models** are C# Classes that define the shape and structure of the data your application works with. They describe how data is structured, how it relates to other data, and how it should be validated before being stored or returned. Models act as the **blueprint** for both database entities and data transfer between the client and server.

Key Features of Models:

The following are the Key Features of Models in ASP.NET Core Web API:

Data Representation (POCOs)

Models are typically POCOs (Plain Old CLR Objects), simple classes that do not depend on any base class or framework inheritance. They define properties that reflect the data structure of your application or API.

```
namespace MyFirstWebAPIProject.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; } = null!;
        public decimal Price { get; set; }
        public string Category { get; set; } = null!;
    }
}
```

⚠ These models are used by EF Core (if enabled) to map to database tables.

Data Validation with Data Annotations

Models can be annotated with attributes from the

System.ComponentModel.DataAnnotations namespace to apply validation rules. These rules validate incoming data automatically before it reaches business logic.

Common Data Annotations:

- [Required] → ensures a property is not null or empty.
- [StringLength(50)] → restricts text length.
- [Range(1, 100)] → enforces numeric ranges.
- [RegularExpression] → checks formats (e.g., email, phone numbers).

This reduces bugs and ensures the integrity of incoming data. For example:

```
using System.ComponentModel.DataAnnotations;
public class Product
{
    public int Id { get; set; }

    [Required]
    [StringLength(100, MinimumLength = 3)]
    public string Name { get; set; } = null!;

    [Range(1, 10000.00)]
    public decimal Price { get; set; }

    [Required]
    public string Category { get; set; } = null!;
}
```

Defining Relationships (EF Core Models)

When used with Entity Framework Core, models can define relationships such as:

This is done using **Navigation Properties** and **foreign key conventions**. For example, A Product belongs to one Category, while a Category can have many Products (one-to-many).

- One-to-Many
- One-to-One

- Many-to-Many

```
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }

    // One-to-Many
    public List<Product> Products { get; set; }
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }

    public int CategoryId { get; set; } // FK
    public Category Category { get; set; }
}
```

Navigation properties (Category Category, List<Product> Products) allow EF Core to manage relationships and enable LINQ queries across related data. EF Core uses these relationships to generate the correct schema and handle joins.

DTOs (Data Transfer Objects)

DTOs are not Entity Models — they are **Lightweight Classes** designed to:

- Models can also act as **DTOs**, which are specialized objects for shaping the data exposed by the API.
- Hide sensitive/internal fields (like passwords or internal IDs) and only return what the client actually needs.
- DTOs improve **security, performance, and clarity** of API responses by sending only required data.

Example:

```
public class ProductDTO
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```



Note: You typically need to use AutoMapper or manual mapping to convert between Entity Models and DTOs

Entity Framework Core Integration

When working with EF Core:

- Models map to database **tables**.
- Properties map to **columns**.
- Annotations (such as [Key] and [ForeignKey]) control schema behavior.

EF Core uses these models to generate SQL commands for **CRUD** (Create, Read, Update, Delete) operations.

```
public class ApplicationDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
}
```

EF Core will map Product to a Products table and generate SQL queries like `SELECT * FROM Products`.

[Workshop] Controllers & Models : CRUD without dbs

Analogy

Think of **Models** as the **blueprint of a house**:

- They define what rooms (properties) exist, how rooms connect (relationships), and the rules for construction (validation, i.e., size, safety rules).
- Entity Framework is like the Builder who uses this blueprint to construct the actual house (database tables).
- DTOs are like sales brochures; you don't show the wiring and plumbing (sensitive data), only what the customer needs to see.

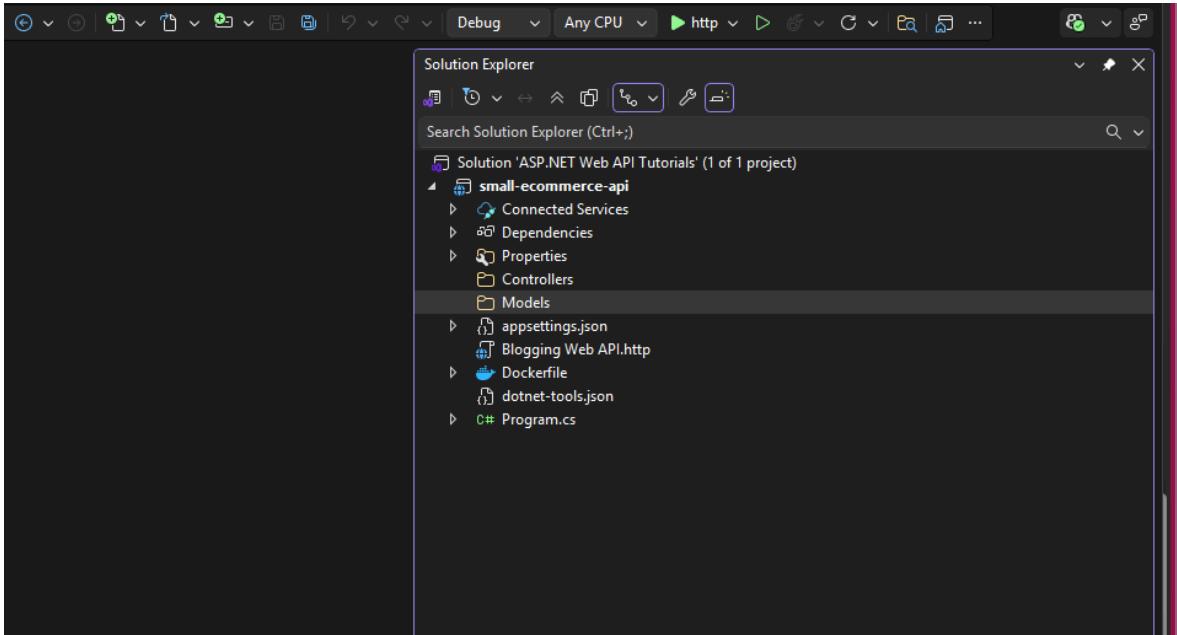
Creating Models in ASP.NET Core Web API

In ASP.NET Core Web API, Models represent the structure of your application's data. Models can technically be placed anywhere, but for clarity and maintainability, you can create models:

- In a Dedicated Models folder within the project (recommended for clarity).
- In any Folder/Subfolder of the project (common in small projects).

Folder Creation

Create a ASP.NET Core Web API named `small_ecommerce_api`, then create a model folder inside our project,



image_256.png

Product Model (Entity)

Now, let us add a basic model to represent the Product information. Right-click on the **Models** folder, add a class file named **Product.cs**, and copy and paste the following code. This is a simple model representing a product. It contains properties like **ID**, **Name**, **Price**, and **Category**. It includes validation attributes and a relationship with a Category entity.

```
using System.ComponentModel.DataAnnotations;

namespace small_ecommerce_api.Models
{
    public class Product
    {
        public int Id { get; set; }
        [Required]
        [StringLength(100, MinimumLength = 3)]
        public string Name { get; set; } = null!;
        [Range(0.01, 10000.00)]
        public decimal Price { get; set; }
        // Foreign Key for Category
        public int CategoryId { get; set; }
        // Navigation property (Relationship)
```

```
    public Category Category { get; set; } = null!;  
}  
}
```

Explanation:

- CategoryId: Foreign key that links each Product to a Category.
- Category: navigation property → EF Core uses it to establish the relationship.

Category Model (Entity)

We also create a **Category** entity to define a **one-to-many relationship** (one Category can have many Products). So, create a class file named **Category.cs** within the Models folder and copy-paste the following code.

```
using System.ComponentModel.DataAnnotations;  
  
namespace small_ecommerce_api.Models  
{  
    public class Category  
    {  
        public int Id { get; set; }  
        [Required]  
        [StringLength(50)]  
        public string Name { get; set; } = null!;  
        // Navigation property (One-to-Many relationship)  
        public ICollection<Product>? Products { get; set; }  
    }  
}
```

Explanation: The Products collection represents the one-to-many relationship (1 Category → many Products).

What are DTOs?

DTOs (**Data Transfer Objects**) are simple C# classes used to transfer data between the client and server in an ASP.NET Core Web API. They act as a **contract** that defines exactly

what data should be sent or received, without exposing the full database entities.

Do We Need Different DTOs for Create, Update, and Retrieve Operations?

Yes, it is **highly recommended** to create separate DTOs (Data Transfer Objects) for different CRUD operations, especially for Create (POST), Update (PUT/PATCH), and Retrieve (GET). This design pattern ensures clarity, maintainability, and validation precision, aligning with the Single Responsibility Principle.

Each operation has different requirements in terms of the data it expects or returns. Using dedicated DTOs helps you avoid exposing internal or sensitive fields, simplifies validation logic, and makes your API more robust and scalable.

Create DTO (for POST operations)

A **Create DTO** is used when a new resource needs to be added to the system. Since the database usually generates the ID, this DTO contains only the required properties to create a record. It ensures that only the necessary fields are submitted by the client and applies validation rules to prevent insufficient data from being inserted.

- Excludes the Id property (auto-generated by the database).
- Contains required fields such as Name, Price, and Category ID.
- Enforces validation rules using data annotations.
- Prevents over-posting by limiting input only to fields needed for creation.

Update DTO (for PUT/PATCH operations)

An **Update DTO** is used when modifying an existing resource. Unlike the Create DTO, it includes the Id so the API knows which record to update. It may also allow partial updates (with PATCH) or require all fields (with PUT). This separation ensures that updates are intentional and validated.

- Includes the Id property to identify the resource.
- Contains fields that can be updated, such as Name, Price, and CategoryId.
- Can support full updates (PUT) or partial updates (PATCH).
- Ensures only valid data is sent for modifications.

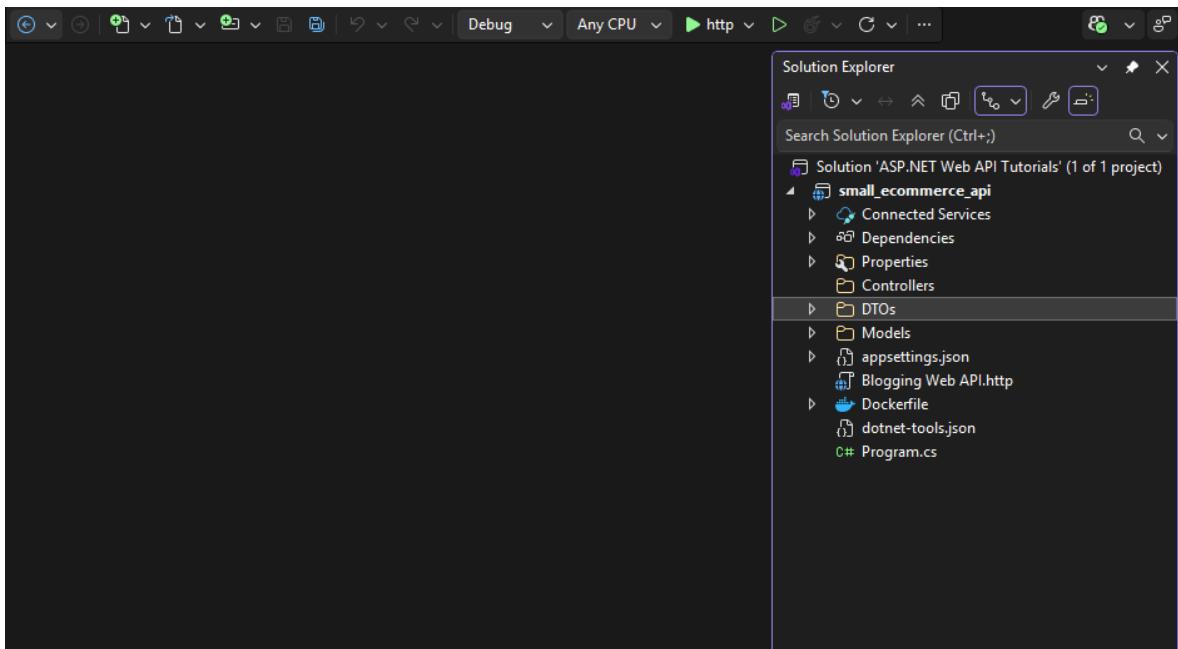
Retrieve DTO (for GET operations)

A **Retrieve DTO** is used to shape the data returned to the client. Instead of exposing the full entity with all fields (including sensitive or unnecessary ones), this DTO returns only what the client should see. It can also transform data for better readability, such as showing CategoryName instead of CategoryId.

- Includes the ID and client-friendly fields.
- Excludes sensitive or internal properties (e.g., cost price, audit fields).
- Can flatten relationships (e.g., return CategoryName instead of a complete Category object).
- Provides a clean and optimized data structure for API responses.
- First, create a folder named ‘DTOs’ in the project root directory, where we will store all our DTOs.

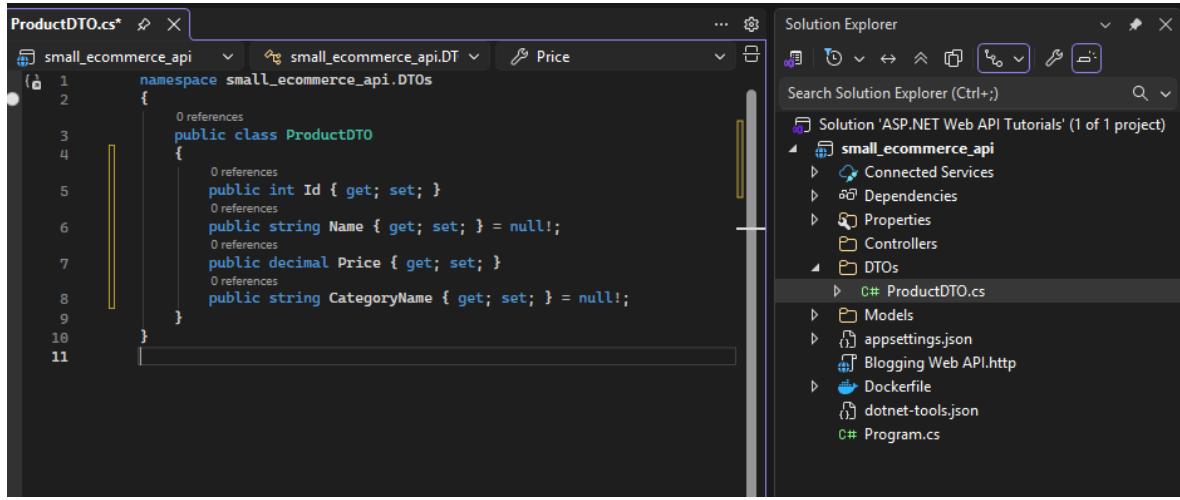
Creating DTOs

Create a folder named ‘DTOs’ in the project root directory, where we will store all our DTOs.



image_257.png

Creating ProductDTO



image_258.png

This DTO is used for returning product details. It excludes sensitive information and only exposes the properties required by the client. Create a class file named ProductDTO.cs within the DTOs folder.

```
namespace small_ecommerce_api.DTOs
{
    public class ProductDTO
    {
        public int Id { get; set; }
        public string Name { get; set; } = null!;
        public decimal Price { get; set; }
        public string CategoryName { get; set; } = null!;
    }
}
```

Explanation:

- ProductDTO **includes only the necessary fields** for API responses.
- Instead of sending the complete Category entity, it exposes CategoryName.
- This ensures that sensitive or unnecessary data isn't exposed to the client.

ProductCreateDTO

This DTO is used when creating a new product. It **excludes the Id** (since the database will generate it) and only includes the properties needed to create a product. Create a class file named **ProductCreateDTO.cs** within the **DTOs** folder.

```
using System.ComponentModel.DataAnnotations;

namespace small_ecommerce_api.DTOs
{
    public class ProductCreateDTO
    {
        [Required(ErrorMessage = "Product name is required.")]
        [StringLength(100, MinimumLength = 3, ErrorMessage = "Product
name must be between 3 and 100 characters.")]
        public string Name { get; set; } = null!;

        [Range(0.01, 10000.00, ErrorMessage = "Price must be between
0.01 and 10,000.")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "CategoryId is required.")]
        public int CategoryId { get; set; }
    }
}
```

Explanation:

- Excludes Id since it is auto-generated.
- Includes essential fields required to create a product (Name, Price, CategoryId).
- CategoryId ensures the new product is linked to a valid category.
- Keeps the DTO **lightweight and focused** only on creation.

ProductUpdateDTO

This DTO is used when updating an existing product. It **includes the ID** so the API knows which record to update. Create a class file named **ProductUpdateDTO.cs** within the **DTOs** folder and then copy and paste the following code.

```

using System.ComponentModel.DataAnnotations;

namespace small_ecommerce_api.DTOs
{
    public class ProductUpdateDTO
    {
        [Required(ErrorMessage = "Product Id is required.")]
        public int Id { get; set; }

        [Required(ErrorMessage = "Product name is required.")]
        [StringLength(100, MinimumLength = 3, ErrorMessage = "Product
name must be between 3 and 100 characters.")]
        public string Name { get; set; } = null!;

        [Range(0.01, 10000.00, ErrorMessage = "Price must be between
0.01 and 10,000.")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "CategoryId is required.")]
        public int CategoryId { get; set; }
    }
}

```

Explanation:

- Includes ID to identify the record being updated.
- Allows modification of product details (Name, Price, CategoryId).
- Suitable for both **PUT (full update)** and **PATCH (partial update with adjustments)**.
- Keeps update logic clear and separate from creation and retrieval.

Using Models and DTOs in Controller:

In ASP.NET Core Web API, **Controllers** handle HTTP requests and responses, and **Models** represent the data being exchanged. When combined with **DTOs**, this creates a clean and secure data flow:

- **GET** → Retrieve model/DTO data and return it to the client.

- **POST** → Accept model/DTO data from the request body to create new records.
- **PUT** → Accept model/DTO data for updating existing records.
- **DELETE** → Accept an identifier to delete the correct record.

By introducing **DTOs**, we separate internal entity representation (database structure) from client-facing data contracts, ensuring security and flexibility.

Creating Products Controller

Let's create a Products Controller to manage Product entities. Right-click on the Controllers folder, select **Add > Controller**, choose **API Controller – Empty**, name it **ProductsController.cs**

```
using Microsoft.AspNetCore.Mvc;
using small_ecommerce_api.DTOs;
using small_ecommerce_api.Models;

namespace small_ecommerce_api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : Controller
    {
        // Hardcoded Categories
        private static List<Category> _categories = new List<Category>
        {
            new Category { Id = 1, Name = "Electronics" },
            new Category { Id = 2, Name = "Furniture" },
        };
        // Hardcoded Products
        private static List<Product> _products = new List<Product>
        {
            new Product { Id = 1, Name = "Laptop", Price = 1000.00m,
CategoryId = 1 },
            new Product { Id = 2, Name = "Desktop", Price = 2000.00m,
CategoryId = 1 },
            new Product { Id = 3, Name = "Chair", Price = 150.00m,
CategoryId = 2 }
        };
    }
}
```

```

        CategoryId = 2 },
    };

    // GET: api/products
    [HttpGet]
    public ActionResult<IEnumerable<ProductDTO>> GetProducts()
    {
        var productDTOs = _products.Select(p => new ProductDTO
        {
            Id = p.Id,
            Name = p.Name,
            Price = p.Price,
            CategoryName = _categories.FirstOrDefault(c => c.Id == p.CategoryId)?.Name ?? "Unknown"
        }).ToList();

        return Ok(productDTOs);
    }

    // GET: api/products/{id}
    [HttpGet("{id}")]
    public ActionResult<ProductDTO> GetProduct(int id)
    {
        var product = _products.FirstOrDefault(p => p.Id == id);
        if (product == null)
        {
            return NotFound(new { Message = $"Product with ID {id} not found." });
        }

        var productDTO = new ProductDTO
        {
            Id = product.Id,
            Name = product.Name,
            Price = product.Price,
            CategoryName = _categories.FirstOrDefault(c => c.Id == product.CategoryId)?.Name ?? "Unknown"
        };
        return Ok(productDTO);
    }
}

```

```

    }

    // POST: api/products
    [HttpPost]
    public ActionResult<ProductDTO> PostProduct([FromBody]
ProductCreateDTO createDto)
    {
        var newProduct = new Product
        {
            Id = _products.Max(p => p.Id) + 1,
            Name = createDto.Name,
            Price = createDto.Price,
            CategoryId = createDto.CategoryId
        };
        _products.Add(newProduct);
        var productDTO = new ProductDTO
        {
            Id = newProduct.Id,
            Name = newProduct.Name,
            Price = newProduct.Price,
            CategoryName = _categories.FirstOrDefault(c => c.Id == newProduct.CategoryId)?.Name ?? "Unknown"
        };
        return CreatedAtAction(nameof(GetProduct), new { id = productDTO.Id }, productDTO);
    }

    // PUT: api/products/{id}
    [HttpPut("{id}")]
    public IActionResult UpdateProduct(int id, [FromBody]
ProductUpdateDTO updateDto)
    {
        if (id != updateDto.Id)
        {
            return BadRequest(new { Message = "ID mismatch between route and body." });
        }
        var existingProduct = _products.FirstOrDefault(p => p.Id ==

```

```

        id);
        if (existingProduct == null)
        {
            return NotFound(new { Message = $"Product with ID {id} not found." });
        }
        // Update product
        existingProduct.Name = updateDto.Name;
        existingProduct.Price = updateDto.Price;
        existingProduct.CategoryId = updateDto.CategoryId;
        return NoContent();
    }

    // DELETE: api/products/{id}
    [HttpDelete("{id}")]
    public IActionResult DeleteProduct(int id)
    {
        var product = _products.FirstOrDefault(p => p.Id == id);
        if (product == null)
        {
            return NotFound(new { Message = $"Product with ID {id} not found." });
        }
        _products.Remove(product);

        return NoContent();
    }

}
}

```

Understanding Action Methods:

Let us understand the use of each action method:

GetProducts() – Retrieve All Products

Handles GET /api/products requests.

- Retrieves the full list of products from the in-memory _products collection.
- Maps each Product entity into a ProductDTO (so only safe, necessary fields are exposed).
- Returns the list inside an HTTP 200 OK response.
- Use case: Used by clients to display a product catalog.

GetProduct(int id) – Retrieve a Single Product by ID

Handles GET /api/products/ requests.

- Searches _products for a product with the given id.
- If not found → returns 404 Not Found with a descriptive message.
- If found → maps it to a ProductDTO and returns 200 OK.
- Use case: Used by clients to view details of a specific product (e.g., clicking on a product card).

PostProduct(ProductCreateDTO createDto) – Create a New Product

Handles POST /api/products requests.

- Accepts a ProductCreateDTO from the request body.
- Creates a new Product entity and assigns a new ID (simulating database auto-increment).
- Adds the new product to the _products array.
- Maps the new entity to a ProductDTO for the response.
- Returns 201 Created with the new product's details and a Location header pointing to the new resource.
- Use case: Used by clients to add a new product (e.g., admin adds a new item to

inventory).

UpdateProduct(int id, ProductUpdateDTO updateDto) – Update an Existing Product

Handles PUT /api/products/ requests.

- Checks that the ID in the route matches the updateDto.Id in the request body (to prevent mismatches).
- Searches for the existing product.
- If not found → returns 404 Not Found.
- If found → updates the product's properties (Name, Price, CategoryId).
- Returns 204 No Content (success but no response body).
- Use case: Used by clients/admins to update an existing product's details (e.g., change price or category).

DeleteProduct(int id) – Remove a Product

Handles DELETE /api/products/ requests.

- Looks up the product by id.
- If not found → returns 404 Not Found.
- If found → removes it from _products.
- Returns 204 No Content (success with no body).
- Use case: Used by admins to remove discontinued products from the catalog.

Testing the Products Controller APIs using .HTTP File:

You can test the APIs in various ways, such as using **Postman**, **Fiddler**, and **Swagger**. However, .NET 8 provides the .http file, and by using that file, we can also test the functionality.

What is the .http File in .NET?

- When you create an ASP.NET Core Web API project in Visual Studio, it automatically generates a .http file.
- This .http file name will be the same name as your project name. My Project name is MyFirstWebAPIProject, so Visual Studio creates the MyFirstWebAPIProject.http file.
- This file contains sample HTTP requests that you can run directly inside the IDE.
- Each request block starts with ###, and you will see a Send Request link above it to trigger the request.
- The IDE sends the request to your running API and displays the response inline, eliminating the need to switch to Postman.

Modifying MyFirstWebAPIProject.http file

Please open the MyFirstWebAPIProject.http file and copy and paste the following code. Please change the port number with the port number on which your application is running.

```
@MyFirstWebAPIProject_HostAddress = https://localhost:7191

### Get All Products
GET {{MyFirstWebAPIProject_HostAddress}}/api/products
Accept: application/json
###

### Get Product with ID 1
GET {{MyFirstWebAPIProject_HostAddress}}/api/products/1
Accept: application/json
###

### Create a New Product (uses ProductCreateDTO)
POST {{MyFirstWebAPIProject_HostAddress}}/api/products
Content-Type: application/json
Accept: application/json

{
    "name": "New Product",
}
```

```

    "price": 39.99,
    "categoryId": 1
}
###

### Update Product with ID 1 (uses ProductUpdateDTO)
PUT {{MyFirstWebAPIProject_HostAddress}}/api/products/1
Content-Type: application/json
Accept: application/json

{
    "id": 1,
    "name": "Updated Product",
    "price": 49.99,
    "categoryId": 1
}
###

### Delete Product with ID 1
DELETE {{MyFirstWebAPIProject_HostAddress}}/api/products/1
Accept: application/json
###

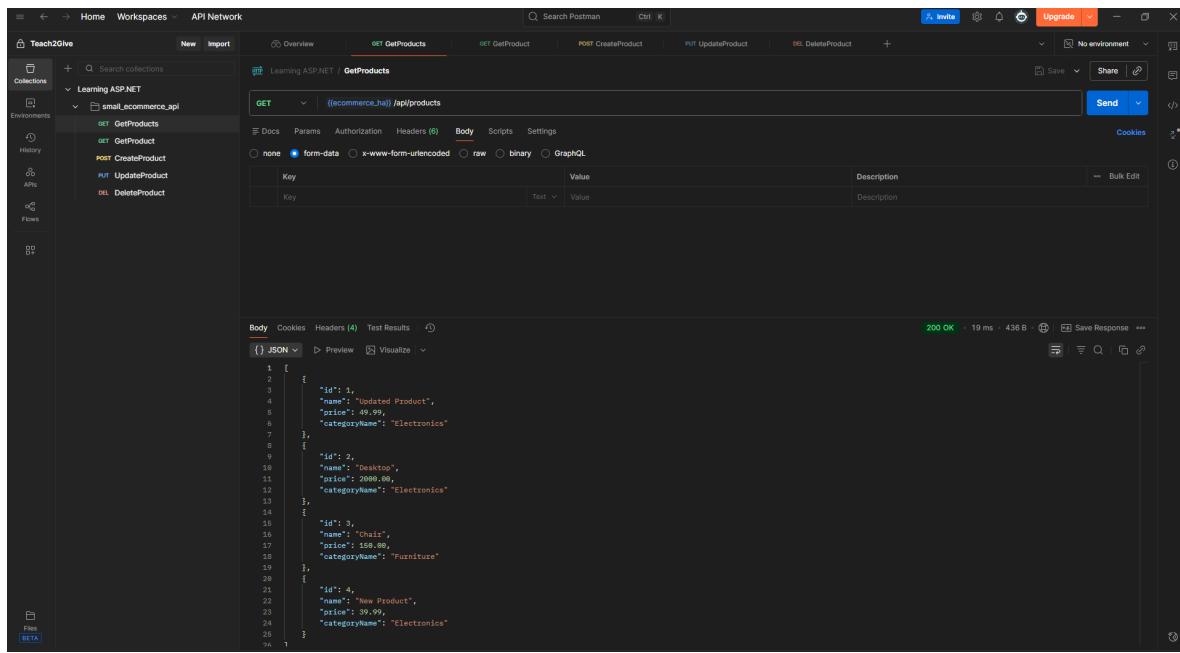
```

Testing:

Before testing, ensure your API runs locally and listens on the correct port. Open the .http file in your IDE, and you should see “**Send Request**” links above each HTTP request. Click these links to execute the requests, and you will see the responses directly in your IDE, as shown in the image below.

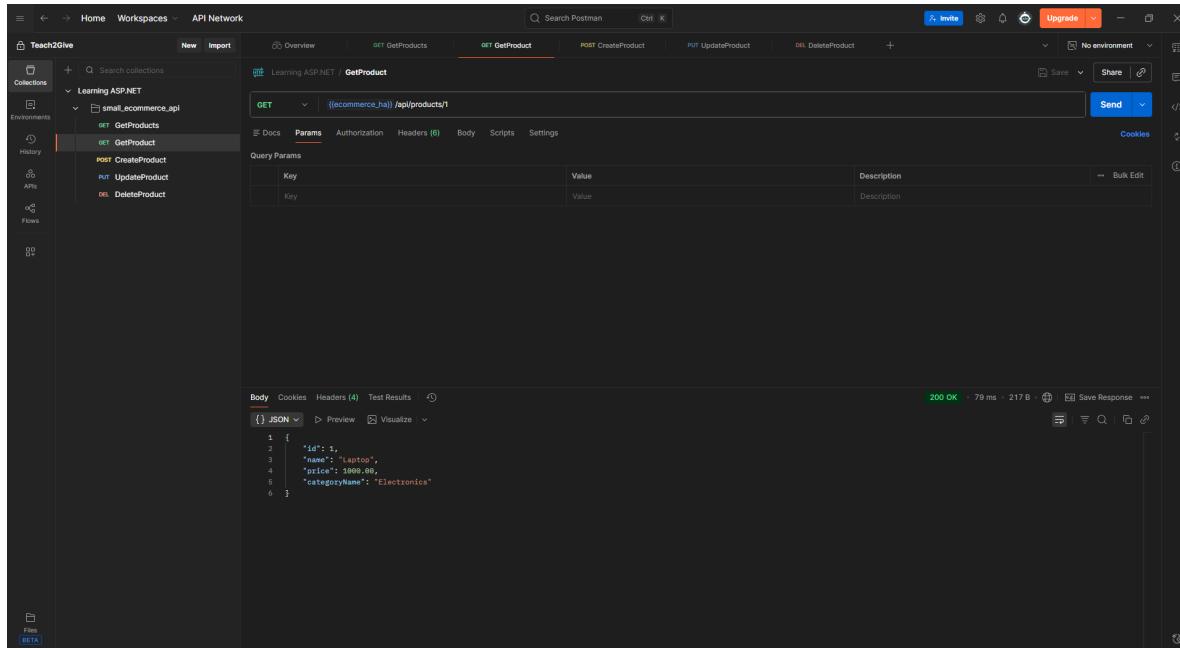
Using Postman

- Get Products



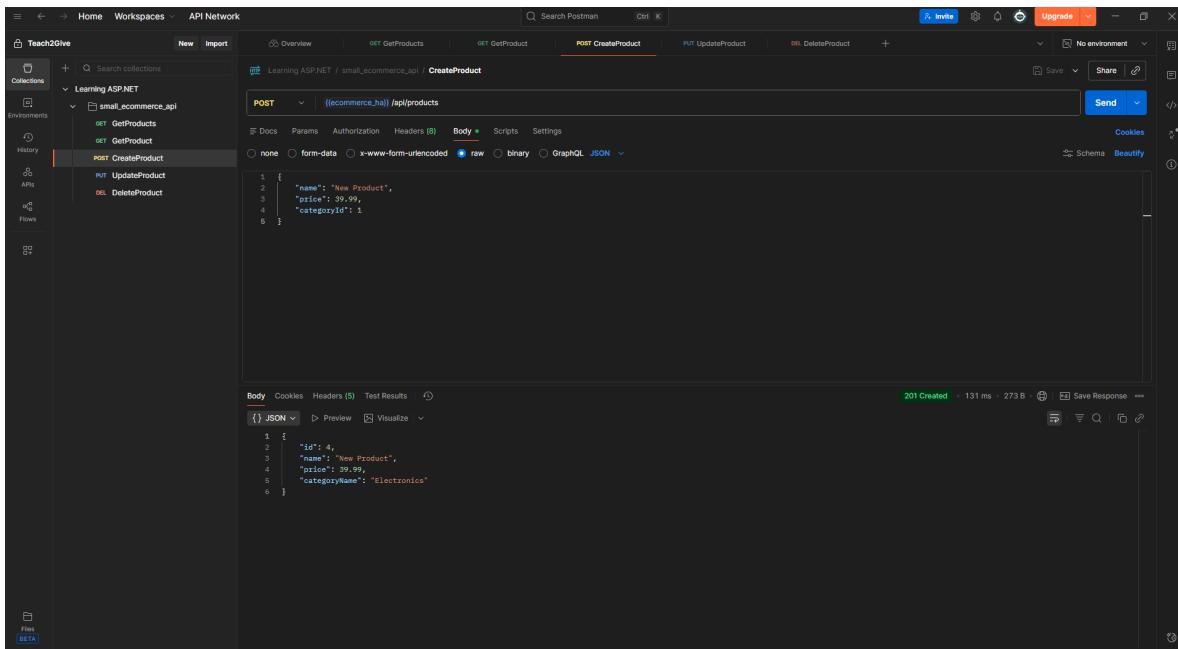
image_259.png

- Get a Product



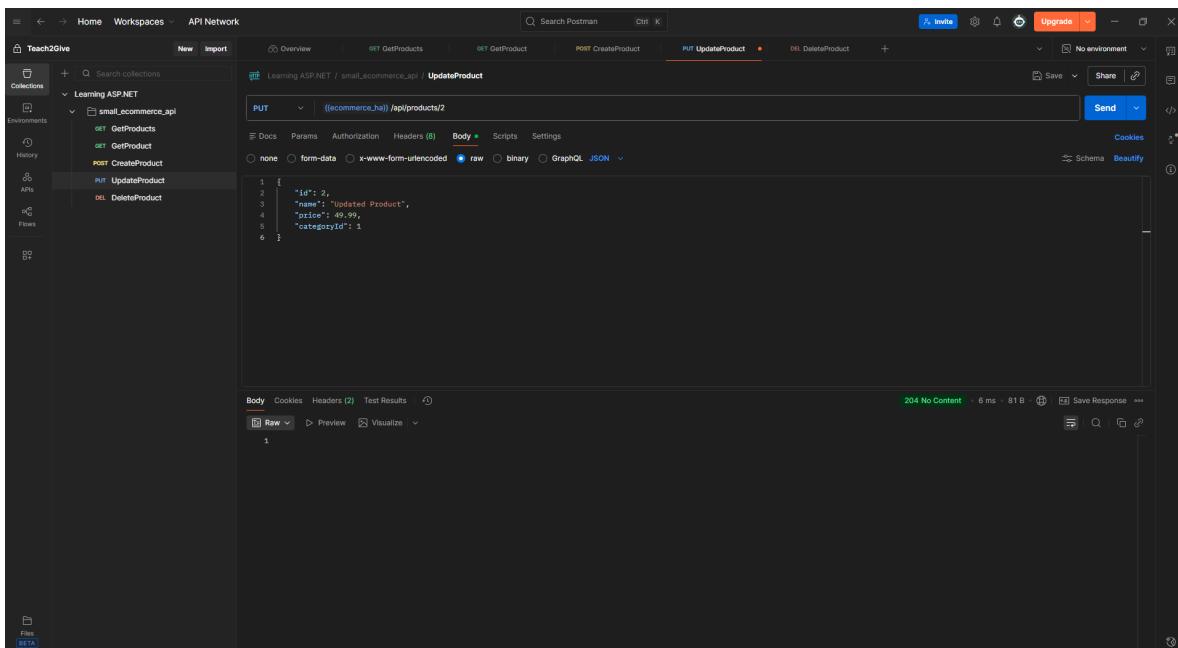
image_260.png

- Create a product



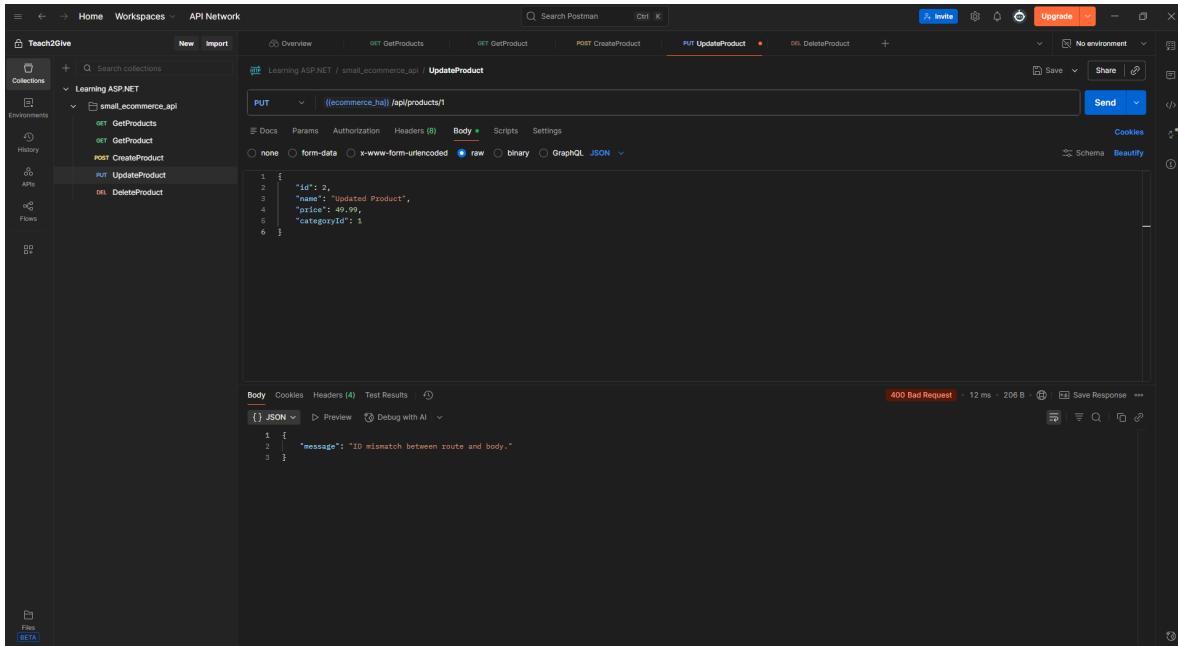
image_261.png

- Update a product



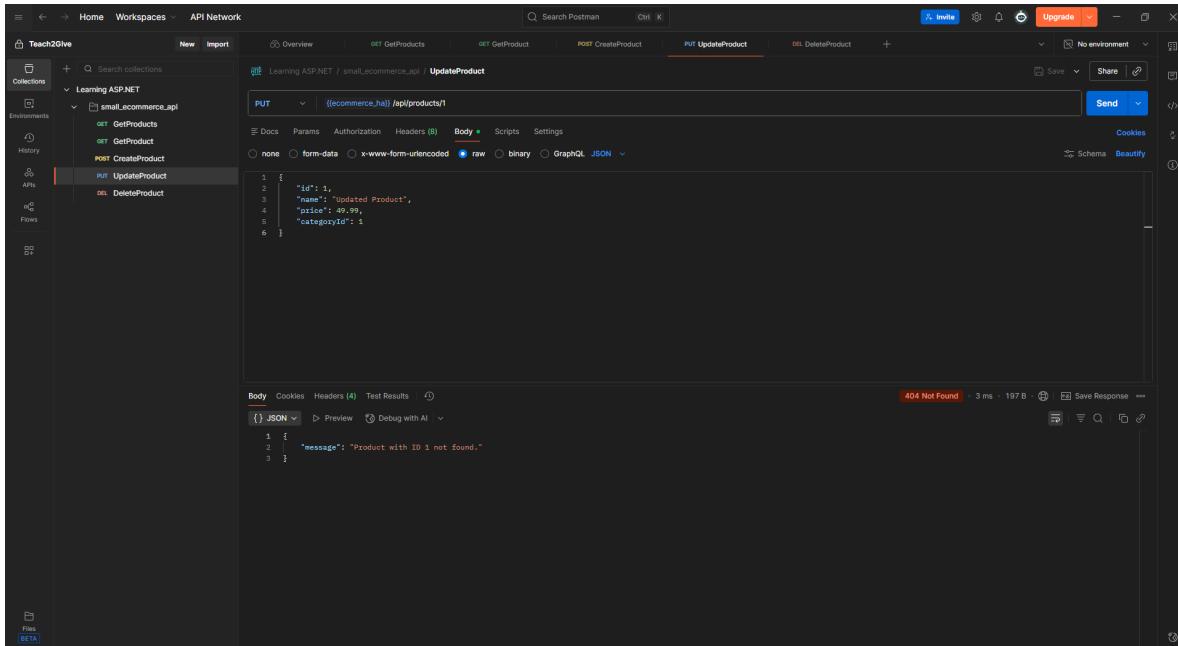
image_264.png

- [error] Update a mismatched id



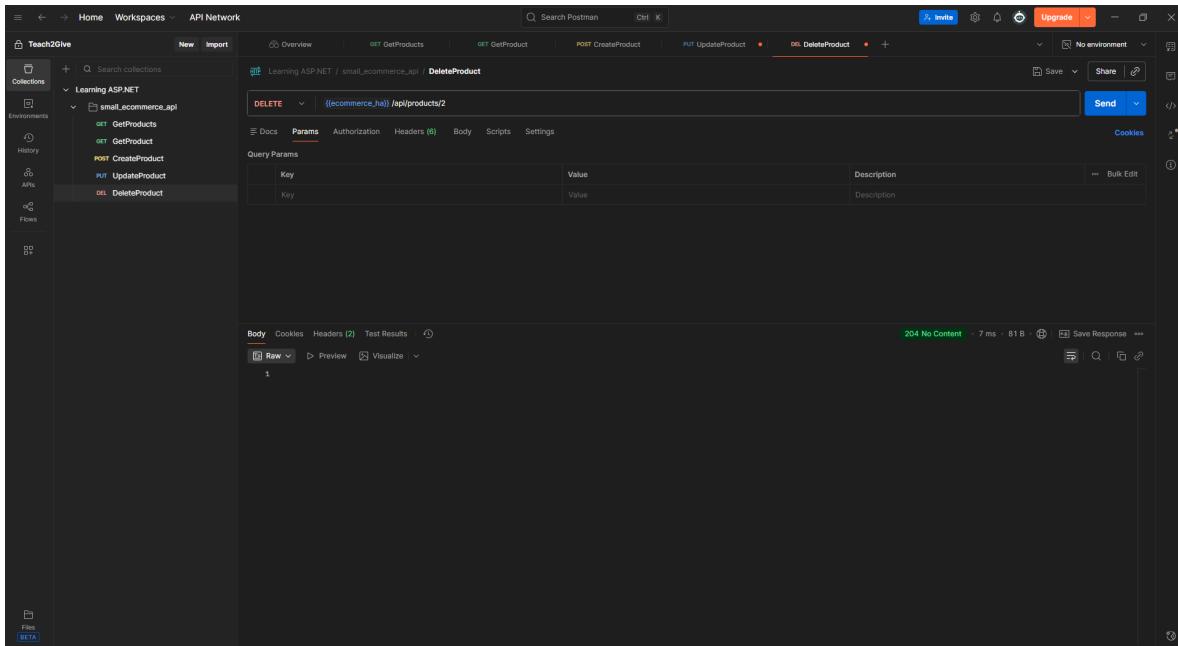
image_263.png

- [error] Update a missing product



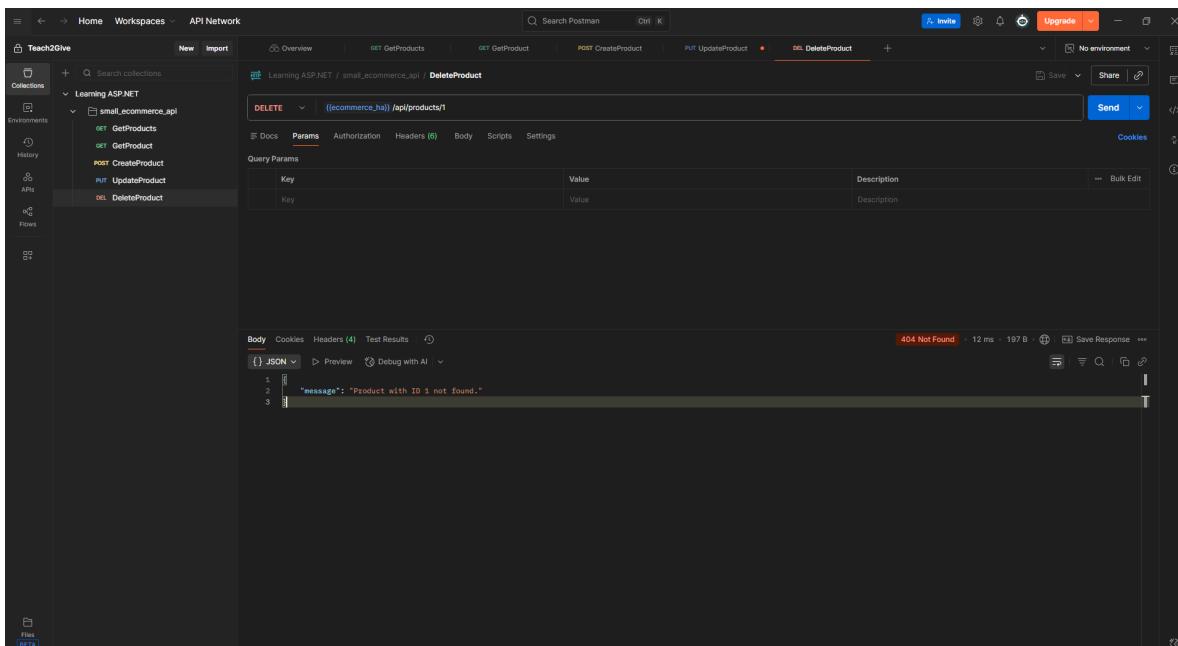
image_262.png

- Delete a product



image_266.png

- [Error] Delete a missing product



image_265.png

Services in ASP.NET Core Web API

In ASP.NET Core Web API, controllers are responsible for handling HTTP requests and returning responses. While it is technically possible to write all business logic and data access logic inside controllers, this approach quickly becomes messy and complicated to maintain as the project grows.

To solve this, **Services** are introduced. Services act as dedicated classes that contain the application's **business logic and data manipulation code**, keeping controllers clean and focused only on coordinating HTTP requests and responses. This separation of concerns improves readability, maintainability, testability, and scalability of the application.

Problems with Writing Logic Inside Controllers

If we directly embed business logic and data access operations inside controllers, the following problems arise:

- **Code Duplication:** Reusing logic across multiple controllers results in repeated code.
- **Reduced Testability:** Testing controllers becomes more challenging because business rules are intertwined with request-handling logic.
- **Tightly Coupled Code:** Changes in business rules force changes in multiple controllers, reducing flexibility.
- **Violation of the Single Responsibility Principle (SRP):** Controllers handle both HTTP request/response management and business rules, making them overloaded.

How Services Solve These Problems

By moving business logic into dedicated service classes, controllers can remain lightweight and focused only on HTTP concerns.

- **Encapsulation of Logic:** Business rules and data access logic live inside services.
- **Reusability:** Services can be shared across multiple controllers.
- **Testability:** Services can be unit-tested independently without worrying about HTTP request handling.

- Maintainability: Easier to manage, extend, or replace logic without affecting controllers.

Where Should We Write Validation Logic?

Validation in ASP.NET Core Web API can be handled in two ways:

- At the Controller Level: ASP.NET Core automatically validates model state if [ApiController] is used. Invalid requests return 400 Bad Request automatically.
- At the Service Level: Business-specific validations (e.g., “price cannot exceed 10,000” or “product name must be unique”) should be placed in services.

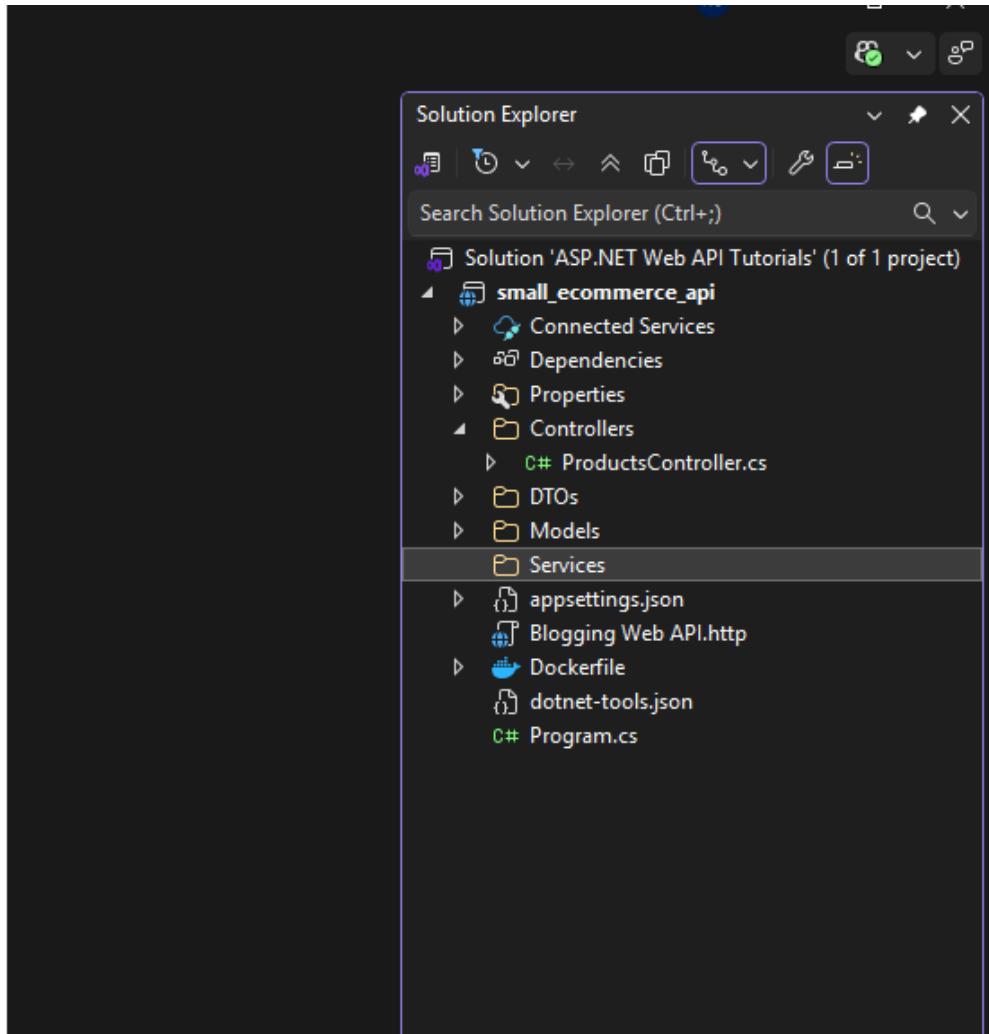


Best Practice:

- Use Data Annotations in Models/DTOs for general input validation (length, required fields, ranges).
- Use Services for business logic validations (unique constraints, cross-entity rules).

Creating a Product Service

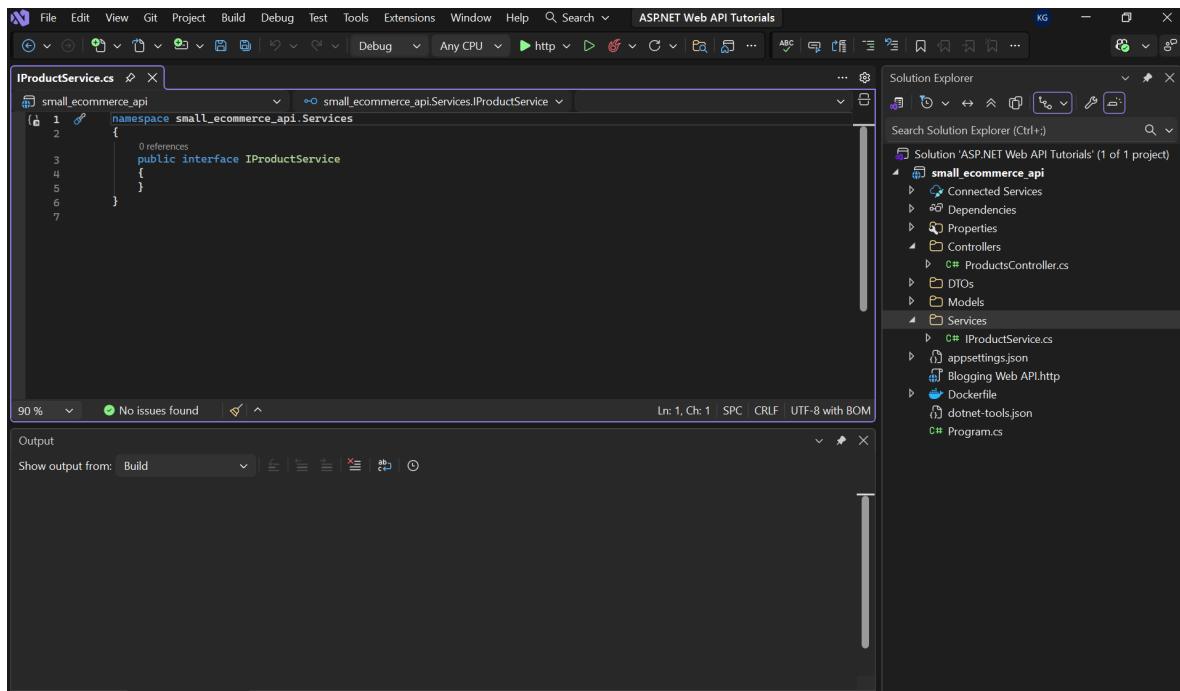
First, create a folder named ‘**Services**’ at the project root directory, where we will create all service interfaces and their corresponding concrete implementations.



image_267.png

Define an Interface (Contracts)

Create an interface named `IProductService.cs` within the **Services** folder, and copy-paste the following code.



image_268.png

```
using small_ecommerce_api.DTOs;

namespace small_ecommerce_api.Services
{
    public interface IProductService
    {
        IEnumerable<ProductDTO> GetAllProducts();
        ProductDTO? GetProductById(int id);
        ProductDTO CreateProduct(ProductCreateDTO createDto);
        bool UpdateProduct(int id, ProductUpdateDTO updateDto);
        bool DeleteProduct(int id);
    }
}
```

Code Explanations:

- Interfaces define a **Contract**. They specify **what methods exist** without defining **how they work**.

- This ensures **Loose Coupling**. The controller doesn't care about the actual implementation, only that the service provides these methods.
- It also enables **testability** (we can mock this interface in unit tests).

Implement the Service

Create a class file named `ProductService.cs` within the `Services` folder. The following class implements the `IProductService.cs` interface and provides implementations for all interface methods.

```
using small_ecommerce_api.DTOs;
using small_ecommerce_api.Models;

namespace small_ecommerce_api.Services
{
    public class ProductService : IProductService
    {
        private static List<Category> _categories = new()
        {
            new Category { Id = 1, Name = "Electronics" },
            new Category { Id = 2, Name = "Furniture" },
        };
        private static List<Product> _products = new()
        {
            new Product { Id = 1, Name = "Laptop", Price = 1000.00m,
CategoryId = 1 },
            new Product { Id = 2, Name = "Desktop", Price = 2000.00m,
CategoryId = 1 },
            new Product { Id = 3, Name = "Chair", Price = 150.00m,
CategoryId = 2 },
        };
        public IEnumerable<ProductDTO> GetAllProducts()
        {
            return _products.Select(p => new ProductDTO
            {
                Id = p.Id,
                Name = p.Name,
                Price = p.Price,
            });
        }
    }
}
```

```

        CategoryName = _categories.FirstOrDefault(c => c.Id ==
p.CategoryId)?.Name ?? "Unknown"
    });
}
public ProductDTO? GetProductById(int id)
{
    var product = _products.FirstOrDefault(p => p.Id == id);
    if (product == null) return null;
    return new ProductDTO
    {
        Id = product.Id,
        Name = product.Name,
        Price = product.Price,
        CategoryName = _categories.FirstOrDefault(c => c.Id ==
product.CategoryId)?.Name ?? "Unknown"
    };
}
public ProductDTO CreateProduct(ProductCreateDTO createDto)
{
    var newProduct = new Product
    {
        Id = _products.Max(p => p.Id) + 1,
        Name = createDto.Name,
        Price = createDto.Price,
        CategoryId = createDto.CategoryId
    };
    _products.Add(newProduct);
    return new ProductDTO
    {
        Id = newProduct.Id,
        Name = newProduct.Name,
        Price = newProduct.Price,
        CategoryName = _categories.FirstOrDefault(c => c.Id ==
newProduct.CategoryId)?.Name ?? "Unknown"
    };
}
public bool UpdateProduct(int id, ProductUpdateDTO updateDto)
{

```

```

        var product = _products.FirstOrDefault(p => p.Id == id);
        if (product == null) return false;
        product.Name = updateDto.Name;
        product.Price = updateDto.Price;
        product.CategoryId = updateDto.CategoryId;
        return true;
    }
    public bool DeleteProduct(int id)
    {
        var product = _products.FirstOrDefault(p => p.Id == id);
        if (product == null) return false;
        _products.Remove(product);
        return true;
    }
}

```

Code Explanations:

- This class provides the **Actual Implementation** of the interface.
- It encapsulates the **Business Logic** (e.g., creating, updating, and deleting products).
- It isolates **Data Handling** (right now, hardcoded lists; later, database logic).
- If the logic changes (say you move from in-memory list → EF Core database), only the service changes, not the controller.

Register the Service in the Program.cs

```

using small_ecommerce_api.Services;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();

// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
builder.Services.AddOpenApi();

```

```

// register product service
// AddScoped means a New Instance of ProductService will be created for
// each HTTP request.
builder.Services.AddScoped<IPrductService, ProductService>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

```

Code Explanations:

- ASP.NET Core uses **Dependency Injection (DI)** by default.
- Here, we register IPrductService with its implementation, ProductService.
- **AddScoped means a New Instance of ProductService will be created for each HTTP request.**
- This ensures services are **injected** automatically into controllers wherever required.



Real-time Analogy: This is like telling the Restaurant Manager: “Whenever a customer orders something from the menu (IPrductService), send it to this specific kitchen (ProductService).”

Use the Service in the Controller

Finally, update the Products Controller as follows.

```
using Microsoft.AspNetCore.Mvc;
using small_ecommerce_api.DTOs;
using small_ecommerce_api.Models;
using small_ecommerce_api.Services;

namespace small_ecommerce_api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : Controller
    {
        private readonly IProductService _productService;

        public ProductsController(IProductService productService)
        {
            _productService = productService;
        }

        // GET: api/products
        [HttpGet]
        public ActionResult<IEnumerable<ProductDTO>> GetProducts()
        {
            var Products = _productService.GetAllProducts();

            if (Products == null) {
                return NotFound(new { Message = "No products found." });
            }

            return Ok(Products);
        }

        // GET: api/products/{id}
        [HttpGet("{id}")]
        public ActionResult<ProductDTO> GetProduct(int id)
```

```

    {
        var product = _productService.GetProductById(id);
        if (product == null)
        {
            return NotFound(new { Message = $"Product with ID {id} not found." });
        }

        return Ok(product);
    }

    // POST: api/products
    [HttpPost]
    public ActionResult<ProductDTO> PostProduct([FromBody] ProductCreateDTO createDto)
    {
        var newProduct = _productService.CreateProduct(createDto);

        return CreatedAtAction(nameof(GetProduct), new { id = newProduct.Id }, newProduct);
    }

    // PUT: api/products/{id}
    [HttpPut("{id}")]
    public IActionResult UpdateProduct(int id, [FromBody] ProductUpdateDTO updateDto)
    {
        if (id != updateDto.Id)
        {
            return BadRequest(new { Message = "ID mismatch between route and body." });
        }

        var existingProduct = _productService.GetProductById(id);

        if (existingProduct == null)
        {
            return NotFound(new { Message = $"Product with ID {id}" });
        }
    }
}

```

```

        not found." });
    }

    return NoContent();
}

// DELETE: api/products/{id}
[HttpDelete("{id}")]
public IActionResult DeleteProduct(int id)
{
    var product = _productService.GetProductById(id);

    if (product == null)
    {
        return NotFound(new { Message = $"Product with ID {id} not found." });
    }

    return NoContent();
}

}

```

Code Explanations:

- The controller **depends** on IProductService, not directly on ProductService.
- ASP.NET Core's DI system automatically provides an instance of ProductService when the controller is created.
- This keeps the controller **clean**:
 - No product management logic inside.
 - Only coordinates between HTTP requests and the service methods.

⚠ Real-time Analogy: The waiter (**controller**) doesn't cook the food. He takes the order and passes it to the chef (**service**), then brings the result back to the customer.

Dependency Injection in ASP.NET Core Web API

Building modern applications requires writing clean, testable, and maintainable code. One of the most common challenges developers face is **managing dependencies**, which is the way one class relies on another to perform its work.

Need for Dependency Injection in ASP.NET Core

In Traditional Applications, classes often create and manage their own dependencies using the new keyword. This leads to Tight Coupling between classes, making the code difficult to maintain, extend, and test. So, when components create their own dependencies (tight coupling), we get several pain points:

- **Tight Coupling:** Changing a concrete implementation (e.g., swapping an in-memory store for a database) triggers edits across the codebase.
- **Difficult Testing:** Unit testing becomes more challenging as it is difficult to easily substitute dependencies with mocks.
- **Code Duplication:** Instantiating the same services in multiple places leads to redundancy.
- **Violation of SRP and OCP:** Your classes end up doing too much, and changes require touching multiple parts of the system.

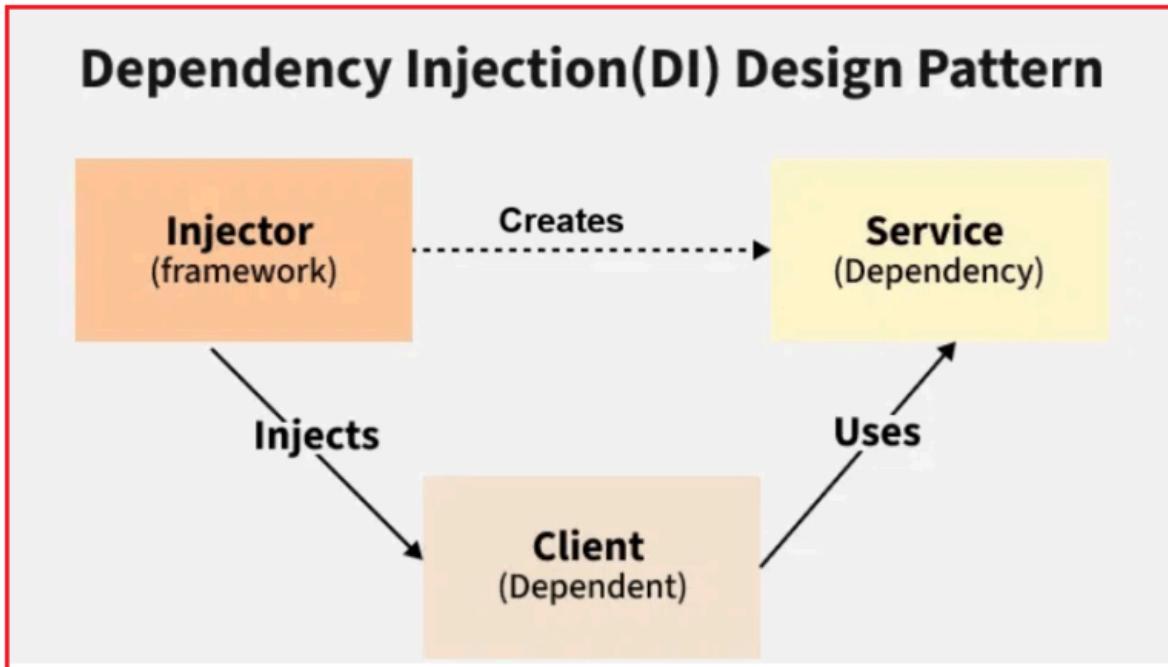
What is the Dependency Injection (DI) Design Pattern?

Dependency Injection is a Design Pattern that Removes Dependency Creation from a class and instead provides (injects) them externally.

- **Without DI:** Class A creates an instance of Class B.
- **With DI:** Class A receives an instance of Class B from outside (Container).

Components involved in the Dependency Injection (DI) Design Pattern

There are four components involved in the Dependency Injection Design Pattern. For a better understanding, please have a look at the following diagram. Here, Service represents two things: the Interface and its implementation class.



image_269.png

Abstraction (Interface / Contract)

An abstraction is like a **contract** that defines what the service can do, but not how it does it. In C#, this is usually an **interface**. The client depends only on this abstraction, not on the actual service implementation. This way, the client doesn't care which service it gets, as long as it follows the contract.

Key Points:

- Defines what needs to be done, not how.
- Usually written as an **interface** in code (e.g., `IProductService`).
- Makes the system **flexible** — you can easily swap implementations.
- Keeps the **client (user)** free from knowing service details.

Service (Dependency / Implementation)

A service is the class that actually does the work. It provides some functionality that other parts of the application need. It contains the logic that your application needs, like

sending an email, saving data to a database, or processing a payment. Without the service, your program cannot complete its tasks.

Key Points:

- It is the **real worker** of your system.
- Contains the actual **business logic**.
- It is the **dependency** that other classes need.
- Examples: EmailService, ProductRepository, PaymentService.
- Can have **multiple versions** (e.g., EmailService using Gmail or Outlook).

Client (Consumer / Dependent Class)

A client is the class that **needs the service** to perform its job. It doesn't do the actual work itself but depends on a service to get things done. The client depends on the service but does not care about how it is created. For example, an OrderController needs an IEmailService to send order confirmation emails.

Key Points:

- It is the **user of the service**.
- Relies on an abstraction (interface), not on the concrete service.
- It only **uses** the service; it does not **create** it.
- Example: A controller using IProductService for CRUD operations.
- Becomes **simpler and cleaner** because it doesn't create dependencies on its own.

Injector (Dependency Injection Container)

The injector is like the **manager** who provides the right worker (service) to the client. In ASP.NET Core, this is the **built-in DI container**. It creates service objects, gives them to the classes that need them, and manages their lifetime (i.e., how long they live).

Key Points:

Also called the **DI container**. **Registers** services with their matching interfaces. **Resolves** dependencies when a client needs them. **Injects** the correct implementation at runtime.

Manages **lifetimes**: Transient (new every time), Scoped (per request), Singleton (once for the whole app).

Types of Services in ASP.NET Core: Custom vs Built-in Services

They are mainly of two types:

Built-in Services:

ASP.NET Core provides many built-in services, such as:

- **Logging** (ILogger),
- **Configuration** (IConfiguration),
- **Options** (IOptions, IOptionsMonitor),
- **HTTP** (IHttpClientFactory),
- **Caching** (IMemoryCache, IDistributedCache),
- **Authentication/Authorization, Routing/MVC** services, etc.

Custom Services:

You can create your own services (e.g., `IStudentRepository`, `EmailService`) and register them in the DI container

- Email sending (EmailService).
- Background job scheduler (JobService).
- Notification manager (NotificationService).
- Third-party API integrations (e.g., SMS, payments).

How to Register a Service with the ASP.NET Core DI Container?

We need to register a service with the ASP.NET Core Dependency Injection Container within the Main method of the Program class. All registrations happen in the Program.cs (composition root) using builder.Services. ASP.NET Core provides three main lifetimes:

- **Singleton**: One instance throughout the application. Syntax:

```
builder.Services.AddSingleton<IPrductRepository, ProductRepository>();
```

- **Scoped**: One instance per HTTP request. Syntax:

```
builder.Services.AddScoped<IPrductRepository, ProductRepository>();
```

- **Transient**: New instance every time it's requested. Syntax:

```
builder.Services.AddTransient<IPrductRepository, ProductRepository>();
```

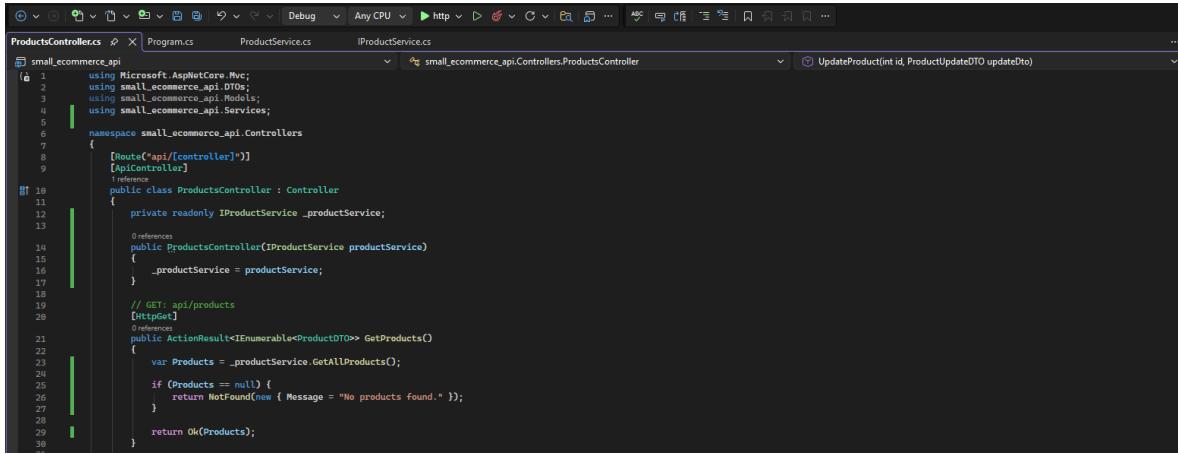
Different Ways of Accessing Dependencies in ASP.NET Core Web API

There are 3 different approaches that we can use to inject the dependency object into a class in ASP.NET Core.

- **Constructor Injection** – The Recommended Approach
- Action Method Injection – For Occasional Dependencies
- Manual Service Resolution – Use Only When Necessary

Constructor Injection (The Recommended Approach)

Constructor Injection is the most commonly used and recommended method of injecting dependencies. The DI container automatically creates and passes dependencies to the class constructor at runtime. This ensures your object is fully initialized and ready to perform its intended tasks



```
1  using Microsoft.AspNetCore.Mvc;
2  using small_ecommerce_api.DTOs;
3  using small_ecommerce_api.Models;
4  using small_ecommerce_api.Services;
5
6  namespace small_ecommerce_api.Controllers
7  {
8      [Route("api/[controller]")]
9      [ApiController]
10     public class ProductsController : Controller
11     {
12         private readonly IProductService _productService;
13
14         public ProductsController(IProductService productService)
15         {
16             _productService = productService;
17         }
18
19         // GET: api/products
20         [HttpGet]
21         public ActionResult<IEnumerable<ProductDTO>> GetProducts()
22         {
23             var Products = _productService.GetAllProducts();
24
25             if (Products == null)
26             {
27                 return NotFound(new { Message = "No products found." });
28             }
29
30             return Ok(Products);
31         }
32     }
33 }
```

image_270.png

Real-Time Use Cases:

- Business services like IProductService, IOrderService, ILogger, or DbContext.
- When the service is needed by multiple action methods in the same controller.
- Example: A ProductsController needs IProductService for CRUD operations

Action-Method Injection (For Occasional Dependencies)

If a service is needed **only inside a specific action, one or a few endpoints** (not the whole controller), injecting it into the controller constructor would be unnecessary. Instead, use **[FromServices]** in the method signature, i.e., inject it at the action parameter using **[FromServices]**.

Syntax:

```
[HttpGet("{id:int}")]
1 reference
public ActionResult<Product> GetById(
    int id,
    [FromServices] IProductService productService) // injected only for this action
{
    var product = productService.GetById(id);
    return product is null ? NotFound() : Ok(product);
}
```

image_271.png

Real-Time Use Cases

- Dependencies that are **used only in one or two endpoints**
- A service used for **special cases**, such as sending an email notification, generating a report, or exporting data. **Example:**
 - A **one-off service** for generating PDF reports (**IReportService**) used in only one endpoint.
 - A **temporary feature**, such as **ICaptchaValidator**, is used only in the registration API.

Manually Resolving Services (Use Only When Necessary)

You need to use `HttpContext.RequestServices.GetService<T>()` to manually resolve a service at runtime. This is useful in **middleware**, **filters**, or in situations where injection isn't possible.

Syntax:

```
public ActionResult<Product> Create([FromBody] Product input)
{
    var productService = HttpContext.RequestServices.GetRequiredService<IPrductService>();

    try
    {
        var created = productService.Create(input);
        return CreatedAtAction(nameof(GetById), new { id = created.Id }, created);
    }
    catch (ArgumentException ex)
    {
        return ValidationProblem(detail: ex.Message);
    }
}
```

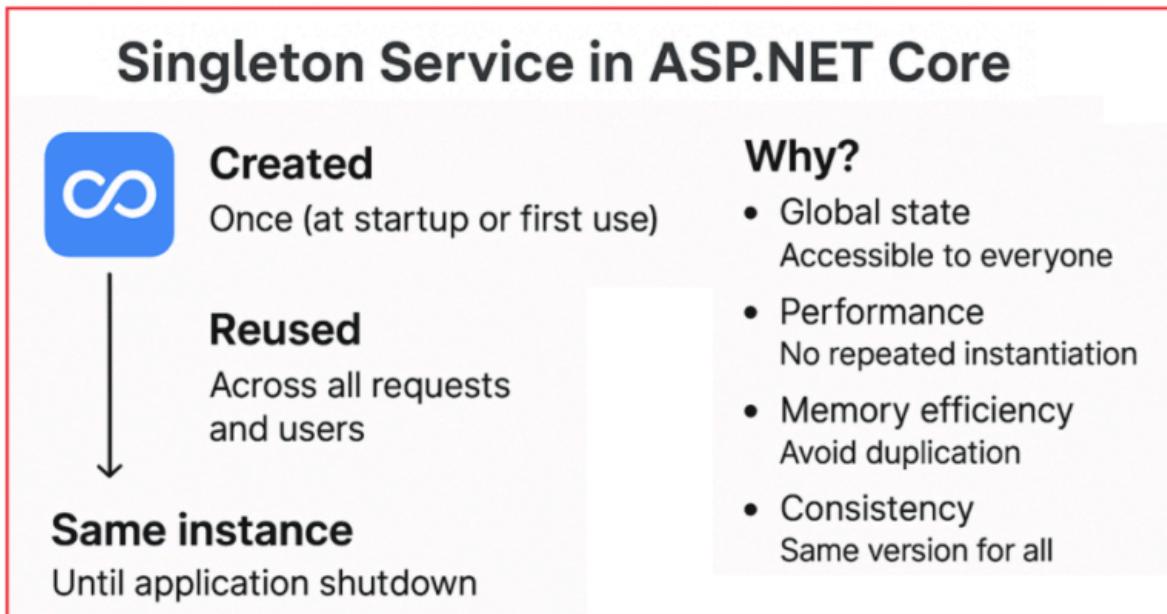
image_272.png

DI : Singleton vs Scoped vs Transient in ASP.NET Core Web API

Singleton Service in ASP.NET Core

A Singleton Service is the simplest and most powerful way to share one instance across the entire Application Lifetime. That means:

- It is **created only once** (at application startup or on the first time it's requested).
- It is **then reused across all HTTP requests, all controllers, and all users**.
- The same object instance lives until the application shuts down.



image_273.png

Why Singleton Service in ASP.NET Core Web API?

Singleton services are best suited when you want a single shared resource that does not depend on request-specific data. Some key benefits:

- **Global State** → If you want to keep something in memory and accessible to everyone (such as a tax configuration or a cache), a Singleton works perfectly.

- **Performance** → Since the service is created only once, you avoid the cost of creating a new instance repeatedly.
- **Memory Efficiency** → Objects that are expensive to create (e.g., database connection pools, API clients, machine learning models, etc.) don't need to be duplicated for every request.
- **Consistency** → All requests see the same version of the service.

Use Case of Singleton Service in ASP.NET Core:

Some common real-world scenarios where Singleton makes sense:

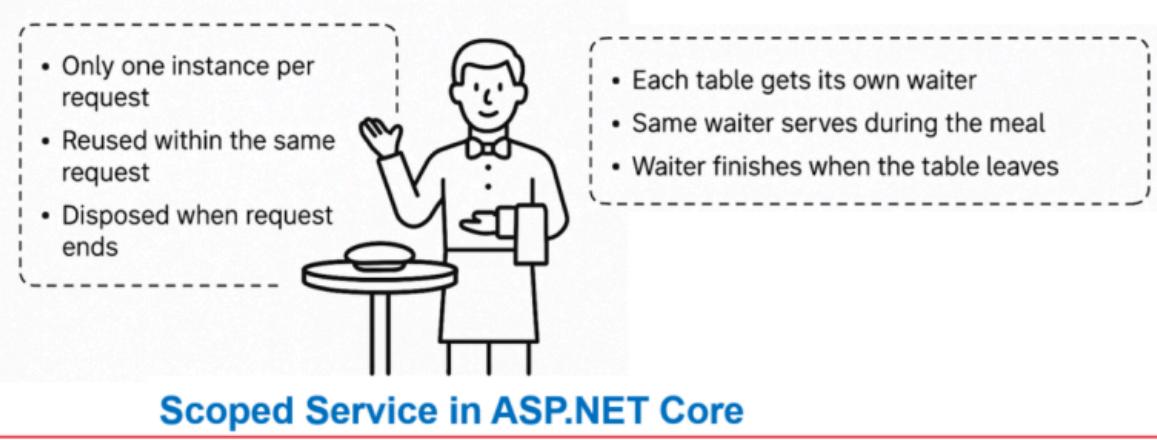
- **Caching** → Store frequently used data (e.g., product categories, configuration, lookup tables) in memory for fast access.
- **Configuration Providers** → A central service that loads configuration settings (e.g., appsettings.json, environment variables) and provides them to other services.
- **Logging** → A logging service can be shared across the entire app (all requests log to the same logger).
- **External API Clients** → If your app needs to call external APIs (e.g., payment gateway, weather API), you don't want to create a new HTTP client every time — instead, use IHttpClientFactory or a Singleton service.

Scoped Service in ASP.NET Core

A **Scoped service** is created **once per HTTP request**.

- Within a single request, the **same instance is reused** everywhere it's injected.
- But as soon as a new HTTP request starts, ASP.NET Core creates a **new instance** for that request.
- Once the request ends, the instance is **disposed** (cleaned up by the DI container).

So, it lives only as long as the HTTP request scope. For a better understanding, please refer to the following diagram.



image_274.png

Why Scoped Service in ASP.NET Core Web API?

Scoped services are useful when you need to **maintain state per request**, but you don't want that state to leak across different users or requests.

Examples:

- A shopping cart that exists only while a request is being processed.
- A UserContext service that holds info about the currently logged-in user.
- A Unit of Work service where one request = one transaction.

Scoped ensures isolation → each request gets its own copy, but consistency inside that request.

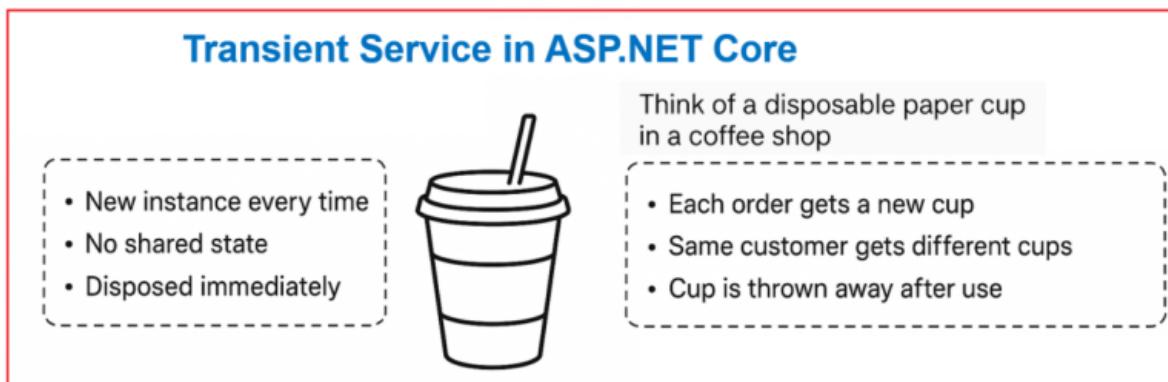
Transient Service in ASP.NET Core

A **Transient service** is the **shortest-lived service** in ASP.NET Core's Dependency Injection system.

- A **new instance is created every time** it is requested from the DI container.
- Even within the same request, if you inject it twice, you will get **two completely separate objects**.
- As a result, the service **doesn't hold any shared state** across or within requests.

- Once used, it's disposed (garbage collected) quickly, no reuse at all.

This makes transient services **ideal for lightweight, stateless tasks** that don't rely on user data, shared context, or long-lived memory. For a better understanding, please refer to the following diagram.



image_275.png

When to Use Transient Service in ASP.NET Core:

Best for services that are stateless and lightweight:

- Calculation helpers** → e.g., discount calculation, price formatting, currency conversion.
- Utility classes** → string formatters, mappers, validators.
- Random generators/ ID generators** → if you specifically want a different value each time.
- Stateless API wrappers** → if they don't need to cache or persist anything.

[Workshop] DI in ShoppingCart API

Imagine an e-commerce system.

1. The **CartService** stores items added by the user.
2. The **DiscountService** provides dynamic discounts.
3. The **AppConfigService** provides global configuration (like tax rate, delivery fee).

Create a New ASP.NET Core Web API Project

First, create a new ASP.NET Core Web API Project, name it ShoppingCartAPI.

Define Models

Create a new folder, **Models**, at the project root directory. Then, add the following classes inside the **Models** folder.

CartItem.cs

Represents an item added to the cart, including *ProductId*, *Name*, *Price*, and *Quantity*.

Create a class file named **CartItem.cs** within the **Models** folder.

```
namespace ShoppingCartAPI.Models
{
    public class CartItem
    {
        public int ProductId { get; set; }
        public string ProductName { get; set; } = string.Empty;
        public decimal Price { get; set; }
        public int Quantity { get; set; }
    }
}
```

CartSummary.cs

Represents the final computed values of a cart: Subtotal, Discount, Tax, Delivery Fee, and Final Total. Create a class file named **CartSummary.cs** within the **Models** folder.

```
namespace ShoppingCartAPI.Models
{
    public class CartSummary
    {
        public decimal SubTotal { get; set; }
        public decimal Discount { get; set; }
        public decimal Tax { get; set; }
        public decimal DeliveryFee { get; set; }
        public decimal Total { get; set; }
    }
}
```

Define Services

Create a new folder **Services** and **Interfaces** at the project root directory. Then, add the following classes and interfaces inside the Services and interfaces folder respectively.

Singleton Service → AppConfigService

Create an interface named **IAppConfigService.cs** within the Services/Interface folder.

```
namespace ShoppingCartAPI.Services.Interfaces
{
    public interface IAppConfigService
    {
        decimal GetTaxRate();
        decimal GetDeliveryFee(decimal orderAmount);
    }
}
```

AppConfigService

This class provides application-wide configuration data, such as the **GST tax rate** and **delivery fee**, based on the cart amount. These values don't change per request or user. Create a class file named **AppConfigService.cs** within the **Services** folder.

```

using ShoppingCartAPI.Services.Interfaces;

namespace ShoppingCartAPI.Services
{
    public class AppConfigService : IAppConfigService
    {
        private readonly decimal _taxRate = 0.18m; // GST rate = 18%

        public AppConfigService	ILogger<AppConfigService> logger)
        {
            // Log only once since Singleton → single instance shared
            across app
            logger.LogInformation("AppConfigService (Singleton) instance
created.");
        }

        public decimal GetDeliveryFee(decimal orderAmount)
        {

            if (orderAmount < 500)
                return 50; // Flat ₹50 for small orders
            else if (orderAmount >= 500 && orderAmount <= 2000)
                return 30; // Reduced fee for mid-sized orders
            else
                return 0; // Free delivery for big orders
        }

        public decimal GetTaxRate()
        {
            return _taxRate;
        }
    }
}

```

Why Singleton?

- It holds static, read-mostly configuration.
- It's shared across all users and requests.
- Saves memory and avoids unnecessary object creation.

Scoped Service → CartService

Stores the **shopping cart** for a single user request. Create an interface named **ICartService.cs** within the **Services/Interfaces** folder.

```
using ShoppingCartAPI.Models;

namespace ShoppingCartAPI.Services.Interfaces
{
    public interface ICartService
    {
        void AddItem(CartItem item);
        List<CartItem> GetItems();
        void ClearCart();
    }
}
```

CartService

This service manages the user's shopping cart for the current HTTP request. It fetches the cart from **IMemoryCache**, adds/updates items, and clears the cart as needed. Create a class file named **CartService.cs** within the **Services** folder.

```
using Microsoft.Extensions.Caching.Memory;
using ShoppingCartAPI.Models;
using ShoppingCartAPI.Services.Interfaces;

namespace ShoppingCartAPI.Services
{
    public class CartService : ICartService
    {
        private readonly IMemoryCache _cache;
        private readonly string _userId;
```

```

    private readonly ILogger<CartService> _logger;

    public CartService(IMemoryCache cache, IHttpContextAccessor
httpContextAccessor, ILogger<CartService> logger)
    {
        _cache = cache;
        _userId =
httpContextAccessor.HttpContext?.Request.Headers["UserId"].FirstOrDefault() ?? "guest";
        _logger = logger;
        _logger.LogInformation("CartService (Scoped) instance
created for user {UserId}", _userId);
    }

    public void AddItem(CartItem item)
    {
        _logger.LogInformation("Adding item for user {UserId}:
{ProductName}", _userId, item.ProductName);

        var cart = _cache.GetOrCreate(_userId, entry =>
        {
            _logger.LogInformation("Creating new cart for user
{UserId}", _userId);
            entry SlidingExpiration = TimeSpan.FromMinutes(30);
            return new List<CartItem>();
        })!;

        var existing = cart.FirstOrDefault(x => x.ProductId ==
item.ProductId);
        if (existing != null)
        {
            existing.Quantity += item.Quantity;
            _logger.LogInformation("Updated quantity for ProductId
{ProductId}, new quantity: {Quantity}", item.ProductId,
existing.Quantity);
        }
        else
        {

```

```

        cart.Add(item);
        _logger.LogInformation("Added new item. Cart now has
{Count} items", cart.Count);
    }

    _cache.Set(_userId, cart);
    _logger.LogInformation("Cart saved to cache for user
{UserId}, total items: {Count}", _userId, cart.Count);
}

public List<CartItem> GetItems()
{
    _logger.LogInformation("Getting items for user {UserId}",
.userId);

    if (_cache.TryGetValue(_userId, out List<CartItem>? cart))
    {
        _logger.LogInformation("Cart found for user {UserId},
contains {Count} items", _userId, cart?.Count ?? 0);
        return cart ?? new List<CartItem>();
    }

    _logger.LogWarning("No cart found in cache for user
{UserId}", _userId);
    return new List<CartItem>();
}

public void ClearCart()
{
    _logger.LogInformation("Clearing cart for user {UserId}",
.userId);
    _cache.Remove(_userId);
}
}

```

Why Scoped?

- A new instance is created per request.
- It ensures request isolation, but the user cart persists across requests via `IMemoryCache` (singleton).
- Allows logging and handling per-user state safely within a request.

Transient Service → `DiscountService`

Provides a **tier-based discount logic** (5%, 10%, 15%) depending on the subtotal amount. Create an interface named `IDiscountService.cs` within the `Services/Interfaces` folder.

```
namespace ShoppingCartAPI.Services.Interfaces
{
    public interface IDiscountService
    {
        decimal CalculateDiscount(decimal amount);
    }
}
```

`DiscountService`

Calculates discounts based on order amount (5%, 10%, 15%). Each injection creates a new instance so that you can see multiple calculations in the same request. Create a class file named `DiscountService.cs` within the `Services` folder.

```
using ShoppingCartAPI.Services.Interfaces;

namespace ShoppingCartAPI.Services
{
    public class DiscountService : IDiscountService
    {
        public DiscountService(ILogger<DiscountService> logger)
        {
            // Log whenever a new DiscountService instance is created
            // (helps visualize Transient lifetime → multiple per
request possible)
            logger.LogInformation("DiscountService (Transient) instance
created.");
    }

    decimal CalculateDiscount(decimal amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("Amount must be greater than zero.");
        }

        decimal discount = 0;
        if (amount > 1000)
        {
            discount = amount * 0.15;
        }
        else if (amount > 500)
        {
            discount = amount * 0.1;
        }
        else
        {
            discount = amount * 0.05;
        }

        return discount;
    }
}
```

```

    }

    public decimal CalculateDiscount(decimal amount)
    {
        // Tier-based discount calculation
        decimal discountPercent = 0;
        if (amount >= 5000 && amount <= 20000)
            discountPercent = 5;    // 5% discount for mid-level
orders
        else if (amount > 20000 && amount <= 50000)
            discountPercent = 10;   // 10% discount for higher orders
        else if (amount > 50000)
            discountPercent = 15;   // 15% discount for premium
orders
        // Calculate discount amount
        decimal discount = amount * discountPercent / 100;
        return discount;
    }
}
}

```

Why Transient?

- It is stateless and lightweight.
- We want a new instance every time (even within the same request), especially to show how different injections behave independently.
- Suitable for one-off calculations.

Scoped Service → ICartSummaryService

Create an interface named **ICartSummaryService.cs** within the **Services/Interfaces** folder.

```

using ShoppingCartAPI.Models;

namespace ShoppingCartAPI.Services.Interfaces
{
    public interface ICartSummaryService

```

```
{  
    CartSummary GenerateSummary();  
}  
}
```

CartSummaryService

Generates the **final cart summary** (Subtotal, Discount, Tax, Delivery Fee, and Total).

Depends on:

- CartService (Scoped → per-request cart data),
- DiscountService (Transient → fresh calculations),
- AppConfigService (Singleton → global tax/delivery rules).

Create a class file named `CartSummaryService.cs` within the Services folder.

```
using ShoppingCartAPI.Models;  
using ShoppingCartAPI.Services.Interfaces;  
  
namespace ShoppingCartAPI.Services  
{  
    public class CartSummaryService : ICartSummaryService  
    {  
        // Dependencies injected via constructor  
        private readonly ICartService _cartService;          // To fetch  
        items already added to the cart  
        private readonly IDiscountService _discount1;        // First  
        transient discount service  
        private readonly IDiscountService _discount2;        // Second  
        transient discount service (for demo showing new instance)  
        private readonly IAppConfigService _config;           // For global  
        configuration (tax rate, delivery fee)  
        // Constructor injection - services are provided by DI container  
        public CartSummaryService(  
            ICartService cartService,  
            IDiscountService discount1,
```

```

        IDiscountService discount2,
        IAppConfigService config)
    {
        _cartService = cartService;
        _discount1 = discount1;
        _discount2 = discount2;
        _config = config;
    }
    // Main method to calculate and return cart summary
    public CartSummary GenerateSummary()
    {
        // Fetch all cart items from the cart service
        var items = _cartService.GetItems();
        // Calculate subtotal = sum of price * quantity for all
        items
        decimal subTotal = items.Sum(i => i.Price * i.Quantity);
        // Apply discounts using two transient instances
        // (each transient service will likely give different
        results,
        // to demonstrate lifetime differences)
        decimal discount1 = _discount1.CalculateDiscount(subTotal);
        decimal discount2 = _discount2.CalculateDiscount(subTotal);
        // Calculate tax using the Singleton AppConfigService
        decimal tax = subTotal * _config.GetTaxRate();
        // Calculate delivery fee using dynamic business rules
        decimal delivery = _config.GetDeliveryFee(subTotal);
        // Build and return the summary object
        return new CartSummary
        {
            SubTotal = subTotal,                                //
            Original total before tax/discounts
            Discount = (discount1 + discount2) / 2,           //
            Average of both discounts (demo purpose)
            Tax = tax,                                         //
            Calculated tax
            DeliveryFee = delivery,                           //
            Delivery fee based on subtotal
            Total = subTotal - ((discount1 + discount2) / 2) // Net
        };
    }
}

```

```

        total = subtotal - discount + tax + delivery
                + tax
                + delivery
            };
        }
    }
}

```

Why Scoped?

- It depends on the scoped CartService, transient DiscountService, and singleton AppConfigService.
- Scoped ensures a **fresh summary per request**, yet internally respects the correct lifetimes of its dependencies.

Register Services in Program.cs

Now, we need to register the services with a proper lifetime. So, please update the `Program.cs` class file as follows:

```

using ShoppingCartAPI.Services;
using ShoppingCartAPI.Services.Interfaces;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
builder.Services.AddOpenApi();

// Service Register lifetimes
// Register AppConfigService as Singleton
// One instance is created and shared across the entire application
lifetime
//      - Good for global config like tax rate, delivery fee
//      - Created once, reused everywhere

```

```

builder.Services.AddSingleton<IAppConfigService, AppConfigService>();
// Register CartService as Scoped
// One instance per HTTP request
//   - Each request gets its own CartService instance
//   - Items stored in CartService are isolated to that request
//   - Demonstrates per-request lifetime
builder.Services.AddScoped<ICartService, CartService>();
// Register DiscountService as Transient
// A new instance is created every time it is requested
//   - Even within the same request, multiple injections create
different objects
//   - Great for lightweight, stateless operations like discount
calculations
builder.Services.AddTransient<IDiscountService, DiscountService>();
// Register CartSummaryService as Scoped
// New instance per HTTP request
//   - Depends on CartService, DiscountService, AppConfigService
//   - Calculates cart totals (subtotal, discount, tax, delivery, final
total)
//   - Scoped makes sense because summary is tied to the current
cart/request
builder.Services.AddScoped<ICartSummaryService, CartSummaryService>();
// Register HttpContextAccessor
// Provides access to HttpContext (e.g., reading headers like "UserId")
//   - Needed for CartService to know which user's cart to manage
builder.Services.AddHttpContextAccessor();
// Register In-Memory Cache
// Provides IMemoryCache (Singleton under the hood)
//   - Used by CartService to persist cart data across requests for a
given user
//   - Supports expiration policies (e.g., cart expires after 30
minutes of inactivity)
builder.Services.AddMemoryCache();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())

```

```
{  
    app.MapOpenApi();  
}  
  
app.UseHttpsRedirection();  
  
app.UseAuthorization();  
  
app.MapControllers();  
  
app.Run();
```

Creating Controllers

CartController.cs

Handles the HTTP endpoints:

- POST /add to add an item to the cart
- GET /items to view cart contents
- GET /summary to calculate totals
- DELETE /clear to empty the cart

It demonstrates how services with different lifetimes work together to build a real-time user experience. So, create an Empty API Controller named CartController within the Controllers folder.

```
using Microsoft.AspNetCore.Mvc;  
using ShoppingCartAPI.Models;  
using ShoppingCartAPI.Services.Interfaces;  
  
namespace ShoppingCartAPI.Controllers  
{  
    [Route("api/[controller]")]  
    public class CartController : Controller  
    {  
        // Injected dependency for cart operations (Scoped service)
```

```

private readonly ICartService _cartService;
// Constructor Injection
// ASP.NET Core automatically resolves ICartService from the DI
container
public CartController(ICartService cartService)
{
    _cartService = cartService;
}

// POST: api/cart/add
// Adds an item to the user's cart
[HttpPost("add")]
public IActionResult AddItem([FromBody] CartItem item)
{
    // Call cart service to add item to the in-memory cache
    _cartService.AddItem(item);
    // Return success response with a message
    return Ok(new { Message = $"{item.ProductName} added to
cart." });
}
// GET: api/cart/items
// Returns all items currently in the user's cart
[HttpGet("items")]
public IActionResult GetItems()
{
    // Fetch items from cart service
    return Ok(_cartService.GetItems());
}
// GET: api/cart/summary
// Returns the calculated cart summary (subtotal, discounts,
tax, delivery fee, total)
[HttpGet("summary")]
public IActionResult GetSummary([FromServices]
ICartSummaryService summaryService)
{
    // Here, CartSummaryService is injected per-request (Scoped)
    // It internally uses:
    //     - ICartService (Scoped, manages cart data)
}

```

```

        // - IDiscountService (Transient, two different instances
for discount calculations)
        // - IAppConfigService (Singleton, global tax & delivery
fee rules)
        var summary = summaryService.GenerateSummary();
        // Return summary object as JSON response
        return Ok(summary);
    }
    // DELETE: api/cart/clear
    // Clears all items from the user's cart
    [HttpDelete("clear")]
    public IActionResult ClearCart()
    {
        // Clear user's cart from cache
        _cartService.ClearCart();
        // Return success response
        return Ok(new { Message = "Cart cleared." });
    }
}

```

Testing ShoppingCartAPI in Postman

Request 1: Add Laptop

POST /api/cart/add

```
{
  "productId": 1,
  "productName": "Laptop",
  "price": 60000,
  "quantity": 1
}
```

Here:

- A **Scoped CartService** instance is created for this request.
- The Laptop item is added to the cart (stored in **IMemoryCache**, keyed by **UserId**).

- **Log Output:**

- CartService (Scoped) instance created for user user123
- If this is the very first request, you'll also see: AppConfigService (Singleton) instance created.

Request 2: Add Mobile

POST /api/cart/add

```
{  
  "productId": 2,  
  "productName": "Mobile",  
  "price": 10000,  
  "quantity": 2  
}
```

View Items

GET /api/cart/items

Here,

- Creates another **Scoped CartService** instance.
- Retrieves cart items from **IMemoryCache**.
- You'll see **both Laptop + Mobile**, because cache persists data beyond request lifetime.
- **Log Output:**
 - CartService (Scoped) instance created for user user123.

Get Cart Summary

GET /api/cart/summary

Here,

- Creates a new Scoped CartService instance.
- Fetches all cart items (Laptop + Mobile).

- Two Transient DiscountService instances are created:
 - `_discount1`
 - `_discount2`
- Each calculates a discount based on the subtotal (tiered: 5%, 10%, 15%).
- Singleton AppConfigService is reused (no new log).
 - It now calculates the delivery fee dynamically:
 - Subtotal < 500 → ₹50
 - Subtotal 500–2000 → ₹30
 - Subtotal > 2000 → Free delivery (₹0)

Log Output:

- CartService (Scoped) instance created for user user123
- DiscountService (Transient) instance created.
- DiscountService (Transient) instance created.

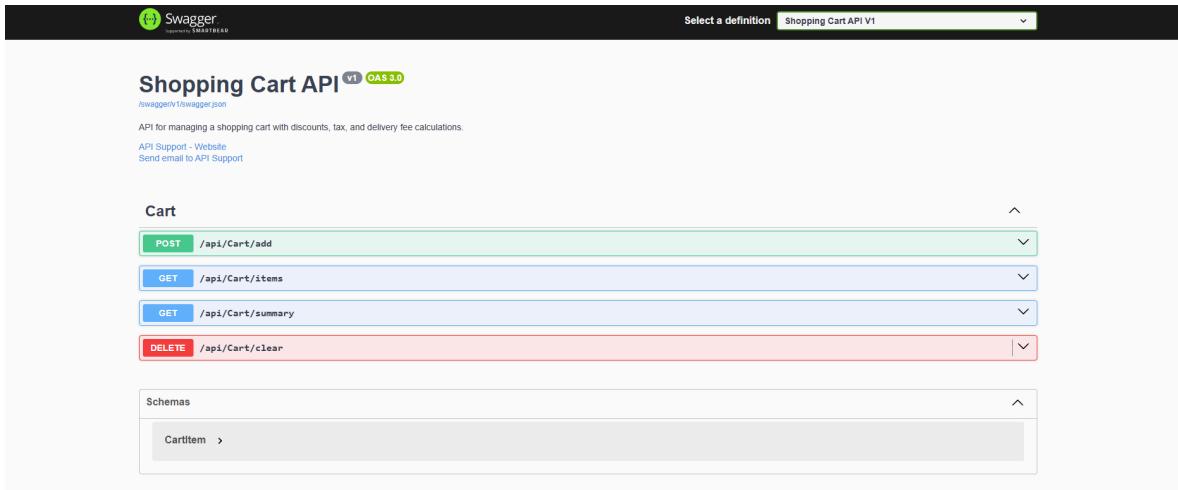
Best Practices

- Choose **Singleton** for shared, thread-safe, read-mostly resources you want to reuse.
- Choose **Scoped** for anything that logically lives for a single HTTP request.
- Choose **Transient** for small, stateless helpers you don't mind recreating often.

Understanding these lifetimes is crucial for building efficient and bug-free ASP.NET Core Web APIs. Improper use can cause:

- Memory leaks (e.g., injecting scoped into a singleton),
- Unwanted shared states, or
- Performance issues.

Swagger API in ASP.NET Core Web API



image_278.png

Swagger, often implemented using the **Swashbuckle** package, is a powerful tool used for documenting RESTful APIs developed with ASP.NET Core Web API. It provides a way to help plan, design, and document RESTful APIs. Swagger in ASP.NET Core Web API helps developers create, document, and consume RESTful APIs.

Key Features of Swagger in ASP.NET Core Web API:

- API Documentation:** Swagger automatically generates interactive API documentation, known as Swagger UI, based on your ASP.NET Core Web API Project. This documentation includes details about endpoints, parameters, request and response schemas, and even allows users to try out API calls directly from the documentation interface. The documentation can be dynamically updated as changes occur in the API.
- Standardization:** Swagger uses the OpenAPI Specification (OAS), which is a widely adopted industry standard for documenting RESTful APIs.
- Testing and Debugging:** Through Swagger UI, developers can send requests to the API, view the responses, and effectively test API functionality in a user-friendly web

interface without the need for a separate tool or writing additional code. This real-time interaction helps in debugging and improving the API during the development phase.

- **API Versioning:** Swagger supports easy API versioning, which is important for evolving an API without breaking the existing client integrations. It provides a clear and structured way to communicate changes and deprecations to API consumers.

How to Install the Swagger Package in ASP.NET Core Web API:

⚠ Please note this tutorial is for Swashbuckle.AspNetCore v10 and above. Below this there are breaking changes.

Integrating Swagger into the ASP.NET Core Web API project is straightforward using the `Swashbuckle.AspNetCore` NuGet package. Once we install the package and register the required Swagger services and middleware components, Swagger automatically generates the API documentation, reducing the effort required to maintain separate documentation files.

`Swashbuckle.AspNetCore` consists of multiple components that can be used together or individually depending on your needs.

At its core,

- `Swashbuckle.AspNetCore.Swagger`
(<https://www.nuget.org/packages/Swashbuckle.AspNetCore.Swagger>): there's an OpenAPI generator
- `Swashbuckle.AspNetCore.SwaggerGen`
(<https://www.nuget.org/packages/Swashbuckle.AspNetCore.SwaggerGen>): middleware to expose OpenAPI (Swagger) documentation as JSON endpoints.
- `Swashbuckle.AspNetCore.SwaggerUI`
(<https://www.nuget.org/packages/Swashbuckle.AspNetCore.SwaggerUI>): a packaged version of the swagger-ui.

These three packages can be installed with the `Swashbuckle.AspNetCore` "metapackage"

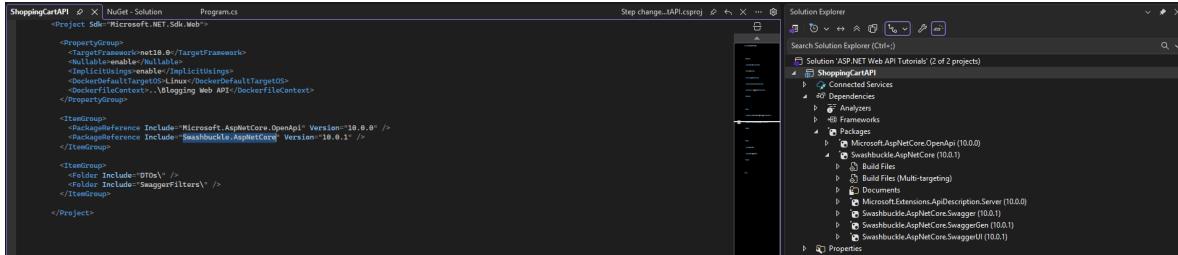
⚠ Version 10.0 of Swashbuckle.AspNetCore introduces breaking changes due to upgrading our dependency on Microsoft.OpenApi (<https://www.nuget.org/packages/Microsoft.OpenApi>) to version 2.x.x to add support for generating OpenAPI 3.1 documents.

Installation

install below packages

`Microsoft.AspNetCore.OpenApi`

`Swashbuckle.AspNetCore`



image_276.png

Configure Swagger Services:

Register AddSwaggerGen() service

```
builder.Services.AddSwaggerGen(options =>
{
    // Define API metadata (title, version, description, contact) shown
    // in Swagger UI
    options.SwaggerDoc("v1", new()
    {
        Title = "Shopping Cart API",
        Version = "v1",
        Description = "API for managing a shopping cart with discounts,
        tax, and delivery fee calculations.",
        Contact = new()
        {
            ...
        }
    })
});
```

```

        Name = "API Support",
        Email = "support@shoppingcart.com",
        Url = new Uri("https://www.shoppingcart.com/support")
    }
});

});

```

Enable Swagger only in Development environment for security, prevents exposing API structure in production.

```

if (app.Environment.IsDevelopment())
{
    // Enable middleware to serve generated Swagger as JSON endpoint
    app.UseSwagger();
    // Enable middleware to serve Swagger UI (HTML, JS, CSS, etc.)
    // Provides interactive documentation at /swagger
    app.UseSwaggerUI(options =>
    {
        // Configure the JSON endpoint for Swagger UI to consume
        options.SwaggerEndpoint("/swagger/v1/swagger.json", "Shopping
Cart API V1");
    });
}

```

Incase you have issues where your using productName and ProductName interchangeably, price and Price please add **.AddJsonOptions()** with **PropertyNameCaseInsensitive = true** as shown below

```

using ShoppingCartAPI.Services;
using ShoppingCartAPI.Services.Interfaces;

var builder = WebApplication.CreateBuilder(args);

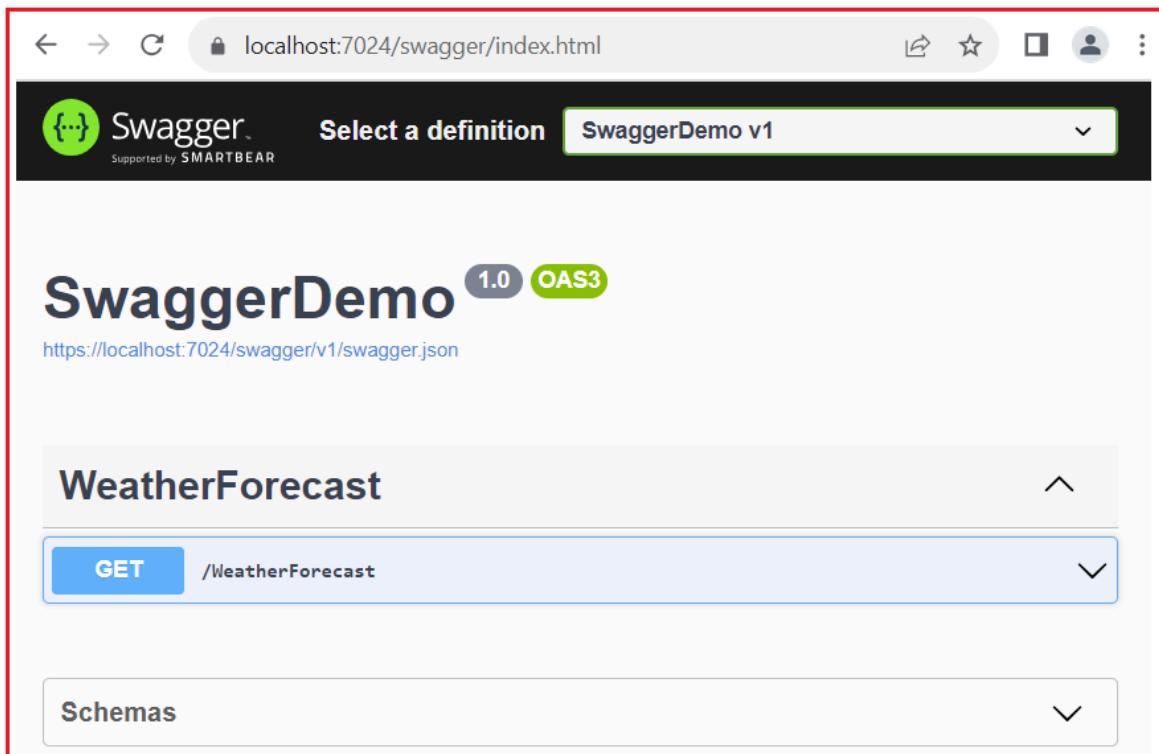
// Add services to the container.
builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {

```

```
options.JsonSerializerOptions.PropertyNameCaseInsensitive =  
true;  
});
```

Run the API:

Now, we need to Build and Run the project. Then, access the Swagger UI by navigating to the URL: `http://localhost:<port>/swagger`, as shown in the below image.



image_279.png

Customizing Swagger in ASP.NET Core Web API

Customizing Swagger in an ASP.NET Core Web API project can be essential in various scenarios to enhance documentation, usability, and security and meet the specific requirements of your API consumers. This can also include custom information, such as API descriptions and contact information, licenses, versions, etc. So, modify the `AddSwaggerGen` service as follows. As you can see here, we have provided more information about our APIs, such as the **Title**, **Version**, **Description**, **Terms of Service**, **Contact**, and **License**.

```

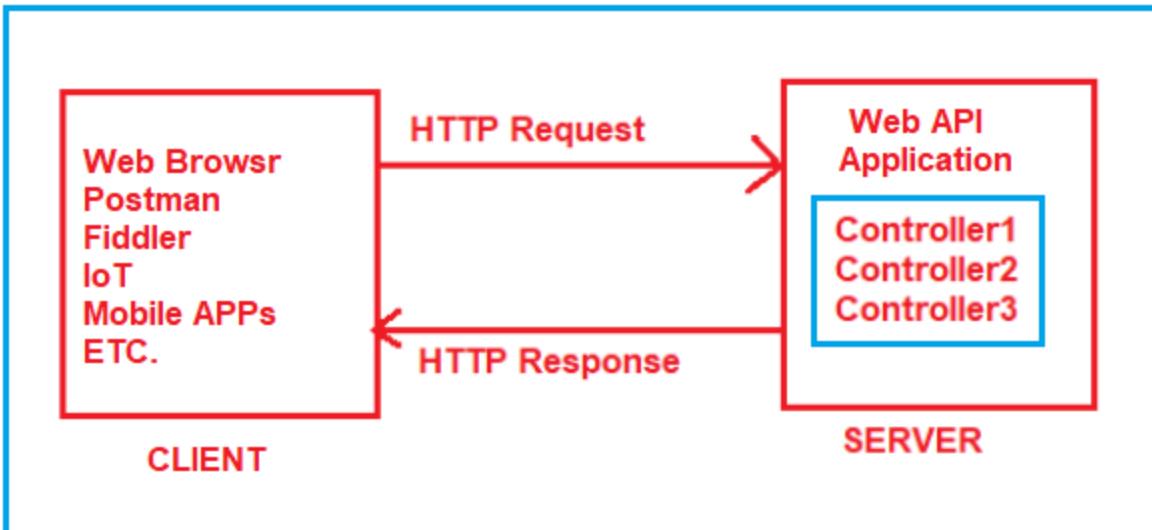
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("V10", new OpenApiInfo
    {
        Title = "My Custom API",
        Version = "V1",
        Description = "A Brief Description of My APIs",
        TermsOfService = new Uri("https://shoppingcart.net/privacy-
policy"),
        Contact = new OpenApiContact
        {
            Name = "Support",
            Email = "support@shoppingcart.net",
            Url = new Uri("https://shoppingcart.net/contact/")
        },
        License = new OpenApiLicense
        {
            Name = "Use Under XYZ",
            Url = new Uri("https://shoppingcart.net/about-us/")
        }
    });
});
});
```

By default, Swagger API looks for version V1, but we have changed the version to version V10 here. So, we also need to configure the same into the Swagger Middleware component. So, next, modify the UseSwaggerUI as follows:

```

app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/V10/swagger.json", "My API V10");
});
```

Routing



image_280.png

In an ASP.NET Core Web API application, **Routing** is a fundamental concept that connects the **incoming HTTP request** from a client to the correct **controller action method** within the server application. Let us understand routing in ASP.NET Core Web API with an example. Please have a look at the following image.

On the left-hand side, we have multiple clients, such as:

- A **Web Browser** (e.g., Chrome, Edge)
- **Postman** or **Fiddler** (API testing tools)
- **Swagger UI** (auto-generated API explorer)
- A **Mobile App** or a **Desktop Application** consuming your API

What Is Routing in ASP.NET Core and How Does It Work

In ASP.NET Core Web API, **Routing** is the process that maps incoming HTTP requests to the corresponding **Controller Actions**. Think of it as a **Traffic Director**.

In earlier versions of ASP.NET Core, developers had to call two middleware components: explicitly

- `UseRouting()` → to match the request to an endpoint
- `UseEndpoints()` → to execute the matched endpoint

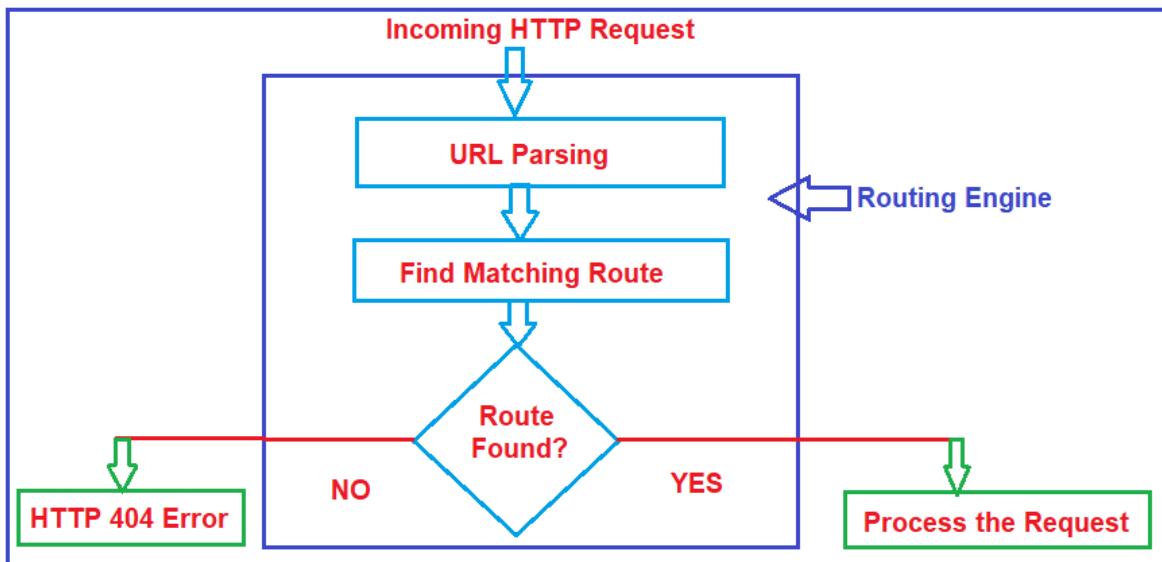
However, starting from **.NET 6 and later**, these are automatically configured when we use endpoint mapping methods such as: `app.MapControllers()`;

So, when we call the `app.MapControllers()`, the framework internally:

- Adds the **Routing Middleware** (`UseRouting()`) to match incoming requests to controller actions.
- Adds the **Endpoint Middleware** (`UseEndpoints()`) to invoke the matched controller action.

How Routing Works in ASP.NET Core

To understand how Routing works in ASP.NET Core, please have a look at the following diagram. Here, in the image below, the Routing Engine represents both Routing Middleware and Endpoint Middleware.



image_281.png

Incoming HTTP Request

When a client (such as a browser, Postman, Swagger UI, or mobile app) sends a request to the web server, it includes:

- **HTTP Method:** The type of action requested (GET, POST, PUT, DELETE, PATCH).
- **URL:** The target endpoint (e.g., `https://example.com/api/customers/5`).
- **Headers:** Metadata like Content-Type, Authorization, or Accept.
- **Query String (optional):** Additional parameters sent as part of the URL (e.g., `?search=abc`).
- **Request Body (optional):** Data sent with POST, PUT, or PATCH requests (e.g., JSON payload for creating or updating a resource).

Once received, the request flows through the ASP.NET Core **Middleware Pipeline**. One of the essential middleware components in this pipeline is the **Routing Middleware**, which starts the process of matching the request to an endpoint.

URL Parsing (Handled by Routing Middleware)

When the Routing Middleware is invoked, it first parses the incoming URL into its components. For example, consider the request:

- `https://example.com/api/customers/5?search=abc` This URL is broken down into:
 1. **Scheme:** https (communication protocol)
 2. **Host:** example.com (domain)
 3. **Path:** /api/customers/5 (endpoint path)
 4. **Query String:** Additional parameters if provided (e.g., `?search=abc`)

The **Routing Middleware** uses this path and HTTP method to find a matching endpoint among those registered in the application during startup.

Finding a Matching Endpoint

After the URL is parsed, the Routing Middleware (UseRouting) begins searching for a matching endpoint among all endpoints registered during application startup.

Endpoint Found or Not?

At this stage, the Routing Middleware has either identified a matching endpoint or found

none. The outcome determines the next step in the request pipeline.

Scenario 1: No Matching Endpoint (HTTP 404 Error)

- If no endpoint matches the request, the `HttpContext` remains without an assigned endpoint.
- The request continues through the remaining middleware, but since **Endpoint Middleware** has nothing to execute, the framework finally returns an HTTP 404 Not Found response to the client.
- This indicates that the requested URL or resource does not exist in the application.

Scenario 2: Matching Endpoint Found (Process the Request)

- When a valid endpoint is found, the **Routing Middleware** attaches the endpoint and extracted **Route Values** to the `HttpContext`.

Policy Middlewares Use the Selected Endpoint (Optional)

Once the Routing Middleware (`UseRouting`) has successfully matched the incoming request to a specific endpoint, ASP.NET Core allows several Policy Middlewares to act before the endpoint is executed.

These middlewares enforce cross-cutting concerns such as **Authorization**, **CORS**, and **Rate Limiting**, using the metadata already attached to the selected endpoint. These are often referred to as Policy Middlewares, because they apply policies defined in the endpoint's metadata.

Common Policy Middlewares Include:

`UseCors()`

- Applies Cross-Origin Resource Sharing rules.
- Checks whether the current request's origin, headers, and method are allowed by the configured CORS policy.
- Can be configured globally or per-endpoint using `[EnableCors]` or `[DisableCors]` attributes.

UseAuthentication()

- Authenticates the user based on **tokens, cookies, or credentials**.
- Populates the `HttpContext.User` property with the identity and claims if authentication succeeds.
- **Runs before authorization**, because authorization depends on the authenticated identity.

UseAuthorization()

- Uses endpoint metadata (like `[Authorize]` attributes or **authorization policies**) to decide whether **the current request is allowed to proceed**.
- If the user is not authorized, it short-circuits the pipeline with a **403 Forbidden** or **401 Unauthorized** response.
- If authorized, **the request continues to the next middleware**.

UseRateLimiter()

- Enforces rate-limiting policies attached to endpoints.
- Controls how many requests a client can make within a defined time window.

Route Parameters and Query Strings

ASP.NET Core provides two main ways to receive such data:

1. Route Data

2. Query Strings

Both mechanisms help your API capture values directly from the request URL, but they serve different purposes.

- **Route Data** is used when the value is **an essential part of the resource path**, for example, `/api/employees/5` directly identifies the employee with ID 5.

- **Query Strings**, on the other hand, are used when the values are **optional filters or modifiers**, for example, `/api/employees?department=HR&sortBy=name` retrieves employees in the HR department sorted by name.

In short:

- Use **Route Data** for mandatory and identity-based parameters (like **IDs**).
- Use **Query Strings** for **optional or filtering parameters** (like **search**, **filter**, or **sort options**).

Fetching Employee by ID using Route Data

Suppose we want to fetch one employee's details by their ID. The **ID value** can naturally be part of the URL, since it uniquely identifies a resource. So, we need to define one parameter to take the ID value within the method signature, as shown in the image below

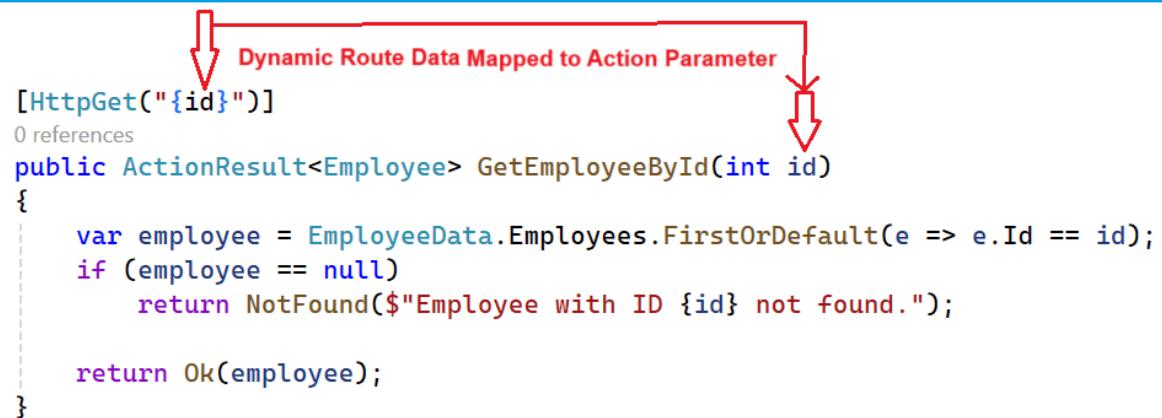
```
public ActionResult<Employee> GetEmployeeById(int id)
{
    var employee = EmployeeData.Employees.FirstOrDefault(e => e.Id == id);
    if (employee == null)
        return NotFound($"Employee with ID {id} not found.");

    return Ok(employee);
}
```

image_288.png

In ASP.NET Core Web API, if you want to pass anything as part of the URL Path (Route data), you need to use curly braces `{}`. Inside the curly braces, provide the same name as the method parameter.

In our example, the `GetEmployeeById` method takes the `id` parameter, so we need to pass the `id` within the curly braces of the Route or `HttpGet` attribute, as shown in the image below. Here, we are using `HttpGet`, but you can also use the `Route Attribute`.



image_289.png

So, please modify the EmployeeController class as shown below.

```
using Microsoft.AspNetCore.Mvc;
using RoutingInASPNETCoreWebAPI.Data;
using RoutingInASPNETCoreWebAPI.Models;

namespace RoutingInASPNETCoreWebAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class EmployeeController : ControllerBase
    {
        // GET api/Employee/1

        [HttpGet("{id}")]
        public ActionResult<Employee> GetEmployeeById(int id)
        {
            var employee = EmployeeData.Employees.FirstOrDefault(e =>
e.Id == id);
            if (employee == null)
                return NotFound($"Employee with ID {id} not found.");

            return Ok(employee);
        }
    }
}
```

```
    }  
}
```

- **Route Attribute:** [Route("api/[controller]")] → Base route (api/Employee) for all endpoints under EmployeeController.
- **Route Parameter:** [HttpGet("")]
- **Model Binding:** ASP.NET Core automatically binds the value from the URL to the id parameter in the method.
- **Action Method:** GetEmployeeById retrieves the employee with the specified ID from the in-memory data.

Handling Multiple Route Parameters in ASP.NET Core Web API:

In some cases, you may need to pass **multiple route parameters**, for instance, fetching employees by both Gender and City. Here, we want the Gender to be Male or Female and the city name to be specified. So, we will create an action method that takes two parameters, both of which are string types, as shown in the image below.

```
public ActionResult<IEnumerable<Employee>> GetEmployeesByGenderAndCity(string gender, string city)  
{  
    var filteredEmployees = EmployeeData.Employees  
        .Where(e => e.Gender.Equals(gender, StringComparison.OrdinalIgnoreCase) &&  
                  e.City.Equals(city, StringComparison.OrdinalIgnoreCase))  
        .ToList();  
  
    if (!filteredEmployees.Any())  
        return NotFound($"No employees found with Gender '{gender}' in City '{city}'.");  
  
    return Ok(filteredEmployees);  
}
```

image_290.png

Now we want to access the above GetEmployeesByGenderAndCity action method using the URL: api/Employee/Gender/Male/City/Los Angeles

Here, Male and Los Angeles are the dynamic values. So, we need to decorate the GetEmployeesByGenderAndCity method with the Route or HttpGet Attribute, as shown in the image below. Here, we are passing the gender and city parameters in curly braces.

```

[HttpGet("Gender/{gender}/City/{city}")]
0 references
public ActionResult<IEnumerable<Employee>> GetEmployeesByGenderAndCity(string gender, string city)
{
    var filteredEmployees = EmployeeData.Employees
        .Where(e => e.Gender.Equals(gender, StringComparison.OrdinalIgnoreCase) &&
                    e.City.Equals(city, StringComparison.OrdinalIgnoreCase))
        .ToList();

    if (!filteredEmployees.Any())
        return NotFound($"No employees found with Gender '{gender}' in City '{city}'.");

    return Ok(filteredEmployees);
}

```

image_291.png

Please modify the EmployeeController class as shown below.

```

using Microsoft.AspNetCore.Mvc;
using RoutingInASPNETCoreWebAPI.Data;
using RoutingInASPNETCoreWebAPI.Models;

namespace RoutingInASPNETCoreWebAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class EmployeeController : ControllerBase
    {
        // GET api/Employee/Gender/Male/City/Los Angeles

        [HttpGet("Gender/{gender}/City/{city}")]
        public ActionResult<IEnumerable<Employee>>
GetEmployeesByGenderAndCity(string gender, string city)
        {
            var filteredEmployees = EmployeeData.Employees
                .Where(e => e.Gender.Equals(gender,
StringComparison.OrdinalIgnoreCase) &&
                            e.City.Equals(city,
StringComparison.OrdinalIgnoreCase))
                .ToList();

            if (!filteredEmployees.Any())
                return NotFound($"No employees found with Gender

```

```

'{gender}' in City '{city}'.");
}

return Ok(filteredEmployees);
}
}
}

```

- **Route Template:** “Gender//City/” defines two dynamic route parameters: gender and city.
- **Action Method:** Filters employees based on the provided gender and city.
- **Case-insensitive Comparison:** Ensures the search is user-friendly regardless of input casing.

What Is a Query String?

Query Strings are key-value pairs appended to the URL after a question mark (?). Multiple query parameters are separated by &. They typically provide optional criteria or additional information and are best suited for:

- Filtering
- Searching
- Sorting
- Paging

Query Strings don't identify the resource itself but rather **modify the result** returned for that resource. You can specify as many optional query parameters as you want, in any order.

Let us understand the query string with an example. Now, we want to search employees by **department** without making it part of the route; we can use a **query string parameter**.

Here, we have created a method named SearchEmployee with one parameter called Department. Further notice, we have not included that parameter in the Route Attribute.

```

using Microsoft.AspNetCore.Mvc;
using RoutingInASPNETCoreWebAPI.Data;

```

```

using RoutingInASPNETCoreWebAPI.Models;

namespace RoutingInASPNETCoreWebAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class EmployeeController : ControllerBase
    {
        // GET api/Employee/Search?Department=HR
        [HttpGet("Search")]
        public ActionResult<IEnumerable<Employee>>
        SearchEmployees([FromQuery] string department)
        {
            var filteredEmployees = EmployeeData.Employees
                .Where(e => e.Department.Equals(department,
StringComparison.OrdinalIgnoreCase))
                .ToList();

            if (!filteredEmployees.Any())
                return NotFound($"No employees found in Department
'{department}'.");

            return Ok(filteredEmployees);
        }
    }
}

```

- **Route Attribute:** `[HttpGet("Search")]` sets the endpoint to `api/Employee/Search`.
- **Query Parameter:** `[FromQuery]` tells ASP.NET Core to bind the department from the query string. Using the `[FromQuery]` attribute here is optional, as by default, value type parameters are bound from the Query string.
- **Action Method:** Filters employees based on the provided department. If no employee matches, it returns a 404 with a descriptive message.

Multiple Query String Parameters in ASP.NET Core Web API

In real-world scenarios, search functionalities often require multiple optional parameters to filter data. Let's say we want to filter employees by **City**, **Gender**, and **Department**. In that case, our action method should accept three parameters. So, modify the Employee Controller as follows. If you want to make a query string parameter optional, you need to use ? or initialize the parameter with a default value.

```
using Microsoft.AspNetCore.Mvc;
using RoutingInASPNETCoreWebAPI.Data;
using RoutingInASPNETCoreWebAPI.Models;

namespace RoutingInASPNETCoreWebAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class EmployeeController : ControllerBase
    {
        // GET api/Employee/Search?Gender=Male&Department=IT&City=Los
        Angeles
        [HttpGet("Search")]
        public ActionResult<IEnumerable<Employee>>
        SearchEmployees([FromQuery] string? gender, [FromQuery] string?
        department, [FromQuery] string? city)
        {
            var filteredEmployees =
            EmployeeData.Employees.AsQueryable();

            if (!string.IsNullOrEmpty(gender))
                filteredEmployees = filteredEmployees.Where(e =>
e.Gender.Equals(gender, StringComparison.OrdinalIgnoreCase));

            if (!string.IsNullOrEmpty(department))
                filteredEmployees = filteredEmployees.Where(e =>
e.Department.Equals(department, StringComparison.OrdinalIgnoreCase));

            if (!string.IsNullOrEmpty(city))
                filteredEmployees = filteredEmployees.Where(e =>
e.City.Equals(city, StringComparison.OrdinalIgnoreCase));
        }
    }
}
```

```

        var result = filteredEmployees.ToList();

        if (!result.Any())
            return NotFound("No employees match the provided search
criteria.");

        return Ok(result);
    }
}
}

```

- **Optional Parameters:** The ? in string? makes the parameter optional.
- **Dynamic Filtering:** Filters are applied only when values are provided.
- **Parameter Order:** The order of query parameters in the URL does not matter.

Testing the Endpoint

Filter by Gender Only: Let's say we want to filter the employees by Gender only. Then, you can only pass the Gender query string parameter in the URL (`/api/Employee/Search?Gender=Male`), as shown in the image below.



```
[  
  {  
    "id": 2,  
    "name": "Bob Smith",  
    "gender": "Male",  
    "department": "IT",  
    "city": "Los Angeles"  
  },  
  {  
    "id": 3,  
    "name": "Charlie Davis",  
    "gender": "Male",  
    "department": "Finance",  
    "city": "Chicago"  
  },  
  {  
    "id": 5,  
    "name": "James Smith",  
    "gender": "Male",  
    "department": "IT",  
    "city": "Chicago"  
  }  
]
```

image_292.png

Filter by Gender and Department: Now, if you want to search employees by **Gender** and **Department**, then you need to pass two query string parameters in the URL (`/api/Employee/Search?Gender=Male&Department=IT`), as shown in the image below.



image_293.png

Combining Route Parameters and Query Strings in ASP.NET Core Web API

ASP.NET Core Web API allows combining both **Route Parameters** and **Query Strings** for maximum flexibility within the same endpoint. Let's consider an example where we have an API endpoint to get employee details with the following criteria:

- **Route Parameter (mandatory):** Gender (e.g., Male, Female) identifies a subset of employees.
- **Query Strings (optional):** Department and City provide additional filtering within the specified gender.

So, modify the **Employee Controller** as follows:

```
using Microsoft.AspNetCore.Mvc;  
using RoutingInASPNETCoreWebAPI.Data;  
using RoutingInASPNETCoreWebAPI.Models;  
  
namespace RoutingInASPNETCoreWebAPI.Controllers  
{  
    [ApiController]  
    [Route("api/[controller]")]
```

```

public class EmployeeController : ControllerBase
{
    // GET api/Employee/Gender/Male?Department=IT&City=Los Angeles
    [HttpGet("Gender/{gender}")]
    public ActionResult<IEnumerable<Employee>>
    GetEmployeesByGender([FromRoute] string gender, [FromQuery] string?
    department, [FromQuery] string? city)
    {
        var filteredEmployees = EmployeeData.Employees
            .Where(e => e.Gender.Equals(gender,
StringComparison.OrdinalIgnoreCase));

        if (!string.IsNullOrEmpty(department))
            filteredEmployees = filteredEmployees.Where(e =>
e.Department.Equals(department, StringComparison.OrdinalIgnoreCase));

        if (!string.IsNullOrEmpty(city))
            filteredEmployees = filteredEmployees.Where(e =>
e.City.Equals(city, StringComparison.OrdinalIgnoreCase));

        var result = filteredEmployees.ToList();

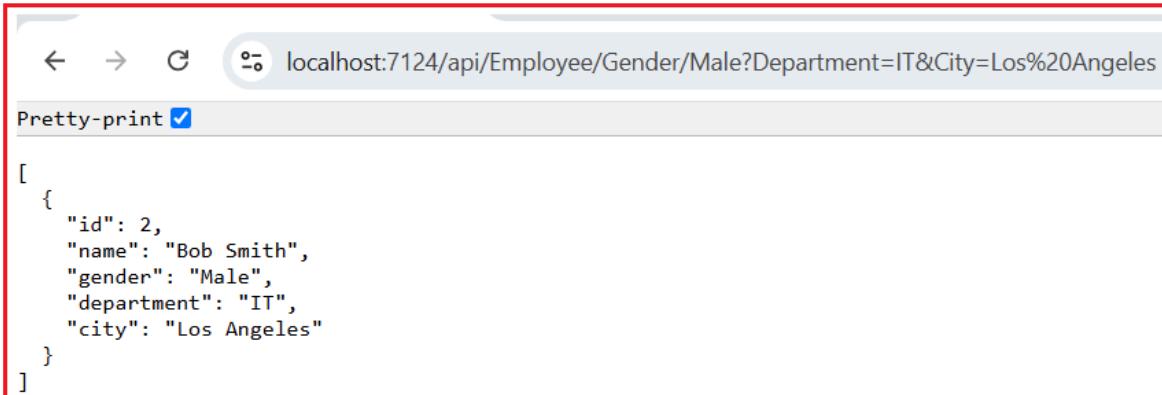
        if (!result.Any())
            return NotFound("No employees match the provided search
criteria.");

        return Ok(result);
    }
}

```

- **Route Parameter:** {gender} is part of the route path, identifying a subset of employees.
- **Query Strings:** department and city provide additional optional filtering.

Now, with the above changes in place, run the application and search for employees by Gender as Route Data, Department, and City as query strings in the URL (**/api/Employee/Gender/Male?Department=IT&City=Los Angeles**). You should get the following result.



A screenshot of a web browser window with a red border around the content area. The address bar shows the URL: `localhost:7124/api/Employee/Gender/Male?Department=IT&City=Los%20Angeles`. Below the address bar, there is a "Pretty-print" checkbox which is checked. The main content area displays a JSON array with one element:

```
[  
  {  
    "id": 2,  
    "name": "Bob Smith",  
    "gender": "Male",  
    "department": "IT",  
    "city": "Los Angeles"  
}]
```

image_294.png

Action Return Types in ASP.NET

When developing **ASP.NET Core Web APIs**, every controller action we write must return some result to the client, maybe a simple value, a complex object, or an HTTP response message. This “result” that our action returns is known as the **Return Type**.

Categories of Return Types

In ASP.NET Core Web API, we can return data in multiple ways, depending on the level of control and type safety we need. These return types fall into three main categories:

Specific Types (e.g., `string`, `int`, `object`, `Product`)

These are direct return types. ASP.NET Core automatically assumes a successful request and sends back HTTP 200 OK.

- If a reference type returns null, the framework sends 204 No Content.
- This approach is simple but lacks flexibility for real-world error handling.

`IActionResult` / `ActionResult`

These return types give us complete control over the HTTP response. We can manually decide:

- The status code (e.g., 200, 404, 400),
- The response message,
- And even attach headers.

You can return various predefined results, such as:

- `Ok (object)` → 200 OK
- `NotFound (string)` → 404 Not Found
- `BadRequest (string)` → 400 Bad Request
- `CreatedAtAction ()` → 201 Created

This is perfect when you want to return different outcomes (success, error, not found, etc.) within the same method. It is used heavily in real-world REST APIs.

ActionResult<T>

This is a **Generic Return Type** that merges the benefits of both previous categories. It is introduced in ASP.NET Core 2.1, which combines:

- **Strong Typing** (Swagger knows exactly what T is)
- **Flexible HTTP Control** (you can still return `Ok()`, `NotFound()`, `BadRequest()`)

Hence, it's considered the **best practice for modern APIs**.

Asynchronous Return Types

All the above can be combined with `Task` or `Task<T>` for asynchronous programming:

- `Task<int>`
- `Task<ActionResult>` or `Task<ActionResult>`
- `Task<ActionResult<Product>>`

This is important for **non-blocking operations**, like database queries or external API calls, which improve scalability and performance by freeing up threads during I/O-bound operations.

Examples to Understand Action Return Types in ASP.NET Core Web API

To understand how these return types work in practice, let's create a **Product Management API** that mimics an online store like Amazon or Flipkart. Instead of using a database, we will use **in-memory data for simplicity**. This API demonstrates how different return types behave in real scenarios, such as when we:

- Fetch the count of all products,
- Retrieve product details by ID,
- Get all product names, or

- Fetch a complete product list.

Each of these examples shows how **Return Types** change the API's Response Behavior.

Create a new project.

Open Visual Studio and create a new **ASP.NET Core Web API** project named **ReturnTypeDemo**. Inside the root directory, create two folders:

- **Models** — to store data structures (like `Product.cs`)
- **Services** — to contain business logic (`ProductService.cs`)

This folder structure separates concerns: **Models handle data**, while **Services handle logic**, making the project more modular and easier to maintain.

Creating Product Model

The **Product** class represents an individual product in our e-commerce catalog. Each property in this class defines a **column-like structure** that would typically correspond to a database table in a real-world application.

This class serves as a **data transfer model**, bridging data between the service and controller layers. When a controller returns a **Product**, it's automatically serialized into JSON for the client. So, add a class file named **Product.cs** within the **Models** folder

```
namespace ReturnTypeDemo.Models
{
    // Represents a Product entity in our in-memory system
    public class Product
    {
        public int Id { get; set; }           // Unique product
        identifier
        public string Name { get; set; }      // Product name (e.g.,
        "HP Laptop")
        public string Category { get; set; }   // Category (e.g.,
        "Electronics", "Clothing")
        public double Price { get; set; }      // Price in INR
        public bool InStock { get; set; }       // Availability flag
    }
}
```

```
    }  
}
```

- **Id:** Unique identifier for each product (like a ProductID in a table).
- **Name:** The product's display name.
- **Category:** Logical grouping (e.g., Electronics, Clothing).
- **Price:** The product's cost in Indian Rupees (INR).
- **InStock:** A boolean flag indicating whether the product is currently available.

Creating Product Service

The **ProductService** class acts as the **Business Logic Layer** or **Data Service Layer**. Instead of connecting to an actual database, it uses a static in-memory `List<Product>` that simulates a data table. This layer decouples business logic from the controller, meaning the controller focuses on **HTTP Handling**, while the service handles **Data Retrieval Logic**.

In production APIs, such services would interact with:

- Databases using EF Core,
- External APIs,
- Or caching systems like Redis.

So, add a class file named `ProductService.cs` within the `Services` folder. It's a static class that stores a few hard-coded `Product` objects and provides methods to fetch or filter them.

```
using ReturnTypeDemo.Models;  
namespace ReturnTypeDemo.Services  
{  
    public static class ProductService  
    {  
        // In-memory product list simulating a database table  
        private static readonly List<Product> _products = new()  
        {  
            new Product { Id = 1, Name = "HP Laptop", Category =
```

```

    "Electronics", Price = 55000, InStock = true },
        new Product { Id = 2, Name = "iPhone 15", Category =
"Mobiles", Price = 125000, InStock = true },
            new Product { Id = 3, Name = "Samsung TV", Category =
"Electronics", Price = 78000, InStock = false },
                new Product { Id = 4, Name = "Nike Shoes", Category =
"Footwear", Price = 8500, InStock = true },
                    new Product { Id = 5, Name = "Levi's Jeans", Category =
"Clothing", Price = 4500, InStock = true }
    };

    // Returns the total number of products.
    // Demonstrates returning a primitive type(int).
    public static async Task<int> GetProductCountAsync()
    {
        await Task.Delay(300); // Simulate async DB call
        return _products.Count;
    }

    // Returns all available products.
    // Demonstrates returning a collection of complex
types(List<Product>).
    public static async Task<List<Product>> GetAllProductsAsync()
    {
        await Task.Delay(400); // Simulate async DB call
        return _products;
    }

    // Searches for a single product by ID.
    // Returns a single complex type or null if not found.
    public static async Task<Product?> GetProductByIdAsync(int id)
    {
        await Task.Delay(300); // Simulate async DB call
        return _products.FirstOrDefault(p => p.Id == id);
    }

    // Returns only product names as strings.
    // Demonstrates returning a collection of primitive

```

```

types(List<string>).
    public static async Task<List<string>> GetAllProductNamesAsync()
    {
        await Task.Delay(250); // Simulate async DB call
        return _products.Select(p => p.Name).ToList();
    }
}
}

```

- The static `_products` list holds a predefined set of product data. This makes it easier to test the API without requiring a database.
- Each method in this service is marked as `async` and uses `Task.Delay()` to simulate I/O latency, mimicking how real database calls behave asynchronously.

Controller Action Returning Primitive & Complex Types

Add an empty Web API Controller named **ProductController** within the **Controllers** folder. The controller consumes `ProductService` and exposes endpoints demonstrating different return types.

```

using Microsoft.AspNetCore.Mvc;
using ReturnTypeDemo.Models;
using ReturnTypeDemo.Services;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ReturnTypeDemo.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ProductController : ControllerBase
    {
        // Returning Primitive Type - Product Count
        // Example: GET /api/product/GetProductCount
        // Returns an integer (number of products)
        [HttpGet("GetProductCount")]
        public async Task<int> GetProductCount()

```

```

    {
        return await ProductService.GetProductCountAsync();
    }

    // Returning Complex Type - Single Product
    // Example: GET /api/product/GetProductById/2
    // Returns a product object (complex type)
    [HttpGet("GetProductById/{id}")]
    public async Task<Product?> GetProductById(int id)
    {
        return await ProductService.GetProductByIdAsync(id);
    }

    // Returning Collection of Complex Types - List<Product>
    // Example: GET /api/product GetAllProducts
    // Returns list of products (complex type collection)
    [HttpGet("GetAllProducts")]
    public async Task<List<Product>> GetAllProducts()
    {
        return await ProductService.GetAllProductsAsync();
    }

    // Returning Collection of Primitive Types - List<string>
    // Example: GET /api/product/GetAllProductNames
    // Returns list of product names only (primitive type
collection)
    [HttpGet("GetAllProductNames")]
    public async Task<List<string>> GetAllProductNames()
    {
        return await ProductService.GetAllProductNamesAsync();
    }
}

```

Returning IActionResult Return Type

IActionResult is an interface that represents the result of an action method in ASP.NET Core Web API. It is a key part of **flexible response handling**, allowing you to have

complete control over the HTTP response, including:

- **Status Codes** (e.g., 200 OK, 404 Not Found, 400 Bad Request, 201 Created).
- **Response Body** (e.g., the data or message you want to send).
- **Headers** (e.g., metadata, ETag, or caching headers).

This makes it far more **flexible** and **expressive** than returning a primitive or complex type directly. Instead of simply letting ASP.NET Core assume a successful response (200 OK), you can explicitly choose the most appropriate status code and response message depending on what happened in your logic.

For example:

- If the operation succeeded → return Ok(data)
- If the input is invalid → return BadRequest("Invalid input")
- If no resource was found → return NotFound("Item not found")
- If no data needs to be sent → return NoContent()
- If a resource was created → return CreatedAtAction("GetById", new , data)

When to Use this Return Type?

When you use simple return types (like int or Product), ASP.NET Core automatically assumes everything is fine and sends back 200 OK. However, real-world APIs are rarely that simple — things go wrong all the time!

Imagine these scenarios:

- The client requests a product ID that doesn't exist → should return **404 Not Found**.
- The client sends malformed JSON → should return **400 Bad Request**.
- The server encounters an unexpected exception → should return **500 Internal Server Error**.
- A new product is successfully created → should return **201 Created**.

⚠ That's where `IActionResult` becomes invaluable. It allows your controller to adapt the response to the situation.

Use `IActionResult` When:

- **Multiple Outcomes Are Possible:** Return 200 when data exists, and 404 when it doesn't. Example: fetching a product by ID.
- **Error Handling Is Important:** You want to handle and communicate validation errors or invalid requests clearly.
- **Custom HTTP Codes Are Needed:** You need to return non-standard responses, such as 201 Created, 202 Accepted, or 500 Internal Server Error.
- **POST, PUT, DELETE Operations:** These actions often return conditional results, success, validation error, or conflict, that require explicit status codes.
- **Consistency in API Behavior:** Using `IActionResult` ensures that your API speaks the universal HTTP “language” that frontend developers and tools (like Postman, Swagger, or Angular services) expect.

Rewriting Our Example Using `IActionResult`

Let's rewrite the previous **Product Management API** using the `IActionResult` return type. We will keep:

- The same **Product Model**
- The same **ProductService** (static async in-memory)
- But rewrite the **Controller** to use `IActionResult` for each endpoint.

So, please modify the Product Controller as follows:

```
using Microsoft.AspNetCore.Mvc;
using ReturnTypeDemo.Services;

namespace ReturnTypeDemo.Controllers
{
```

```

[ApiController]
[Route("api/[controller]")]
public class ProductController : ControllerBase
{
    // Get Total Product Count (Primitive Type wrapped in Ok)
    // Example: GET /api/product/GetProductCount
    // Returns product count with HTTP 200 OK
    [HttpGet("GetProductCount")]
    public async Task<IActionResult> GetProductCount()
    {
        int count = await ProductService.GetProductCountAsync();

        // Always return Ok() to wrap primitive result
        return Ok(count);
    }

    // Get Product by Id (Complex Type)
    // Example: GET /api/product/GetProductById/2
    // Returns product if found; otherwise returns 404 Not Found
    [HttpGet("GetProductById/{id}")]
    public async Task<IActionResult> GetProductById(int id)
    {
        var product = await ProductService.GetProductByIdAsync(id);

        if (product == null)
            return NotFound($"Product with ID = {id} was not
found.");

        return Ok(product);
    }

    // Get All Products (Collection of Complex Types)
    // Example: GET /api/product/GetAllProducts
    // Returns list of products (HTTP 200 OK)
    [HttpGet("GetAllProducts")]
    public async Task<IActionResult> GetAllProducts()
    {
        var products = await ProductService.GetAllProductsAsync();
    }
}

```

```

        if (products == null || products.Count == 0)
            return NoContent(); // 204 No Content if list is empty

        return Ok(products);
    }

    // Get All Product Names (Collection of Primitive Types)
    // Example: GET /api/product/GetAllProductNames
    // Returns product names; if none, returns 404 Not Found
    [HttpGet("GetAllProductNames")]
    public async Task<IActionResult> GetAllProductNames()
    {
        var names = await ProductService.GetAllProductNamesAsync();

        if (names == null || names.Count == 0)
            return NotFound("No product names found.");

        return Ok(names);
    }

    // Get Product Details (Custom Error Example)
    // Example: GET /api/product/GetProductDetails/99
    // Demonstrates returning BadRequest & NotFound
    [HttpGet("GetProductDetails/{id}")]
    public async Task<IActionResult> GetProductDetails(int id)
    {
        if (id <= 0)
            return BadRequest("Invalid product ID. Must be greater
than zero.");

        var product = await ProductService.GetProductByIdAsync(id);

        if (product == null)
            return NotFound($"Product with ID = {id} not found.");

        return Ok(product);
    }
}

```

```
}
```

Returning ActionResult Return Type (Non-Generic)

ActionResult is a **Non-Generic Concrete Class** in ASP.NET Core that represents the result of an action method. It's similar in purpose to IActionResult, but unlike IActionResult (which is an interface), ActionResult is a Base Class for many built-in result types, such as OkResult, BadRequestResult, NotFoundResult, and so on.

When to Use ActionResult

You should use **ActionResult (non-generic)** when your API method's primary purpose is to **perform an action** (create, update, delete) rather than **return a dataset**. Let's explore each case.

When You Don't Need to Return Typed Data If your method only needs to indicate whether an operation succeeded or failed, you can use ActionResult. Example:

- return Ok(); // Operation succeeded
- return BadRequest(); // Input was invalid
- return NotFound(); // Resource missing

There's no need to return `ActionResult<Product>` or `ActionResult<List<Product>>` because the purpose isn't to send data — it's to report the outcome.

For POST, PUT, or DELETE Endpoints

Operations that modify data often have **binary outcomes** — they either succeed or fail:

- POST → create a new record
- PUT → update an existing record
- DELETE → remove an item

In these cases, you usually return:

- 201 Created or 200 OK on success,
- 400 Bad Request if the input was invalid,
- 404 Not Found if the resource doesn't exist.

Returning a simple ActionResult allows you to send these HTTP codes cleanly, without worrying about the specific return type of the data.

Rewrite the Product Management API using ActionResult.

In this version of the **Product Management API**, each endpoint returns ActionResult instead of IActionResult. While the behavior is similar to IActionResult, this version makes it clearer that the method is returning **standardized HTTP responses** rather than arbitrary results. So, please modify the **ProductController** as follows:

```
using Microsoft.AspNetCore.Mvc;
using ReturnTypeDemo.Services;

namespace ReturnTypeDemo.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ProductController : ControllerBase
    {
        // -----
        // Get Total Product Count (Primitive Type wrapped with Ok)
        // -----
        // Example: GET /api/product/GetProductCount
        [HttpGet("GetProductCount")]
        public async Task<ActionResult> GetProductCount()
        {
            int count = await ProductService.GetProductCountAsync();
            return Ok(count); // 200 OK with primitive data
        }

        // -----
    }
}
```

```
-----  
    // Get Product by Id (Complex Type)  
    // -----  
  
-----  
    // Example: GET /api/product/GetProductById/2  
    [HttpGet("GetProductById/{id}")]  
    public async Task<ActionResult> GetProductById(int id)  
    {  
        if (id <= 0)  
            return BadRequest("Invalid Product ID. Must be greater  
than zero.");  
  
        var product = await ProductService.GetProductByIdAsync(id);  
  
        if (product == null)  
            return NotFound($"Product with ID = {id} not found.");  
  
        return Ok(product); // 200 OK with object  
    }  
  
    // -----  
-----  
    // Get All Products (Collection of Complex Types)  
    // -----  
  
-----  
    // Example: GET /api/product/GetAllProducts  
    [HttpGet("GetAllProducts")]  
    public async Task<ActionResult> GetAllProducts()  
    {  
        var products = await ProductService.GetAllProductsAsync();  
  
        if (products == null || products.Count == 0)  
            return NoContent(); // 204 No Content  
  
        return Ok(products); // 200 OK with JSON list  
    }  
  
    // -----
```

```
-----  
// Get All Product Names (Collection of Primitive Types)  
// -----  
  
-----  
// Example: GET /api/product/GetAllProductNames  
[HttpGet("GetAllProductNames")]  
public async Task<ActionResult> GetAllProductNames()  
{  
    var names = await ProductService.GetAllProductNamesAsync();  
  
    if (names == null || names.Count == 0)  
        return NotFound("No product names found.");  
  
    return Ok(names); // 200 OK with primitive list  
}  
  
// -----  
-----  
// Demonstration of Multiple Status Codes  
// -----  
  
-----  
// Example: GET /api/product/GetProductDetails/0  
// Shows how to send 400, 404, or 200 based on conditions.  
[HttpGet("GetProductDetails/{id}")]  
public async Task<ActionResult> GetProductDetails(int id)  
{  
    if (id <= 0)  
        return BadRequest("Invalid product ID.");  
  
    var product = await ProductService.GetProductByIdAsync(id);  
  
    if (product == null)  
        return NotFound($"Product with ID = {id} not found.");  
  
    return Ok(product);  
}
```

```
    }  
}
```

Limitations of ActionResult Although ActionResult is powerful, it has several drawbacks compared to ActionResult<T>, especially when your API returns typed data frequently.

Model Binding

Model Binding in ASP.NET Core Web API is the process that automatically maps incoming HTTP request data, from routes, query strings, headers, forms, or request body, to controller action parameters or model objects.

Instead of manually reading and parsing values from `HttpContext.Request`, the framework intelligently handles this for you, converting raw input into strongly-typed .NET objects. This makes your code cleaner, safer, and easier to maintain by removing repetitive parsing logic and ensuring consistent validation across all API endpoints.

This feature abstracts the complexities of extracting and converting raw request data from `HttpRequest` into strongly-typed .NET objects. ASP.NET Core uses model binding to map data from multiple sources:

- **Query Strings:** Parameters appended to the URL.
- **Route Data:** Parameters defined in the URL path.
- **Request Body:** Payload data, often in JSON or XML format (commonly for POST, PUT, or PATCH requests).
- **Headers:** Custom data sent within HTTP headers.



Key Idea: Model binding converts raw request data (text, JSON, form, etc.) into .NET objects, making APIs cleaner, type-safe, and easy to maintain. So, you don't need to parse request data manually; the framework does it for you.

What are the Model Binding Techniques Used in ASP.NET Core Web API?

ASP.NET Core provides several techniques for model binding depending on the **source of the data**. You can explicitly tell the framework where to look for data using specific attributes.

[FromRoute] – Binding from Route Parameters

The `[FromRoute]` is used when data is part of the URL Route itself. The model binder reads the value from the route template defined in your controller's endpoint. It's commonly used for identifying a resource by its unique **ID**, such as `/api/products/5`.

Syntax:

```
[HttpGet("products/{id}")]  
public IActionResult GetProduct([FromRoute] int id)
```

Use Cases:

- When your API endpoint includes dynamic route segments.
- To extract resource identifiers like `userId`, `orderId`, or `productId`.
- When designing RESTful URLs, such as `/api/users/` or `/api/orders/`.

[FromQuery] – Binding from Query String

The `[FromQuery]` tells ASP.NET Core to **bind data from the Query String Parameters appended to the URL**. It's best suited for **optional parameters**, **filters**, **pagination**, or **sorting options** where you want to keep the URL readable and flexible.

Syntax:

```
[HttpGet("products")]  
public IActionResult SearchProducts([FromQuery] string category,  
[FromQuery] int page = 1)
```

Use Cases:

- Implementing **filtering**, **sorting**, or **pagination** (`/api/products?category=Electronics&page=2`).
- Passing optional parameters without changing the route structure.
- **Searching**, **filtering**, or **listing data dynamically**.

[FromBody] – Binding from Request Body

The `[FromBody]` attribute is used when data is sent in the HTTP Request Body, usually as JSON or XML. ASP.NET Core uses input formatters (like JSON or XML) to deserialize the body into a strongly-typed object. Only one parameter per action can use `[FromBody]` because the request body can be read only once.

Syntax:

```
[HttpPost("add")]
public IActionResult AddProduct([FromBody] ProductDTO product)
```

Use Cases:

- When sending complex objects or large data in POST/PUT requests.
- For creating or updating records (e.g., new user, new product).
- When consuming APIs with JSON request bodies

[FromHeader] – Binding from Request Headers

`[FromHeader]` binds values from specific HTTP request headers, such as **Authorization**, **Accept-Language**, or **custom headers** used for security or tracking.

Syntax:

```
[HttpGet("userinfo")]
public IActionResult GetUser([FromHeader(Name = "X-User-Id")] string
userId)
```

Use Cases:

- Reading **authentication tokens**, **version information**, or **API keys from headers**.
- Custom tracking IDs or correlation IDs.
- Language or localization preferences via headers.

[FromServices] – Binding from Dependency Injection Container

The `[FromServices]` attribute tells ASP.NET Core to resolve a parameter from the **Dependency Injection (DI) Container instead of the request**. It's not about client-

supplied data; it's about injecting services (e.g., **repositories**, **loggers**, **configuration**). That means **[FromServices]** allows us to inject Services Registered in the DI Container directly into an action method. It's ideal when we need a service just for that specific action without adding it to the constructor.

Syntax:

```
[HttpGet("config")]
public IActionResult GetConfig([FromServices] IConfiguration config)
{
    var env = config["ASPNETCORE_ENVIRONMENT"];
    return Ok(env);
}
```

Use Cases:

- Accessing a lightweight or specialized service in one controller method.
- Avoiding constructor clutter for infrequently used services.
- Logging, Caching, or health-check services.

Manual Data Reading in ASP.NET Core Web API (Without Model Binding)

We will create a simple in-memory Product Management API that performs operations like reading product details, adding new products, and checking system health, all by manually extracting and validating request data. Clients can:

1. Retrieve product details by route.
2. Search products by query parameters.
3. Fetch all active products using a header-based authentication key.
4. Add a new product via a JSON request body.
5. Apply a discount to a product (combination of route, query, and header).

Everything is handled manually, without model binding or [ApiController].

Create the Project

First, create a new ASP.NET Core Web API Project, name it ModelBindingDemo.

Create an In-Memory Product Store

First, create 2 folders named Models and Services in the Project root directory. Inside the Models folder, create a class file named Product.cs.

```
namespace ModelBindingDemo.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public decimal Price { get; set; }
        public string? Category { get; set; }
        public bool IsActive { get; set; }
        public int Stock { get; set; }
        public double Rating { get; set; }
        public DateTime CreatedDate { get; set; }
    }
}
```

Creating Product Service Interface

Create a class file named IProductService.cs within the Services folder.

```
using ModelBindingDemo.Models;
namespace ModelBindingDemo.Services
{
    public interface IProductService
    {
        IEnumerable<Product> GetAll();
        Product? GetById(int id);
        IEnumerable<Product> Search(string? category, decimal? minPrice,
decimal? maxPrice);
        void Add(Product product);
    }
}
```

```
        bool UpdatePrice(int id, decimal discountPercent);  
    }  
}
```

Creating Product Service Implementation

Create a class file named ProductService.cs within the Services folder, and then copy and paste the following code.

```
using ModelBindingDemo.Models;  
  
namespace ModelBindingDemo.Services  
{  
    public class ProductService : IProductService  
    {  
        private readonly List<Product> _products = new()  
        {  
            new Product { Id = 1, Name = "Laptop", Price = 65000,  
Category = "Electronics", IsActive = true, Stock = 20, Rating = 4.5,  
CreatedDate = DateTime.Now.AddDays(-10) },  
            new Product { Id = 2, Name = "Headphones", Price = 2500,  
Category = "Audio", IsActive = true, Stock = 50, Rating = 4.2,  
CreatedDate = DateTime.Now.AddDays(-5) },  
            new Product { Id = 3, Name = "Smartwatch", Price = 12000,  
Category = "Wearables", IsActive = false, Stock = 10, Rating = 3.9,  
CreatedDate = DateTime.Now.AddDays(-15) },  
            new Product { Id = 4, Name = "Keyboard", Price = 1500,  
Category = "Accessories", IsActive = true, Stock = 35, Rating = 4.1,  
CreatedDate = DateTime.Now.AddDays(-2) }  
        };  
  
        public IEnumerable<Product> GetAll()  
        {  
            return _products;  
        }  
  
        public Product? GetById(int id)  
        {
```

```

        return _products.FirstOrDefault(p => p.Id == id);
    }

    public IEnumerable<Product> Search(string? category, decimal?
minPrice, decimal? maxPrice)
    {
        var query = _products.AsQueryable();
        if (!string.IsNullOrWhiteSpace(category))
            query = query.Where(p => p.Category!.Equals(category,
 StringComparison.OrdinalIgnoreCase));
        if (minPrice.HasValue)
            query = query.Where(p => p.Price >= minPrice.Value);
        if (maxPrice.HasValue)
            query = query.Where(p => p.Price <= maxPrice.Value);

        return query.ToList();
    }

    public void Add(Product product)
    {
        product.Id = _products.Max(p => p.Id) + 1;
        product.CreatedDate = DateTime.Now;
        _products.Add(product);
    }

    public bool UpdatePrice(int id, decimal discountPercent)
    {
        var product = _products.FirstOrDefault(p => p.Id == id);
        if (product == null) return false;

        var discountAmount = product.Price * discountPercent / 100;
        product.Price -= discountAmount;
        return true;
    }
}

```

Register Service in Program.cs: Please add the following statement inside the Program class file.

```
builder.Services.AddSingleton<IPIService, ProductService>();
```

Create the Controller – Manual Data Extraction

Create an API Empty Controller named **ProductsController** within the Controllers folder.

```
using Microsoft.AspNetCore.Mvc;
using ModelBindingDemo.Models;
using ModelBindingDemo.Services;
using System.Text.Json;

namespace ModelBindingDemo.Controllers
{
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase
    {
        private readonly ILogger<ProductsController> _logger;
        private readonly IPIService _productService;

        // Constructor injection for logger and service
        public ProductsController(IPIService productService,
        ILogger<ProductsController> logger)
        {
            _productService = productService;
            _logger = logger;
        }

        // -----
        //

        // Read from Route Data
        // ----

        // GET /api/products/details/2
        [HttpGet("details/{id}")]
        public IActionResult GetProductById()
```

```

{
    // Step 1: Read "id" from route values
    var idValue =
HttpContext.Request.RouteValues["id"]?.ToString();

    // Step 2: Validate and parse it into an integer
    if (!int.TryParse(idValue, out int productId))
        return BadRequest("Invalid Product ID format.");

    // Step 3: Use injected service to fetch product
    var product = _productService.GetById(productId);

    // Step 4: Check existence
    if (product == null)
        return NotFound($"Product with ID {productId} not
found.");}

    _logger.LogInformation($"Product {productId} fetched at
{DateTime.UtcNow}");
    return Ok(product);
}

// -----
-
// Read from Query String
// -----
-

// GET /api/products/search?
category=Electronics&minPrice=2000&maxPrice=70000
[HttpGet("search")]
public IActionResult SearchProducts()
{
    // Step 1: Extract query parameters
    string? category = HttpContext.Request.Query["category"];
    decimal.TryParse(HttpContext.Request.Query["minPrice"], out
decimal minPrice);
    decimal.TryParse(HttpContext.Request.Query["maxPrice"], out
decimal maxPrice);
}

```

```

    // Step 2: Call service with filters
    var result = _productService.Search(
        category,
        minPrice > 0 ? minPrice : null,
        maxPrice > 0 ? maxPrice : null
    );

    // Step 3: Log the query and return result
    _logger.LogInformation($"Search executed for Category={category}, MinPrice={minPrice}, MaxPrice={maxPrice}");
    return Ok(result);
}

// -----
-
// Read from Header
// -----
-

// GET /api/products/all
// Header: X-Api-Key: secret123
[HttpGet("all")]
public IActionResult GetAllProducts()
{
    // Step 1: Extract custom header
    var apiKey = HttpContext.Request.Headers["X-Api-Key"].ToString();

    // Step 2: Validate header presence
    if (string.IsNullOrWhiteSpace(apiKey))
        return Unauthorized("Missing API key.");

    // Step 3: Check header value for access control
    if (apiKey != "secret123")
        return Unauthorized("Invalid API key.");

    // Step 4: Get all active products from service
    var products = _productService.GetAll().Where(p =>

```

```

p.IsActive);

        _logger.LogInformation("Active products fetched using API
key authentication.");
        return Ok(products);
    }

    // -----
    //

    // Read from Request Body (Manual JSON Parsing)
    // -----
    //

    // POST /api/products/add
    // Body: {
"name": "Tablet", "price": 25000, "category": "Electronics", "isActive": true, "stock": 15, "rating": 4.6 }
    [HttpPost("add")]
    public async Task<IActionResult> AddProduct()
    {
        // Step 1: Read raw JSON body from request
        string body;
        using (var reader = new
StreamReader(HttpContext.Request.Body))
            body = await reader.ReadToEndAsync();

        // Step 2: Validate empty request
        if (string.IsNullOrWhiteSpace(body))
            return BadRequest("Empty request body.");

        // Step 3: Try to deserialize JSON into Product object
        Product? product;
        try
        {
            product = JsonSerializer.Deserialize<Product>(body, new
JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true // Allows
flexible casing

```

```

        });

    }

    catch
    {
        return BadRequest("Invalid JSON format.");
    }

    // Step 4: Manual validation
    if (product == null ||
string.IsNullOrWhiteSpace(product.Name) || product.Price <= 0)
        return BadRequest("Invalid product data. Name and Price
are required.");

    // Step 5: Add to in-memory collection using service
    _productService.Add(product);

    _logger.LogInformation($"New product '{product.Name}' added
successfully at {DateTime.UtcNow}.");
    return Ok(product);
}

// -----
-
// Mix of Multiple Parameters (Route + Query + Header)
// -----
-

// PUT /api/products/discount/2?discountPercent=10
// Header: X-Api-Key: secret123
[HttpPut("discount/{id}")]
public IActionResult ApplyDiscount()
{
    // Step 1: Read from Route
    var routeId =
HttpContext.Request.RouteValues["id"]?.ToString();
    if (!int.TryParse(routeId, out int productId))
        return BadRequest("Invalid product ID.");

    // Step 2: Read from Query String

```

```

        var discountQuery =
HttpContext.Request.Query["discountPercent"].ToString();
        if (!decimal.TryParse(discountQuery, out decimal
discountPercent) || discountPercent <= 0)
            return BadRequest("Invalid discount percent.");

        // Step 3: Read from Header (simulate authentication)
        var apiKey = HttpContext.Request.Headers["X-Api-
Key"].ToString();
        if (string.IsNullOrWhiteSpace(apiKey))
            return Unauthorized("Missing API key.");
        if (apiKey != "secret123")
            return Unauthorized("Invalid API key.");

        // Step 4: Manually fetch service (for demo purposes)
        var productService =
HttpContext.RequestServices.GetService<IProductService>();
        if (productService == null)
            return StatusCode(500, "Product service unavailable.");

        // Step 5: Apply discount logic
        bool updated = productService.UpdatePrice(productId,
discountPercent);
        if (!updated)
            return NotFound($"No product found with ID
{productId}.");

        _logger.LogInformation($"Discount of {discountPercent}%
applied to Product ID {productId} at {DateTime.UtcNow}.");
        return Ok($"Product #{productId} updated successfully with
{discountPercent}% discount.");
    }
}
}

```

Drawbacks or Challenges without Model Binding

While the above example works, it exposes the following drawbacks or challenges of doing everything manually:

- Excessive Duplicate Code: Every action method contains repetitive code for reading route values, query strings, headers, and request bodies. For instance, `HttpContext.Request.RouteValues`, `HttpContext.Request.Query`, and `HttpContext.Request.Body` appears across multiple methods.
- Manual Type Conversion and Parsing Errors: Since all incoming data is read as strings, we must manually convert them into appropriate .NET types (`int.TryParse`, `decimal.TryParse`, etc.). Each conversion introduces potential for runtime errors and edge cases (e.g., invalid numbers or missing query parameters)
- Lack of Centralized Validation: For every request, the developer must manually check for nulls, empty values, invalid formats, and business constraints using multiple if statements.
- Error Handling is Tedious and Inconsistent: In the manual approach, every possible failure (bad route value, missing header, malformed JSON, invalid discount, etc.) must be caught and handled individually with explicit `BadRequest()`, `Unauthorized()`, or `NotFound()` responses. This creates inconsistencies; one endpoint may return a plain string message, while another may return a differently formatted JSON error.
- No Automatic Validation or 400 Response: Without Model Binding and `[ApiController]`, the framework doesn't automatically validate incoming models or return standardized 400 responses for invalid inputs.

Using Model Binding & `[ApiController]` in ASP.NET Core Web API

In ASP.NET Core Web API, Model Binding automatically maps incoming HTTP request data (from route, query string, headers, form, or body) to method parameters and model objects. When combined with the `[ApiController]` attribute, the framework:

- Automatically Validates Incoming Models and populates `ModelState`.
- Returns 400 Bad Request responses automatically if validation fails.
- Provides Cleaner and More Maintainable controller code.

- Reduces duplicate code by eliminating repetitive parsing, type conversion, and validation code.

Updated Product Entity – With Data Annotations

Please modify the Product Entity as follows.

```
using System.ComponentModel.DataAnnotations;
namespace ModelBindingDemo.Models
{
    public class Product
    {
        public int Id { get; set; }

        [Required(ErrorMessage = "Product name is required.")]
        [StringLength(100, MinimumLength = 3, ErrorMessage = "Product
name must be between 3 and 100 characters.")]
        public string Name { get; set; } = null!;

        [Required(ErrorMessage = "Price is required.")]
        [Range(1, 100000, ErrorMessage = "Price must be between 1 and
100000.")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Category is required.")]
        [StringLength(50, ErrorMessage = "Category name cannot exceed 50
characters.")]
        public string Category { get; set; } = null!;

        public bool IsActive { get; set; } = true;

        [Range(0, 10000, ErrorMessage = "Stock must be between 0 and
10000.")]
        public int Stock { get; set; }

        [Range(0.0, 5.0, ErrorMessage = "Rating must be between 0.0 and
5.0.")]
        public double Rating { get; set; }
    }
}
```

```
        public DateTime CreatedDate { get; set; } = DateTime.UtcNow;
    }
}
```

ProductsController — Using Model Binding & [ApiController]

Please modify the ProductsController as follows.

```
using Microsoft.AspNetCore.Mvc;
using ModelBindingDemo.Models;
using ModelBindingDemo.Services;

namespace ModelBindingDemo.Controllers
{
    // Enables automatic model binding, validation, and 400 error
    responses
    [ApiController]
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase
    {
        private readonly ILogger<ProductsController> _logger;
        private readonly IProductService _productService;

        public ProductsController(IProductService productService,
        ILogger<ProductsController> logger)
        {
            _productService = productService;
            _logger = logger;
        }

        // -----
        //

        // Route Data Example
        // -----
        //

        // GET /api/products/details/2
        [HttpGet("details/{id}")]
        public IActionResult GetProductById([FromRoute] int id)
```

```

{
    // Model Binding automatically maps {id} → int id
    var product = _productService.GetById(id);

    if (product == null)
        return NotFound($"Product with ID {id} not found.");

    _logger.LogInformation($"Product {id} fetched successfully
at {DateTime.UtcNow}.");
    return Ok(product);
}

// -----
-
// Query String Example
// -----
-
// GET /api/products/search?
category=Electronics&minPrice=2000&maxPrice=70000
[HttpGet("search")]
public IActionResult SearchProducts(
    [FromQuery] string? category,
    [FromQuery] decimal? minPrice,
    [FromQuery] decimal? maxPrice)
{
    // Model Binding automatically maps query string parameters
    var result = _productService.Search(category, minPrice,
maxPrice);

    _logger.LogInformation($"Search executed for Category=
{category}, MinPrice={minPrice}, MaxPrice={maxPrice}.");
    return Ok(result);
}

// -----
-
// Header Example
// -----

```

```

-
    // GET /api/products/all
    // Header: X-Api-Key: secret123
    [HttpGet("all")]
    public IActionResult GetAllProducts([FromHeader(Name = "X-Api-
Key")] string apiKey)
    {
        // [FromHeader] automatically binds header values
        if (string.IsNullOrWhiteSpace(apiKey))
            return Unauthorized("Missing API key.");

        if (apiKey != "secret123")
            return Unauthorized("Invalid API key.");

        var products = _productService.GetAll().Where(p =>
p.IsActive);

        _logger.LogInformation("Active products fetched successfully
using API key authentication.");
        return Ok(products);
    }

    // -----
-
    // Request Body Example (with Data Annotations)
    // -----
-
    // POST /api/products/add
    // Body: {
"name": "Tablet", "price": 25000, "category": "Electronics", "isActive": true, "stock": 15, "rating": 4.6 }
    [HttpPost("add")]
    public IActionResult AddProduct([FromBody] Product product)
    {
        // No need to check ModelState.IsValid - [ApiController]
does it automatically
        // If validation fails, ASP.NET Core returns 400 Bad Request
with detailed errors

```

```

        _productService.Add(product);

        _logger.LogInformation($"New product '{product.Name}' added
successfully at {DateTime.UtcNow}.");

        // Returns 201 Created with location header
        return CreatedAtAction(nameof(GetProductById), new { id =
product.Id }, product);
    }

    // -----
    //

    // Mix of Multiple Parameters (Route + Query + Header + Service)
    // -----
    //

    // PUT /api/products/discount/2?discountPercent=10
    // Header: X-Api-Key: secret123
    [HttpPut("discount/{id}")]
    public IActionResult ApplyDiscount(
        [FromRoute] int id,
        [FromQuery] decimal discountPercent,
        [FromHeader(Name = "X-Api-Key")] string apiKey,
        [FromServices] IProductService productService)
    {
        if (string.IsNullOrWhiteSpace(apiKey))
            return Unauthorized("Missing API key.");

        if (apiKey != "secret123")
            return Unauthorized("Invalid API key.");

        if (discountPercent <= 0)
            return BadRequest("Discount percent must be greater than
zero.");
    }

    // Using service from [FromServices] instead of controller
    // field
    bool updated = productService.UpdatePrice(id,

```

```

discountPercent);

    if (!updated)
        return NotFound($"No product found with ID {id}.");

        _logger.LogInformation($"Discount of {discountPercent}%
applied to Product ID {id} at {DateTime.UtcNow}.");
        return Ok($"Product #{id} updated successfully with
{discountPercent}% discount.");
    }
}

```

Advantages of using Model Binding in ASP.NET Core Web API

- Clean, Minimal Controller Code: No more `HttpContext.Request.RouteValues`, `Request.Query`, or `StreamReader`. Model Binding automatically matches data from the request to action parameters.
- Automatic Type Conversion: String values from the request are automatically converted to their proper types (int, decimal, bool, etc.), no need for `TryParse`
- Built-In Validation: With `[ApiController]`, ASP.NET Core automatically:
 - Checks `ModelState` after binding.
 - Returns 400 Bad Request with detailed validation errors if the model is invalid.
 - You can add `[Required]`, `[Range]`, or `[StringLength]` attributes on your model for even stronger validation.
- Consistent Error Responses: Validation errors are automatically returned in a standardized JSON structure using the `ProblemDetails` format, making the API responses consistent and client-friendly.
- Declarative Data Source Mapping: Attributes like `[FromRoute]`, `[FromQuery]`, `[FromHeader]`, and `[FromBody]` clearly indicate where each value is expected to come from, improving readability and maintainability.

Entity Framework Core

Entity Framework Core (EF Core) is Microsoft's **Modern, Lightweight, and Cross-Platform Object-Relational Mapper (ORM)** for .NET applications.

When combined with **ASP.NET Core Web API** and **SQL Server**, EF Core provides a clean, scalable, and testable data access layer, essential for building real-time production APIs.

Why Use EF Core in ASP.NET Core Web API?

- **Productivity Boost:** Automatically maps our C# classes (entities) to database tables. It eliminates repetitive SQL queries for common operations like insert, update, or delete.
- **Cross-Platform & Lightweight:** Runs on Windows, Linux, and macOS. Supports multiple databases like SQL Server, PostgreSQL, MySQL, SQLite, and more.
- **Code First Development:** Define database structure directly from our C# model classes. EF Core generates and maintains the database schema using **migrations**.
- **LINQ Support:** We can query our data using C# LINQ expressions instead of raw SQL, safer and more maintainable.
- **Change Tracking & Transactions:** EF Core automatically tracks entity states (Added, Modified, Deleted) and also handles database transactions efficiently.
- **Unit testing friendly:** Replace the real provider with in-memory or SQLite for tests; mock repositories/services around DbContext.
- **Performance:** Modern change tracker, compiled models, AsNoTracking, batch ops, and provider-specific optimizations.

Which EF Core Packages to Install?

- `Microsoft.EntityFrameworkCore` → The core package that provides EF Core functionalities such as LINQ, Change Tracking, and Model Configuration.
- `Microsoft.EntityFrameworkCore.SqlServer` → The Database Provider for SQL Server, adds SQL Server-specific features and optimizations.

- Microsoft.EntityFrameworkCore.Tools → Provides PowerShell commands for database Migrations, Scaffolding, and Database Updates.

Commonly Used EF Core Database Providers

The following are some of the commonly used database providers in Entity Framework Core. Each of the following packages must be installed separately, depending on your database system.

- Microsoft.EntityFrameworkCore.SqlServer → for SQL Server
- Pomelo.EntityFrameworkCore.MySql → for MySQL
- Npgsql.EntityFrameworkCore.PostgreSQL → for PostgreSQL
- Microsoft.EntityFrameworkCore.SQLite → for SQLite
- Oracle.EntityFrameworkCore → for Oracle

So, the Database Provider Package is the essential layer that gives EF Core its ability to communicate with any relational database. Without it, EF Core cannot function because it wouldn't know:

- What SQL syntax to use,
- How to translate LINQ,
- How to handle schema creation or migrations.

What is a Database Connection String in Entity Framework Core?

In Entity Framework Core (EF Core), a **Database Connection String** is a **Configuration String** that contains all the details required for your application to establish a connection with a database server. Think of it as the **address, credentials**, and instructions EF Core needs to locate and talk to your database.

When your DbContext runs (for example, while fetching data or saving changes), EF Core uses this connection string to:

- Know **Which Server** to connect to (e.g., SQL Server on your local machine or cloud),

- Know **Which Database** to use (e.g., ProductDB),
- Know How to **authenticate** (Windows Authentication or SQL Authentication),
- And manage other connection-level settings (like pooling, timeouts, etc.).

Without it, EF Core doesn't know where to go. It's a small string but carries powerful information, for example: "Server=.; Database=ProductDB; Trusted_Connection=True; MultipleActiveResultSets=True;"

Let's break it down briefly:

1. Server=.; → The dot (.) means the local SQL Server instance.
2. Database=ProductDB; → Name of the database to connect to.
3. Trusted_Connection=True; → Use Windows Authentication instead of username/password.
4. MultipleActiveResultSets=True; → Allows multiple queries to run on one connection (used by EF Core internally).

So, in EF Core, when we call something like `options.UseSqlServer(connectionString)` inside your Program.cs, this connection string is what EF Core uses to communicate with SQL Server.

Storing Database Connection String in appsettings.json File:

Let us proceed and store the connection string inside `appsettings.json` file. The `appsettings.json` file in ASP.NET Core acts like a **Central Configuration File** for our application. It's similar to `web.config` in older .NET Framework apps, but it's **JSON-based**, easier to manage, and supports hierarchical configurations.

Now, within the `appsettings.json` file, we need to add a section for the configuration string. So, please modify the `appsettings.json` file as follows:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

```

    }
},
"AllowedHosts": "*",
"ConnectionStrings": {
    "DefaultConnection": "Server=LAPTOP-
6P5NK25R\\SQLSERVER2022DEV;Database=ProductDB;Trusted_Connection=True;Tr
ustServerCertificate=True;"
}
}

```

Understanding the Hierarchy

- “**ConnectionStrings**” → A logical grouping of all connection strings (you can have multiple).
- “**DefaultConnection**” → The name/key of this connection string (you can name it whatever you like).
- The actual string value → “ **Server=LAPTOP-6P5NK25R\SQLSERVER2022DEV;**
Database=ProductDB; Trusted_Connection=True; TrustServerCertificate=True;“.

How to Configure Database Connection String and DbContext?

Once the connection string is defined, we must tell EF Core how to use it. This is done by registering the DbContext and configuring it to use SQL Server inside the Program class. Let’s understand this step in depth.

In the modern ASP.NET Core Minimal Hosting Model (from .NET 6 onwards), everything is configured inside Program.cs. We no longer have Startup.cs, so the setup happens here. So, please modify the Program class as follows:

```

using Microsoft.EntityFrameworkCore;
using ProductManagementAPI.Data;

namespace ProductManagementAPI
{
    public class Program
    {
        public static void Main(string[] args)

```

```
{  
    var builder = WebApplication.CreateBuilder(args);  
  
    // Add services to the container.  
    builder.Services.AddControllers()  
        .AddJsonOptions(options =>  
    {  
        // This will use the property names as defined in the C#  
model  
        options.JsonSerializerOptions.PropertyNamingPolicy =  
null;  
    });  
  
    builder.Services.AddEndpointsApiExplorer();  
    builder.Services.AddSwaggerGen();  
  
    builder.Services.AddDbContext<AppDbContext>(options =>  
  
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultC  
onnection")));  
  
    var app = builder.Build();  
  
    // Configure the HTTP request pipeline.  
    if (app.Environment.IsDevelopment())  
    {  
        app.UseSwagger();  
        app.UseSwaggerUI();  
    }  
  
    app.UseHttpsRedirection();  
  
    app.UseAuthorization();  
  
    app.MapControllers();  
  
    app.Run();  
}
```

}
 }

[Recap Workshop] Relationships

Entity Framework Core (EF Core) relationships allow us to model real-world associations between business entities. In an E-Commerce application, every entity interacts with others, customers place orders, orders contain products, products belong to categories, and so on. At the end of this session, you will understand the following pointers:

Why Do We Need Relationships in Entity Framework Core?

In a relational database, data is spread across multiple tables, and those tables must be able to “talk” to each other. Relationships ensure this happens correctly and efficiently. Every business domain has entities that are logically connected to one another. For example, in an Ecommerce application:

- A Customer **places** Orders.
- Each Order **contains multiple** Products.
- Each Product **belongs to a specific** Category.

These are not random objects. They are connected through **Relationships** that define how one entity interacts with another.

What Are the Different Types of Relationships in EF Core? EF Core supports three core types of relationships between entities, reflecting real-world relational database design:

- **One-to-One (1:1)** Relationship
- **One-to-Many (1:M)** Relationship
- **Many-to-Many (M:M)** Relationship

E-Commerce Order Management API

The E-Commerce Web API, built with ASP.NET Core and Entity Framework Core, is a scalable, real-world online shopping backend system. It models how customers browse, order, and purchase products using a relational SQL Server database, while maintaining

clean separation of concerns through **entities**, **relationships**, and **data-access abstractions**.

This project demonstrates how EF Core manages **different types of entity relationships (1:1, 1:M, M:M)** using the **EF Core Code-First Approach**. It focuses on designing a domain-driven model that captures all the logical connections found in a real-life commerce environment.

Key Entities and Their Roles

1. **Customer** → The main user of the system. Each customer can have one Profile, multiple Addresses, and multiple Orders.
2. **Profile** → Holds personal information like gender and birth date. It has a strict 1:1 relationship with its customer.
3. **Address** → Stores delivery and billing locations. A Customer can own many addresses (1:M).
4. **Category** → Organizes products into logical groups such as “Electronics” or “Clothing.”
5. **Product** → Represents sellable items, each belonging to exactly one Category but linked to many Orders via OrderItems.
6. **Order** → Captures a customer’s purchase record and links to OrderItems, Addresses, and OrderStatus.
7. **OrderItem** → Acts as a junction table connecting Orders and Products, supporting a Many-to-Many relationship and storing transactional details like quantity and price.
8. **OrderStatus** → A lookup entity seeded from an enum, ensuring standardized status values like Pending, Confirmed, Shipped, Delivered, and Cancelled.
9. **BaseAuditableEntity** → A reusable base class that provides auditing and soft-delete columns (CreatedAt, UpdatedAt, CreatedBy, IsActive), ensuring consistency across all tables.

One-to-Many (1:M) Relationship

A One-to-Many (1:M) relationship means that a **single record in the Principal Entity** can be linked to **multiple records in the Dependent Entity**. However, each record in the

dependent entity is always related to only **one principal entity**. Such relationships are typically defined using:

1. A **Collection Navigation Property** (`ICollection<T>`) on the **principal side**, and
2. A **Foreign Key Property** on the dependent side with a **Reference Navigation Property**.

Example 1: Customer ↔ Address

- A **Customer** can have **multiple Addresses**, for example, Home, Office, Billing, or Shipping.
- Each **Address** belongs to **exactly one Customer**.

Here:

Customer → Principal Entity Address → Dependent Entity (It contains a foreign key pointing to the Customer table)

Example 2: Category ↔ Product

A single **Category** (e.g., “Electronics”) can include many **Products** (e.g., Laptop, Camera, TV). Each **Product** is linked to only **one Category**.

Here:

- **Category → Principal Entity**
- **Product → Dependent Entity** (It contains a foreign key pointing to the Category table)

Example 3: Customer ↔ Order

- A single **Customer** can place **multiple Orders** over time.
- Each **Order** belongs to **exactly one Customer**.

Here:

- **Customer → Principal Entity**

- Order → Dependent Entity (It contains a foreign key pointing to the Customer table)

⚠ In Simple Words, a One-to-Many relationship means One parent → Many children. It models real-world scenarios like:

- **⚠** One Customer has many Addresses and Orders.
- **⚠** One Category containing many Products.

⚠ EF Core automatically builds and manages these links using **Foreign Keys** and **Navigation Properties**, keeping both our database and our object model perfectly aligned.

Many-to-Many (M:M) Relationship

A Many-to-Many (M:M) relationship means that multiple records in one entity can be linked to multiple records in another entity, and vice versa. In simple terms, each record in Entity A can be associated with several records in Entity B, and each record in Entity B can also be associated with several records in Entity A.

Because relational databases don't support direct many-to-many relationships between two tables, Entity Framework Core uses a junction (bridge or joining) table, a third table that contains foreign keys from both entities. This bridge table connects the two entities and can also store additional information about their association.

Example: Order ↔ Product (via OrderItem)

In an E-Commerce system:

- A single Order can contain multiple Products.
- A single Product can appear in multiple Orders.

This is a perfect real-world case of a **Many-to-Many relationship**.

How EF Core Represents It:

To represent this relationship, we introduce a separate entity, **OrderItem**, which serves as a **linking table** between Orders and Products. The **OrderItem** entity includes:

1. OrderId → Foreign Key referencing Orders
2. ProductId → Foreign Key referencing Products

⚠ A Many-to-Many relationship connects two entities through a third entity that acts as a bridge. In our e-commerce system:

- The **Order** represents the customer's purchase.
- The **Product** represents items for sale.
- **OrderItem** connects them, showing which products belong to which orders, along with their quantities and prices.

Create a New Project using Visual Studio

Name your project → ECommerceApp

Install Required EF Core Packages

Open the Package Manager Console in Visual Studio: Tools → NuGet Package Manager → Package Manager Console. Then run the following commands:

- Install-Package Microsoft.EntityFrameworkCore → Core EF ORM features.
- Install-Package Microsoft.EntityFrameworkCore.SqlServer → SQL Server provider.
- Install-Package Microsoft.EntityFrameworkCore.Tools → CLI commands like Add-Migration and Update-Database.
- Install-Package Swashbuckle.AspNetCore → Swagger Docs
- Install-Package Microsoft.AspNetCore.OpenApi → Swagger Schema

Create the Folder Structure

Inside your project, create the following folders to organize your files:

- **Models** → Contains all entity classes that represent the database tables and define the relationships between them.
- **Data** → Holds the AppDbContext class and any data-related configurations like seeding or migrations.
- **Enums** → Stores enumerations such as OrderStatusEnum, used for predefined constant values throughout the application.
- **DTOs** → Contains Data Transfer Objects for handling structured request and response models between API endpoints and clients.

Create Order Status Enum

Create a class file named **OrderStatusEnum.cs** within the Enums folder, then copy and paste the following code. The Enum is used in code to express allowed order states in a type-safe way and to seed the OrderStatus table, keeping domain logic readable while enforcing valid values at the database level.

```
namespace ECommerceOrderManagementAPI.Enums
{
    // Enum used in application logic to represent possible order
    statuses.

    public enum OrderStatusEnum
    {
        Pending = 1,
        Confirmed = 2,
        Shipped = 3,
        Delivered = 4,
        Cancelled = 5
    }
}
```

Create Entity Classes

We will create 9 Entities in the models folder: `BaseAuditableEntity`, `Customer`, `Profile`, `Address`, `Category`, `Product`, `OrderStatus`, `Order`, and `OrderItem`.

We will use Default Conventions only (no Data Annotations, no Fluent API). Relationships are automatically inferred from navigation properties.

`BaseAuditableEntity.cs`

```
namespace ECommerceOrderManagementAPI.Models
{
    // Base class that provides soft-delete and audit tracking.
    // Every domain entity inherits this for consistency.
    public abstract class BaseAuditableEntity
    {
        // Indicates whether the record is active (used for soft delete)
        public bool IsActive { get; set; } = true;
        // Audit Columns
        public DateTime CreatedAt { get; set; } = DateTime.UtcNow;    //
        Record creation timestamp
        public DateTime? UpdatedAt { get; set; }                         //
        Last update timestamp
        public string? CreatedBy { get; set; }                           //
        Optional: can be set by middleware later
        public string? UpdatedBy { get; set; }                           //
        Optional: for tracking modifications
    }
}
```

`Customer.cs` It serves as a Principal Entity for related data such as `Profiles`, `Addresses`, and `Orders`. It holds basic customer details such as `name`, `email`, and `phone number`, and maintains **1:1, 1:M, and M:M relationships** through navigation properties.

```
using System.Net;
using static System.Runtime.InteropServices.JavaScript.JSType;

namespace ECommerceOrderManagementAPI.Models
{
    // Principal for Profile, Addresses, and Orders
    public class Customer : BaseAuditableEntity
```

```

{
    // Primary Key (use explicit name to be clear)
    public int CustomerId { get; set; }
    // Basic Customer Info (Required by default through non-nullable
    refs)
    public string Name { get; set; } = null!;
    public string Email { get; set; } = null!;
    public string Phone { get; set; } = null!;
    // 1:1 → A customer may have one Profile
    public virtual Profile? Profile { get; set; }
    // 1:M → A customer can have many addresses and orders
    public virtual ICollection<Address>? Addresses { get; set; }
    public virtual ICollection<Order>? Orders { get; set; }
}
}

```

Profile.cs The **Profile** class stores extended customer personal details, such as gender, display name, and date of birth. It has a **one-to-one (1:1) relationship with the Customer**, meaning each customer has exactly one profile, and its primary key is also the foreign key referencing the customer.

```

using System.ComponentModel.DataAnnotations;

namespace ECommerceOrderManagementAPI.Models
{
    // Dependent in the 1:1 relationship with Customer
    // PK = FK (same property) is the canonical required 1:1 pattern.
    public class Profile : BaseAuditableEntity
    {
        // Both Primary Key and Foreign Key to Customer
        [Key] // Keep this to make the PK explicit for readers
        public int CustomerId { get; set; }
        // Required 1:1 nav back to principal Customer
        public virtual Customer Customer { get; set; } = null!;
        // Extra profile info
        public string DisplayName { get; set; } = null!;
        public string Gender { get; set; } = null!;
    }
}

```

```
        public DateTime DateOfBirth { get; set; }  
    }  
}
```

Address.cs The **Address** class stores physical addresses used for billing or shipping. It has a **one-to-many (1:M) relationship with the Customer**; one customer can have multiple addresses, but each address belongs to a single customer. It includes address-related fields like street, city, postal code, and country.

```
namespace ECommerceOrderManagementAPI.Models  
{  
    // Dependent in 1:M with Customer  
    public class Address : BaseAuditableEntity  
    {  
        public int AddressId { get; set; }  
        public string Line1 { get; set; } = null!;  
        public string? Line2 { get; set; }  
        public string Street { get; set; } = null!;  
        public string City { get; set; } = null!;  
        public string PostalCode { get; set; } = null!;  
        public string Country { get; set; } = null!;  
        // REQUIRED Relationship: Every Address must belong to a  
        Customer  
        public int CustomerId { get; set; }  
        public virtual Customer Customer { get; set; } = null!;  
    }  
}
```

Category.cs

The **Category** class represents a logical grouping of products, such as “Electronics” or “Books.” It is a **Principal Entity** in a one-to-many relationship with **Product**. Each category can have multiple products, enabling easier organization, filtering, and reporting of items by category.

```
namespace ECommerceOrderManagementAPI.Models  
{  
    // Principal in 1:M with Product
```

```

public class Category : BaseAuditableEntity
{
    public int CategoryId { get; set; }
    public string Name { get; set; } = null!;
    public string? Description { get; set; }
    // One Category → Many Products
    public virtual ICollection<Product>? Products { get; set; }
}

```

Product.cs The Product class represents individual items available for purchase. It belongs to one category and can appear in multiple orders through the **OrderItem joining entity**. It contains properties like name, price, stock, SKU, and description, making it central to the product catalog.

```

using System.ComponentModel.DataAnnotations.Schema;

namespace ECommerceOrderManagementAPI.Models
{
    // Dependent in 1:M with Category, and principal for OrderItems
    public class Product : BaseAuditableEntity
    {
        public int ProductId { get; set; }
        public string Name { get; set; } = null!;
        [Column(TypeName = "decimal(18,2)")]
        public decimal Price { get; set; }
        public int Stock { get; set; }
        public string SKU { get; set; } = null!;
        public string? Description { get; set; }
        // Required FK to Category (1 Product belongs to 1 Category)
        public int CategoryId { get; set; }
        public virtual Category Category { get; set; } = null!;
        // Many-to-Many → OrderItem acts as the bridge entity
        public virtual ICollection<OrderItem>? OrderItems { get; set; }
    }
}

```

OrderStatus.cs

The OrderStatus class serves as a Master Lookup Table for storing predefined order states such as Pending, Confirmed, Shipped, Delivered, and Cancelled. It maintains a one-to-many relationship with the Order entity, ensuring consistent tracking of an order's progress throughout its lifecycle.

```
namespace ECommerceOrderManagementAPI.Models
{
    // Represents a master table for all possible order statuses.
    // The enum values are seeded here for database persistence.
    public class OrderStatus : BaseAuditableEntity
    {
        public int OrderStatusId { get; set; }      // Primary Key
        public string Name { get; set; } = null!;    // e.g., "Pending",
        "Shipped", etc.
        public string? Description { get; set; }
        // Navigation Property (1:M) → One status can be assigned to
        many orders
        public virtual ICollection<Order>? Orders { get; set; }
    }
}
```

Order.cs

The Order class represents a single placed order. It connects customers, addresses, and products through various relationships: **1:M with OrderItems**, **1:M with Customer**, and **M:M via OrderItem with Product**. It tracks order details, including the total amount, order date, and order status.

```
using System.ComponentModel.DataAnnotations.Schema;

namespace ECommerceOrderManagementAPI.Models
{
    // Dependent entity in 1:M with Customer
    // Principal in 1:M with OrderItems
    public class Order : BaseAuditableEntity
    {
```

```

        public int OrderId { get; set; }
        public DateTime OrderDate { get; set; }
        // REQUIRED: Each Order should have a valid Status
        public int OrderStatusId { get; set; }
        public virtual OrderStatus OrderStatus { get; set; } = null!;
        [Column(TypeName = "decimal(18,2)")]
        public decimal TotalAmount { get; set; }
        // Optional: Billing & Shipping addresses are Optional
        public int? ShippingAddressId { get; set; }
        public virtual Address? ShippingAddress { get; set; }
        public int? BillingAddressId { get; set; }
        public virtual Address? BillingAddress { get; set; }
        // REQUIRED: Who placed it
        public int CustomerId { get; set; }
        public virtual Customer Customer { get; set; } = null!;
        // 1:M → Each Order can have multiple items
        public virtual ICollection<OrderItem> OrderItems { get; set; } =
    null!;
    }
}

```

OrderItem.cs

The OrderItem class is a **Junction Entity** between Orders and Products, enabling a many-to-many relationship. It stores item-level details, such as product ID, quantity, and unit price, for each product in an order.

```

using System.ComponentModel.DataAnnotations.Schema;

namespace ECommerceOrderManagementAPI.Models
{
    // Dependent in 1:M with Order
    // Dependent in 1:M with Product
    // This is the Joining Entity enabling a Many-to-Many between Orders
    and Products.

    public class OrderItem : BaseAuditableEntity
    {
        public int OrderItemId { get; set; }

```

```

    public int Quantity { get; set; }
    [Column(TypeName = "decimal(18,2)")]
    public decimal UnitPrice { get; set; } // snapshot of price at
purchase time
    // Foreign Keys
    // REQUIRED: each OrderItem belongs to an Order
    public int OrderId { get; set; }
    public virtual Order Order { get; set; } = null!;
    // REQUIRED: each OrderItem refers to a Product
    public int ProductId { get; set; }
    public virtual Product Product { get; set; } = null!;
}
}

```

⚠ Note: All navigations are **virtual**, so you can later enable lazy loading if you wish; for now, we will use eager loading in controller reads.

Create the DbContext

Create a class file named **AppDbContext.cs** within the **Data folder**, then copy and paste the following code. The AppDbContext class is the **EF Core bridge to the SQL Server database**. It defines all DbSet properties for entities, manages relationships automatically through conventions, and seeds initial data for Customers, Products, Categories, Orders, and Order Statuses using the HasData() method.

```

using ECommerceOrderManagementAPI.Models;
using Microsoft.EntityFrameworkCore;

namespace ECommerceOrderManagementAPI.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options) { }

        public DbSet<Customer> Customers { get; set; }
        public DbSet<Profile> Profiles { get; set; }
        public DbSet<Address> Addresses { get; set; }
    }
}

```

```

        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
        public DbSet<OrderStatus> OrderStatuses { get; set; }
        protected override void OnModelCreating(ModelBuilder
modelBuilder)
{
    DateTime seedDate = new(2025, 01, 01, 10, 00, 00);
    // Order Status
    modelBuilder.Entity<OrderStatus>().HasData(
        new OrderStatus { OrderStatusId = 1, Name = "Pending",
Description = "Awaiting confirmation", CreatedAt = seedDate },
        new OrderStatus { OrderStatusId = 2, Name = "Confirmed",
Description = "Confirmed by admin", CreatedAt = seedDate },
        new OrderStatus { OrderStatusId = 3, Name = "Shipped",
Description = "Dispatched to courier", CreatedAt = seedDate },
        new OrderStatus { OrderStatusId = 4, Name = "Delivered",
Description = "Delivered successfully", CreatedAt = seedDate },
        new OrderStatus { OrderStatusId = 5, Name = "Cancelled",
Description = "Cancelled by user/system", CreatedAt = seedDate }
    );
    // Customers
    modelBuilder.Entity<Customer>().HasData(
        new Customer { CustomerId = 1, Name = "Pranaya Rout",
Email = "pranaya@example.com", Phone = "9876543210", CreatedAt =
seedDate },
        new Customer { CustomerId = 2, Name = "Rakesh Kumar",
Email = "rakesh@example.com", Phone = "9876543211", CreatedAt = seedDate
    }
);
    // Profiles
    modelBuilder.Entity<Profile>().HasData(
        new Profile { CustomerId = 1, DisplayName = "Pranaya",
Gender = "Male", DateOfBirth = new(1990, 05, 10), CreatedAt = seedDate },
        new Profile { CustomerId = 2, DisplayName = "Rakesh",
Gender = "Female", DateOfBirth = new(1992, 08, 22), CreatedAt = seedDate
    )
}

```

```

    );
    // Addresses
    modelBuilder.Entity<Address>().HasData(
        new Address { AddressId = 1, Line1 = "123 Market
Street", Street = "Main Rd", City = "Bhubaneswar", PostalCode =
"751001", Country = "India", CustomerId = 1, CreatedAt = seedDate },
        new Address { AddressId = 2, Line1 = "Tech Park Avenue",
Street = "Silicon Street", City = "Cuttack", PostalCode = "753001",
Country = "India", CustomerId = 1, CreatedAt = seedDate },
        new Address { AddressId = 3, Line1 = "45 Lake View",
Street = "West Road", City = "Bhubaneswar", PostalCode = "751010",
Country = "India", CustomerId = 2, CreatedAt = seedDate }
    );
    // Categories
    modelBuilder.Entity<Category>().HasData(
        new Category { CategoryId = 1, Name = "Electronics",
Description = "Electronic Devices", CreatedAt = seedDate },
        new Category { CategoryId = 2, Name = "Books",
Description = "Educational and Fiction", CreatedAt = seedDate }
    );
    // Products
    modelBuilder.Entity<Product>().HasData(
        new Product { ProductId = 1, Name = "Wireless Mouse",
Price = 1200, Stock = 50, SKU = "ELEC-MOUSE-001", CategoryId = 1,
CreatedAt = seedDate },
        new Product { ProductId = 2, Name = "Bluetooth
Headphones", Price = 2500, Stock = 40, SKU = "ELEC-HEAD-002", CategoryId =
1, CreatedAt = seedDate },
        new Product { ProductId = 3, Name = "C# Programming
Book", Price = 899, Stock = 100, SKU = "BOOK-CSPROG-003", CategoryId =
2, CreatedAt = seedDate }
    );
    // Orders
    modelBuilder.Entity<Order>().HasData(
        new Order { OrderId = 1, OrderDate = seedDate,
OrderStatusId = 2, TotalAmount = 3700, ShippingAddressId = 1,
BillingAddressId = 2, CustomerId = 1, CreatedAt = seedDate },

```

```

        new Order { OrderId = 2, OrderDate =
seedDate.AddDays(1), OrderStatusId = 3, TotalAmount = 899,
ShippingAddressId = 3, BillingAddressId = 3, CustomerId = 2, CreatedAt =
seedDate }
    );
    // OrderItems
    modelBuilder.Entity<OrderItem>().HasData(
        new OrderItem { OrderItemId = 1, OrderId = 1, ProductId
= 1, Quantity = 1, UnitPrice = 1200, CreatedAt = seedDate },
        new OrderItem { OrderItemId = 2, OrderId = 1, ProductId
= 2, Quantity = 1, UnitPrice = 2500, CreatedAt = seedDate },
        new OrderItem { OrderItemId = 3, OrderId = 2, ProductId
= 3, Quantity = 1, UnitPrice = 899, CreatedAt = seedDate }
    );
}
}

```

Configure Database Connection

Please add the database connection string in the `appsettings.json` file as follows:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=localhost;Initial Catalog=order_management_db;Integrated Security=True;Pooling=False;Encrypt=True;Trust Server Certificate=True"
  }
}
```

Configure DbContext in Program.cs

Please modify the Program class as follows.

```
using ECommerceOrderManagementAPI.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers()
    .AddJsonOptions(options =>
{
    // This will use the property names as defined in the C# model
    //options.JsonSerializerOptions.PropertyNamingPolicy = null;

    // This makes JSON property name matching case-insensitive ie
    "customerId" or "CustomerID" will bind to CustomerId
    options.JsonSerializerOptions.PropertyNameCaseInsensitive =
true;
});

// Register DbContext and Connection String
builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultC
onnection")));
}

builder.Services.AddSwaggerGen(options =>
{
    // Define API metadata (title, version, description, contact) shown
    in Swagger UI
    options.SwaggerDoc("v1", new()
    {
        Title = "E-Commerce Order Management API",
        Version = "v1",
    });
})
```

```

        Description = "API for managing e-commerce orders, products,
customers, and order processing operations.",
        Contact = new()
        {
            Name = "API Support",
            Email = "support@ecommerceordermanagement.com",
            Url = new
Uri("https://www.ecommerceordermanagement.com/support")
        }
    });
});

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    // Enable middleware to serve generated Swagger as JSON endpoint
    app.UseSwagger();
    // Enable middleware to serve Swagger UI (HTML, JS, CSS, etc.)
    // Provides interactive documentation at /swagger
    app.UseSwaggerUI(options =>
    {
        // Configure the JSON endpoint for Swagger UI to consume
        options.SwaggerEndpoint("/swagger/v1/swagger.json", "E-Commerce
Order Management API V1");
    });
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

```

Creating DTOs

Let us create the DTOs for exchanging the Data.

OrderItemRequestDTO.cs

Create a class file named OrderRequestDTO.cs within the DTOs folder, then copy and paste the following code. It is used for placing new orders. It includes customer ID, address IDs, and a list of ordered items

```
using System.ComponentModel.DataAnnotations;

namespace ECommerceOrderManagementAPI.DTOs
{
    // Represents an individual product item in a new order request
    public class OrderItemRequestDTO
    {
        // The product being ordered
        [Required(ErrorMessage = "ProductId is required.")]
        [Range(1, int.MaxValue, ErrorMessage = "ProductId must be a
positive integer.")]
        public int ProductId { get; set; }

        // Quantity of that product
        [Required(ErrorMessage = "Quantity is required.")]
        [Range(1, 1000, ErrorMessage = "Quantity must be between 1 and
1000.")]
        public int Quantity { get; set; }
    }
}
```

OrderRequestDTO.cs

Create a class file named OrderRequestDTO.cs within the DTOs folder, then copy and paste the following code. It is used for placing new orders. It includes customer ID, address IDs, and a list of ordered items.

```
using System.ComponentModel.DataAnnotations;

namespace ECommerceOrderManagementAPI.DTOs
{
```

```

// Represents the full data required to place a new order
public class OrderRequestDTO
{
    // The customer placing the order
    [Required(ErrorMessage = "CustomerId is required.")]
    [Range(1, int.MaxValue, ErrorMessage = "CustomerId must be a
positive integer.")]
    public int CustomerId { get; set; }

    // Chosen shipping address
    [Required(ErrorMessage = "ShippingAddressId is required.")]
    [Range(1, int.MaxValue, ErrorMessage = "ShippingAddressId must
be a positive integer.")]
    public int ShippingAddressId { get; set; }

    // Chosen billing address
    [Required(ErrorMessage = "BillingAddressId is required.")]
    [Range(1, int.MaxValue, ErrorMessage = "BillingAddressId must be
a positive integer.")]
    public int BillingAddressId { get; set; }

    // List of products in the order
    [Required(ErrorMessage = "Order must contain at least one
item.")]
    [MinLength(1, ErrorMessage = "At least one order item must be
provided.")]
    public List<OrderItemRequestDTO> Items { get; set; } = new();
}

```

OrderItemResponseDTO.cs

Create a class file named OrderItemResponseDTO.cs within the DTOs folder, then copy and paste the following code. It represents each ordered product in response payloads with product and category info.

```

namespace ECommerceOrderManagementAPI.DTOs
{
    // Represents an item inside an order when returning data
    public class OrderItemResponseDTO
    {

```

```
        public string ProductName { get; set; } = null!;
        public string Category { get; set; } = null!;
        public int Quantity { get; set; }
        public decimal UnitPrice { get; set; }
    }
}
```

CustomerResponseDTO.cs

Create a class file named CustomerResponseDTO.cs within the DTOs folder, then copy and paste the following code. It returns summarized customer data, including name, email, and addresses.

```
namespace ECommerceOrderManagementAPI.DTOs
{
    // Summarized view of the customer associated with an order
    public class CustomerResponseDTO
    {
        public int CustomerId { get; set; }
        public string Name { get; set; } = null!;
        public string Email { get; set; } = null!;
        public string? Phone { get; set; }
        public string? DisplayName { get; set; }
        public string? Gender { get; set; }
        public string? DateOfBirth { get; set; }
    }
}
```

OrderResponseDTO.cs

Create a class file named OrderResponseDTO.cs within the DTOs folder, then copy and paste the following code. It wraps complete order details (customer info, order items, and addresses) for API responses.

```
// Represents an order returned from API endpoints
namespace ECommerceOrderManagementAPI.DTOs
{
    // Represents an order returned from API endpoints
    public class OrderResponseDTO
```

```

    {
        public int OrderId { get; set; }
        public string OrderDate { get; set; } = null!;
        public string Status { get; set; } = null!;
        public decimal TotalAmount { get; set; }
        public string ShippingAddress { get; set; } = null!;
        public string BillingAddress { get; set; } = null!;
        public CustomerResponseDTO Customer { get; set; } = null!;
        public List<OrderItemResponseDTO> Items { get; set; } = new();
    }
}

```

Create Order API Controller

Please create a new Empty API Controller named **OrderController** within the **Controllers** folder and add the following code to it. The **OrderController** handles all **API Endpoints** related to order management, such as retrieving all orders, fetching by ID, and placing new orders. It demonstrates how EF Core relationships are utilized in queries and DTO mapping for clean data transfer. Now, we have only implemented the Place Order method; later, we will implement the other methods.

```

using ECommerceOrderManagementAPI.Data;
using ECommerceOrderManagementAPI.DTOs;
using ECommerceOrderManagementAPI.Enums;
using ECommerceOrderManagementAPI.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

// For more information on enabling Web API for empty projects, visit
https://go.microsoft.com/fwlink/?LinkID=397860

namespace ECommerceOrderManagementAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class OrderController : ControllerBase
    {
        private readonly AppDbContext _context;

```

```

public OrderController(AppDbContext context)
{
    _context = context;
}

// POST api/<OrderController>
// PURPOSE: Places a new order with validation and business
rules
    // Demonstrates: Validations, relationships, transactions, soft
audit fields
    [HttpPost]
    public async Task<IActionResult> PlaceOrder([FromBody]
OrderRequestDTO request)
    {
        try
        {
            // STEP 1: Validate Customer existence
            var customer = await _context.Customers
                .Include(c => c.Addresses)
                .FirstOrDefaultAsync(c => c.CustomerId ==
request.CustomerId && c.IsActive);

            if (customer == null)
            {
                return BadRequest($"customer {request.CustomerId} not found or inactive");
            }

            // STEP 2: Validate Shipping & Billing Addresses (must
belong to the same customer)
            var shipping = await _context.Addresses
                .FirstOrDefaultAsync(a => a.AddressId ==
request.ShippingAddressId && a.CustomerId == request.CustomerId);
            var billing = await _context.Addresses
                .FirstOrDefaultAsync(a => a.AddressId ==
request.BillingAddressId && a.CustomerId == request.CustomerId);
        }
    }
}

```

```

        if (shipping == null || billing == null)
            return BadRequest("Invalid Shipping or Billing
Address for this customer.");

        // STEP 3: Validate Products & Stock availability
        var productIds = request.Items.Select(i =>
i.ProductId).ToList();
        var products = await _context.Products
            .Where(p => productIds.Contains(p.ProductId) &&
p.IsActive)
            .ToListAsync();

        // Check product existence
        if (products.Count != request.Items.Count)
            return BadRequest("Some products are invalid or
inactive.");
        // Check stock for each product
        foreach (var item in request.Items)
        {
            var product = products.First(p => p.ProductId ==
item.ProductId);
            if (product.Stock < item.Quantity)
                return BadRequest($"Insufficient stock for
'{product.Name}'. Available: {product.Stock}, Requested:
{item.Quantity}");
        }

        // STEP 4: Create new Order object
        var order = new Order
        {
            CustomerId = request.CustomerId,
            ShippingAddressId = request.ShippingAddressId,
            BillingAddressId = request.BillingAddressId,
            OrderStatusId = (int)OrderStatusEnum.Pending,
            OrderDate = DateTime.UtcNow,
            CreatedAt = DateTime.UtcNow,
            CreatedBy = "System",
            IsActive = true

```

```

};

// STEP 5: Map Order Items and Deduct Stock
order.OrderItems = new List<OrderItem>();
foreach (var item in request.Items)
{
    var product = products.First(p => p.ProductId ==
item.ProductId);

        // Reduce stock from product
        product.Stock -= item.Quantity;
        product.UpdatedAt = DateTime.UtcNow;
        product.UpdatedBy = "System";

        // Create order item record
        order.OrderItems.Add(new OrderItem
        {
            ProductId = product.ProductId,
            Quantity = item.Quantity,
            UnitPrice = product.Price // use price from DB,
not client
        });
}

// STEP 6: Compute total amount
order.TotalAmount = order.OrderItems.Sum(i => i.Quantity
* i.UnitPrice);

// STEP 7: Save changes(transactionally)
_context.Orders.Add(order);
await _context.SaveChangesAsync();

// STEP 8: Return success response
return Ok(new
{
    Message = "Order placed successfully.",
    OrderId = order.OrderId
});

```

```
        }
    catch (Exception ex)
    {
        return StatusCode(500, new
        {
            Message = "Unexpected error occurred while placing
the order.",
            ErrorMessage = ex.Message
        });
    }
}
```

Run Migrations

Open the **Package Manager Console** and execute:

```
# Create migration
dotnet ef migrations add AddNewFeature
```

```
# Apply to database
dotnet ef database update
```

EF Core will:

- Create the database ECommerceDB.
- Create all tables.
- Insert the seed data defined in AppDbContext.

Lazy Loading in EF Core

EF Core doesn't enable Lazy Loading automatically; it must be configured. Lazy Loading requires EF Core to generate runtime proxy classes that override navigation properties and issue background SQL queries when those properties are accessed. To implement Lazy Loading using EF Core, we need to follow the steps below:

- Install the Required Proxy Package for Lazy Loading [Install-Package Microsoft.EntityFrameworkCore.Proxies](#)
- Enable Proxies in DbContext or Program.cs
- Mark Navigation Properties as Virtual

Enable Proxies in PDbContext

Once the Proxy Package is installed, enable lazy-loading proxies while configuring the DbContext. Call both methods for proxies that implement both **lazy-loading** and **change-tracking**. For example:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseChangeTrackingProxies();
```

Creating Customer Controller to Demonstrate Lazy Loading

Create an API Empty Controller named CustomerController within the Controllers folder and then copy and paste the following code. The following controller demonstrates Lazy Loading. Please read the inline comments for a better understanding.

```
using ECommerceOrderManagementAPI.Data;
using Microsoft.AspNetCore.Mvc;
```

```

// For more information on enabling Web API for empty projects, visit
https://go.microsoft.com/fwlink/?LinkID=397860

namespace ECommerceOrderManagementAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CustomerController : ControllerBase
    {
        private readonly AppDbContext _context;
        public CustomerController(AppDbContext context)
        {
            _context = context;
        }

        // GET: api/<CustomerController>
        [HttpGet]
        //public IEnumerable<string> Get()
        //{
        //    return new string[] { "value1", "value2" };
        //}

        // GET api/<CustomerController>/5
        [HttpGet("{id:int}")]
        public async Task<IActionResult> GetCustomerById(int id)
        {
            // STEP 1: Fetch only the Customer entity by primary key.
            // Lazy loading ensures that related entities are NOT loaded
            at this point.

            var customer = await _context.Customers.FindAsync(id);
            if (customer == null)
                return NotFound($"Customer with ID {id} not found.");
            //

            =====

            // STEP 2: Accessing navigation properties below triggers
            // automatic SQL queries via EF Core's proxy objects.
            // Each navigation property is loaded only when first
        }
    }
}

```

```

accessed.

// =====
// Load Profile automatically when accessed
// EF triggers a SELECT for Profile table
var profile = customer.Profile;
// Load Addresses automatically when accessed
// EF triggers a SELECT for Addresses table
var addresses = customer.Addresses;
// Load Orders automatically when accessed
// EF triggers a SELECT for Orders table
var orders = customer.Orders;
//
// =====
// STEP 3: Construct an Anonymous Object with all details.
// Related entities are now fully available because EF Core
// has executed necessary SQL queries automatically.
//
// =====
var result = new
{
    customer.CustomerId,
    customer.Name,
    customer.Email,
    customer.Phone,
    Profile = profile == null ? null : new
    {
        profile.DisplayName,
        profile.Gender,
        DateOfBirth = profile.DateOfBirth.ToString("yyyy-MM-
dd")
    },
    Addresses = addresses?.Select(a => new
    {
        a.AddressId,
        a.Line1,
        a.Street,
        a.City,
    })
}

```

```

        a.PostalCode,
        a.Country
    }).ToList(),
    Orders = orders?.Select(o => new
    {
        o.OrderId,
        OrderDate = o.OrderDate.ToString("yyyy-MM-dd
HH:mm:ss"),
        o.TotalAmount,
        Status = o.OrderStatus?.Name, // Lazy load triggers
OrderStatus if accessed
        Items = o.OrderItems?.Select(i => new
        {
            i.ProductId,
            ProductName = i.Product.Name,           // Triggers
lazy load of Product
            Category = i.Product.Category.Name,   // Triggers
lazy load of Category
            i.Quantity,
            i.UnitPrice
        })
    }).ToList()

};

// =====
// STEP 4: Return structured anonymous object.
// At this point, EF Core has loaded all the required
related data
// lazily and efficiently, only when it was accessed.
//
=====

return Ok(result);

}

// POST api/<CustomerController>
//[HttpPost]

```

```

//public void Post([FromBody] string value)
//{
//}

// PUT api/<CustomerController>/5
//[HttpPut("{id}")]
//public void Put(int id, [FromBody] string value)
//{
//}

// DELETE api/<CustomerController>/5
//[HttpDelete("{id}")]
//public void Delete(int id)
//{
//}

}

}

```

Code Explanation

- EF Core loads the Customer entity first.
- Each navigation property (Profile, Addresses, Orders) triggers a separate SQL query the first time it's accessed.
- Subsequent access to the same property does not cause another query because EF Core caches the result in the current DbContext instance.

How to Disable Lazy Loading in EF Core?

There are several ways to disable lazy loading, depending on your requirements.

Remove proxy configuration from your DbContext setup:

```
// options.UseLazyLoadingProxies(); // Comment or remove this line
```

Disable it programmatically at runtime:

```
_context.ChangeTracker.LazyLoadingEnabled = false;
```

Remove the virtual keyword from navigation properties if you never intend to use lazy loading.

Disabling lazy loading is useful for debugging, preventing circular references, or optimizing performance when eager loading is preferred.

Drawbacks of Lazy Loading While convenient, Lazy Loading introduces some challenges:

1. **N + 1 Query Problem:** Accessing related data in a loop can result in dozens or hundreds of queries, one for each parent entity.
2. **Performance Overhead:** Creating proxy classes and intercepting property access adds a slight runtime cost.
3. **Serialization Issues:** Proxies maintain bidirectional references, which can cause infinite loops during JSON serialization.
4. **Debugging Difficulty:** Since EF Core silently fires queries behind the scenes, it can be harder to trace performance bottlenecks.

Logging – ASP.NET Core Web API

Start typing here...

Caching – ASP.NET Core Web API

Caching is the process of storing frequently accessed data in a temporary storage (known as the cache) so that subsequent requests for that data can be served faster. Instead of recalculating or re-fetching data from the database or an external API every time, the application retrieves the data from the cache if available. This reduces unnecessary computation and network calls, improving the performance and responsiveness of applications and reducing the load on database servers.

ASP.NET Core provides built-in caching mechanisms, making integrating caching into Web API projects easy. ASP.NET Core supports various caching strategies, such as:

- **In-Memory Caching:** Stores data on the same server where the application is running.
- **Distributed Caching:** Enables caching across multiple servers using external cache stores like Redis or SQL Server.
- **Response Caching:** Caches entire HTTP responses, particularly useful for web APIs that return data that doesn't change frequently.

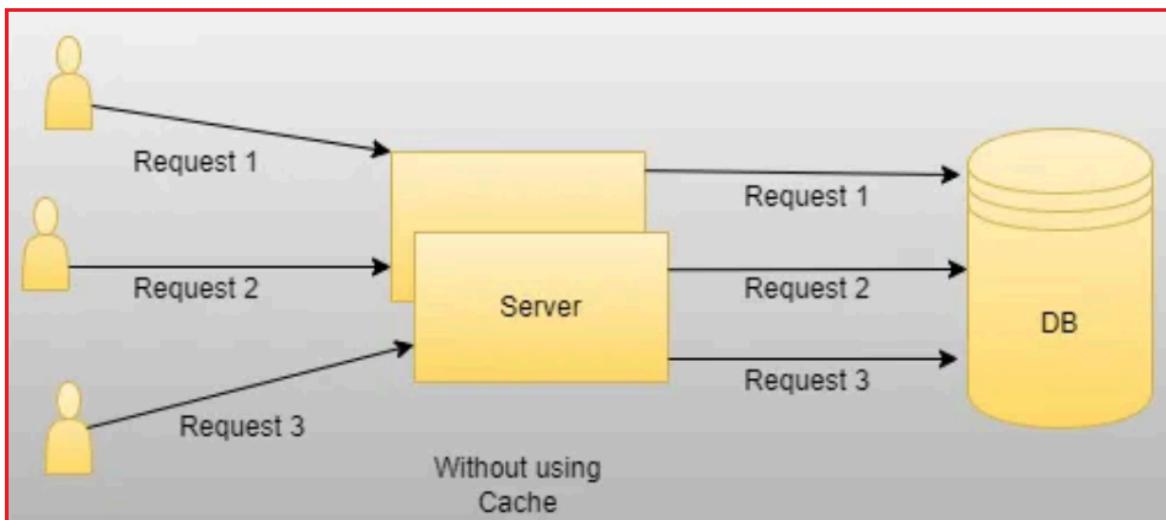
Why is Caching Important in ASP.NET Core Web API?

Caching is essential for improving the performance and scalability of ASP.NET Core Web APIs. It reduces the time required to serve client requests, decreases the load on backend services, and helps manage server resources more efficiently.

- **Performance and Speed:** By reducing round-trips to the database or external services, our API can respond to client requests more quickly.
- **Scalability:** When fewer resources are consumed, the server can handle more concurrent requests with the same hardware.
- **Cost Reduction:** If the database or third-party services charge per request or resource usage, especially in cloud-based environments where such operations may incur additional charges, caching can reduce costs by reducing the number of network calls.
- **Enhanced User Experience:** Faster response times lead to a better user experience, which is key for user retention and satisfaction.

Without Caching:

Each incoming request triggers the full processing pipeline. For example, a database query might be executed every time a user requests a particular resource, resulting in repeated I/O operations, increased load on the database server, and higher resource consumption, which can lead to slower performance under high traffic. That means every time a client requests data, the application must fetch it from the primary data source (e.g., a database or external API), perform any necessary computations, and then return the result. For a better understanding, please look at the following diagram:



image_295.png

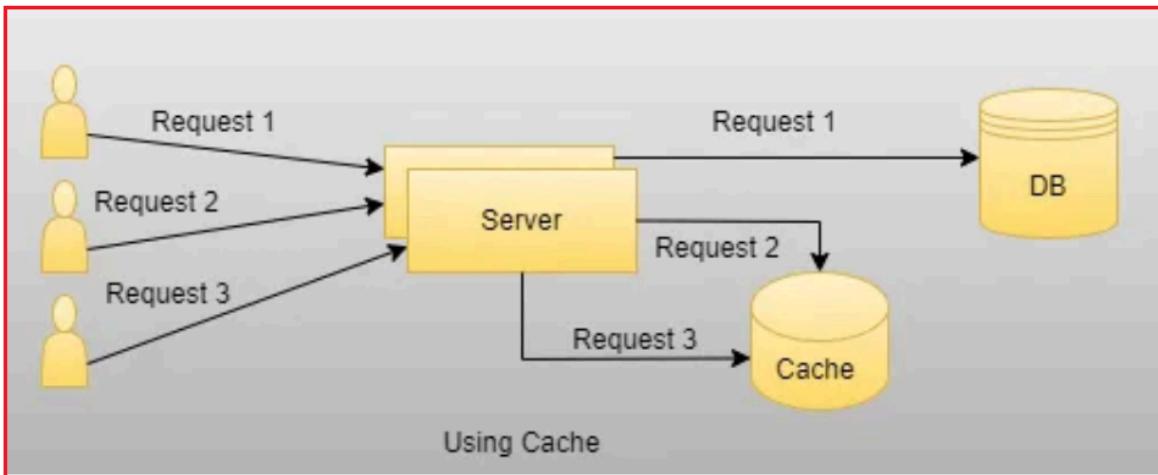
Without Caching

- The client sends a request to the Web API.
- Web API receives the request and queries the database or an external service.
- The database processes the query and returns the data.
- Web API returns data to the client.
- Every request involves a call to the database or external service, even if the requested data is the same.

With Caching:

With Caching, when a request is made, the application first checks if the data is available in the cache. If the data is found (a cache hit), it is returned immediately, bypassing the

need for a full data fetch. If the data is not in the cache (a cache miss), it is retrieved from the primary data source and stored in the cache for future requests. By serving cached data, the system reduces the number of calls to the backend systems. This leads to faster response times and more efficient resource usage. For a better understanding, please look at the following diagram:



image_296.png

With Caching

- The client sends a request to the Web API.
- Web API checks if the requested data is already in the cache.
 - If the data exists in the cache (cache hit), return it immediately from the cache.
 - If the data does not exist in the cache (cache miss), make a call to the database or external service, retrieve the data, and store it in the cache.
- Web API returns the data to the client.
- The data is quickly retrieved from the cache for subsequent requests, bypassing expensive operations.

With caching, the Web API only fetches data from the database or external service on the first request or when the cache expires/invalidates.

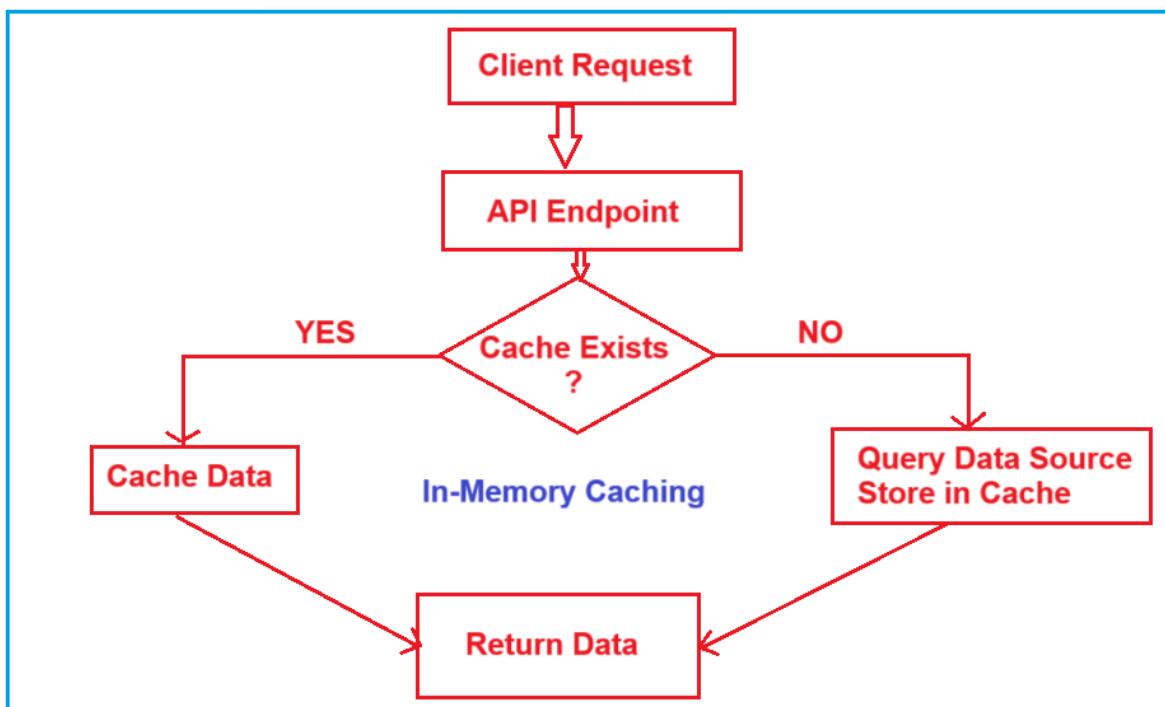
What is a Cache Hit and a Cache Miss?

- **Cache Hit:** When a requested item is found in the cache, it's referred to as a cache hit. The application can quickly retrieve and return the data without performing additional processing. No external or database call is required, resulting in a very fast retrieval.
- **Cache Miss:** When a requested item is not found in the cache, it's called a cache miss. In this case, the application must fetch the data from the original source (e.g., database or external API) and store it in the cache for future use.

In-Memory Caching in ASP.NET Core Web API

In-Memory Caching in ASP.NET Core is a mechanism for storing frequently accessed data in the server's memory (RAM) to improve application performance and reduce database load. When a client makes a request that involves data retrieval, especially data that does not change often (e.g., master data like countries, states, cities), caching that data in memory helps the application to quickly serve subsequent requests without repeatedly querying the database.

For a better understanding of How In-Memory Caching Works in ASP.NET Core Web API Applications, please have a look at the following diagram:



image_297.png

Work Flow of In-Memory Caching in Web API:

- **Client Request:** The process starts when a client sends a request to the API.
- **API Endpoint:** The request is received by the API endpoint, which then processes the incoming request.

- **Cache Check (“Cache Exists?”):** The API checks whether the requested data is available in the in-memory cache.
 - If the data exists, the API retrieves it from the cache.
 - If the data is not found in the cache, the API queries the data source. After fetching, it stores the result in the cache.
- **Return Data:** Finally, the data is returned to the client.

Redis Cache in ASP.NET Core Web API

Distributed Caching is a caching mechanism where data is stored in an external cache system that can be accessed by multiple application servers simultaneously. That means Distributed caching (such as Redis, NCache, etc.) allows multiple application instances to share cached data. Unlike an in-memory cache that is local to one application instance, a distributed cache is typically hosted on a dedicated cache server, i.e., a distributed cache stores data in an external system. All application servers connect to this cache server (external system) to get/set cached data.

When Should We Use a Distributed Cache Instead of In-Memory Cache?

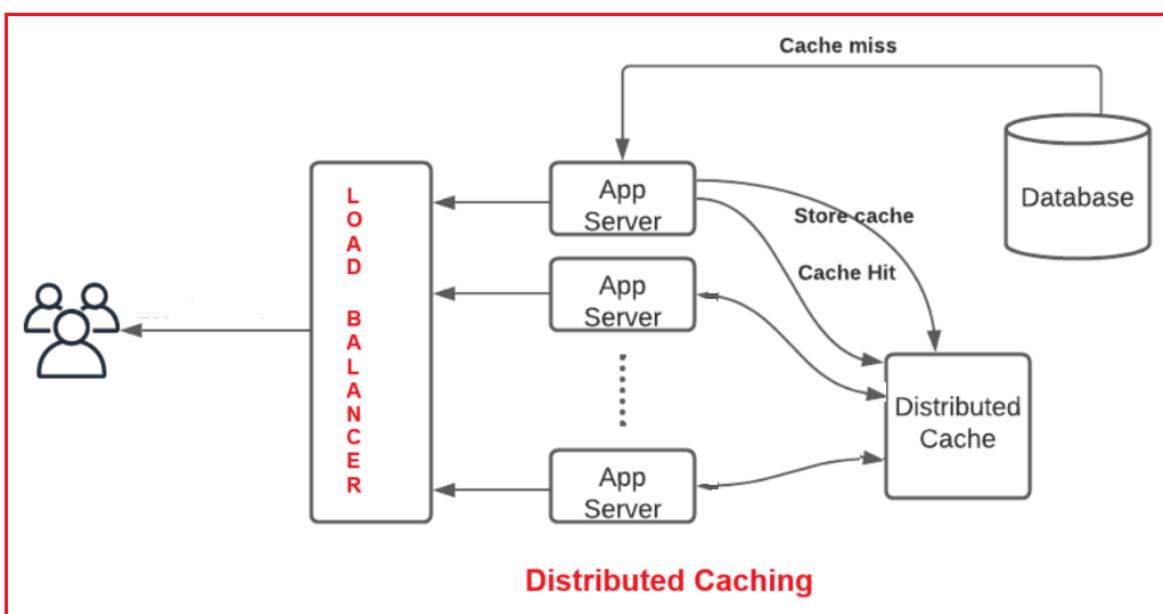
In-memory caching is fast but is limited to a single application instance. If the application is deployed across multiple servers or containers, each instance will have its own separate in-memory cache. This can lead to issues such as cache duplication, increased memory usage, and inconsistent data. In these cases, distributed caching is preferred because it provides a centralized cache that all application instances can access. Additionally, distributed caching is more reliable if an application instance goes down, as the cached data remains available in the external cache-store. So, we need to use a Distributed Cache Instead of In-Memory Cache in the following scenarios:

- **Multiple Server Instances:** If your application runs on multiple servers behind a load balancer, you need a cache that all servers can share. An in-memory cache would be unique to each server, causing inconsistent data or missing data if the request is routed to a different server.
- **Memory Constraints:** Maintaining large caches in each server's memory can be expensive. Offloading the cache to an external system can be more cost-effective and memory-efficient on the app servers.
- **Persistence Across Restarts:** In-memory caches are lost whenever the application restarts. Distributed cache solutions can offer persistence and advanced features like replication, meaning your data remains available even if an app server goes down.

- **High Scalability:** In scenarios with multiple application servers, using an in-memory cache can lead to data inconsistency or duplication because each server holds its own cache. A distributed cache maintains a single source of cached data accessible to all servers.
- **High Availability:** With load balancing, a distributed cache ensures that a cache hit is possible regardless of which server handles the request.

Distributed Caching Architecture:

Let us first understand the architecture of Distributed Caching, and then we will see how to implement Distributed caching using Redis cache in our ASP.NET Core Web API Application. In a distributed caching architecture, several components work together to manage, retrieve, and store cached data efficiently. These components include users, a load balancer, application servers, the distributed cache, and the database. To understand the architecture of Distributed Caching, please have a look at the following diagram.



image_298.png

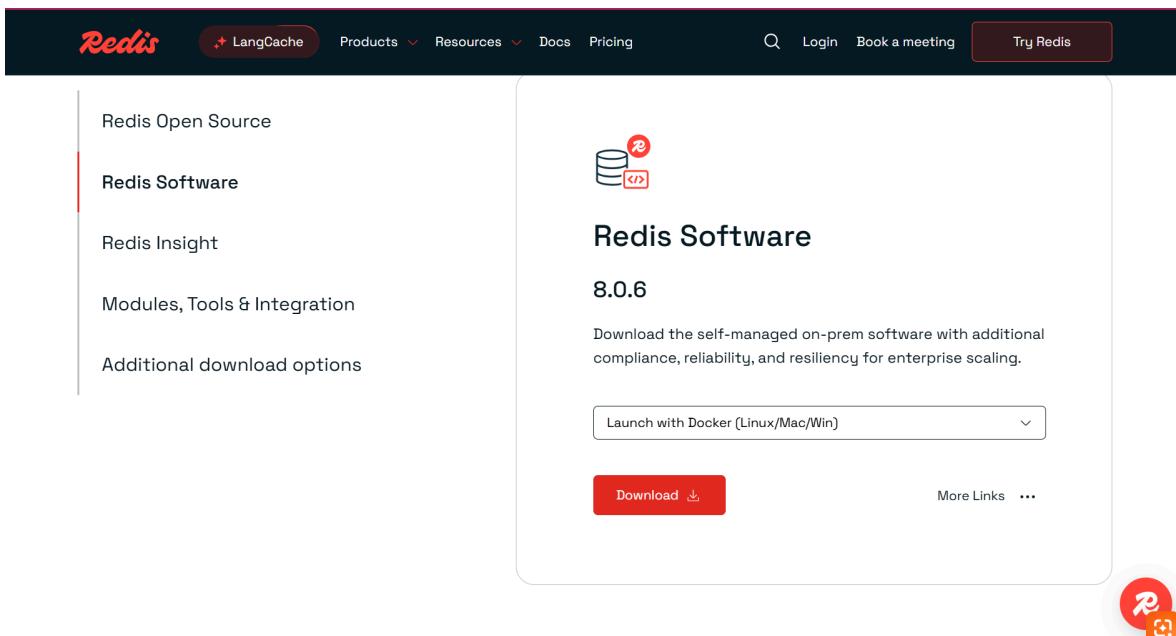
How Distributed Caching Works:

- **User Request:** Users send a request to the application, which first passes through the load balancer.

- **Load Balancer:** The load balancer routes the request to one of the application servers.
- **Cache Check:** The application server checks if the data is available in the distributed cache.
 - **Cache Hit:** If the requested data is found (cache hit) in the distributed cache, the data is retrieved quickly from the cache.
 - **Cache Miss:** If the data is not found in the cache, it is called a Cache Miss. In this case, the App Server queries the Database to retrieve the data.
- **Store Data in Cache:** After retrieving the data from the database, the app server stores it in the distributed cache. Subsequent requests for the same data can be served from the cache instead of the database.
- **Return Data:** The data, whether retrieved from the cache or database, is returned to the user.

Running Redis in docker

To download the latest Redis image (<https://redis.io/downloads/>), run the following command:



image_300.png

```
docker pull redis:latest
```

If you wish to pull a Specific Version of Redis:

```
docker pull redis:8.0
```

Verify the image has been downloaded

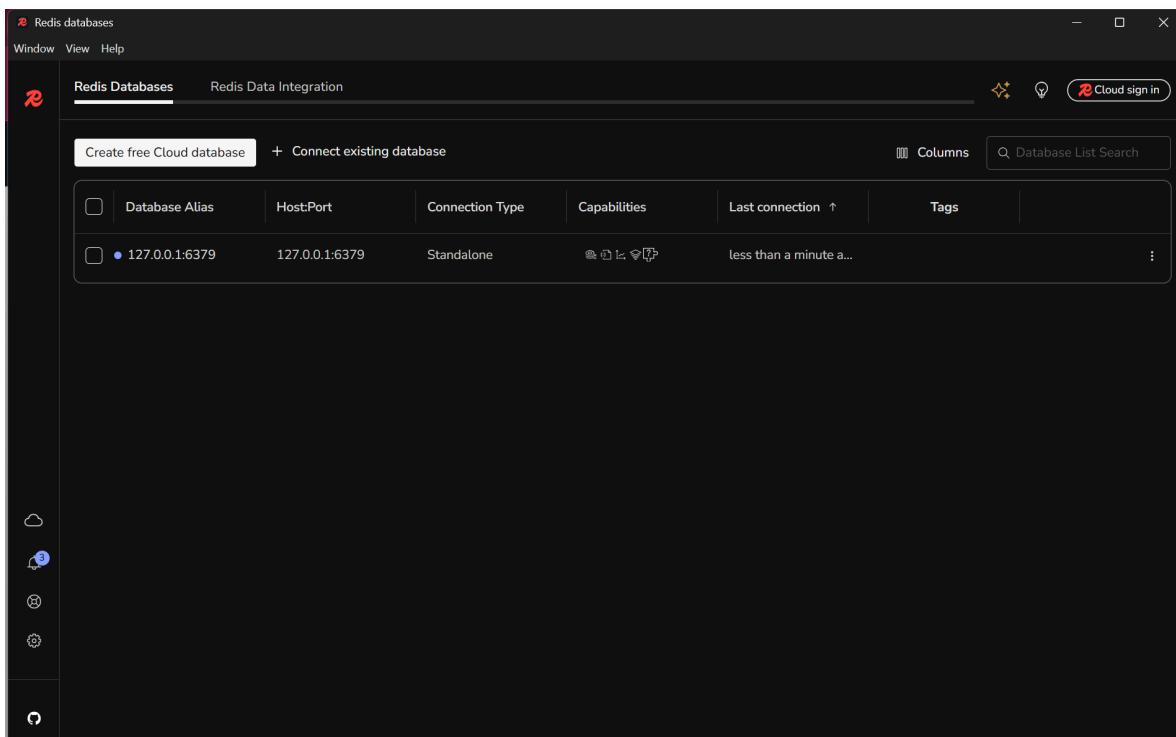
```
docker images
```

Once the image is downloaded, you can proceed to run the Redis container as you initially intended:

```
docker run -d --name redis -p 6379:6379 redis:latest
```

This will run the Redis container in detached mode with the latest image.

Download Redis Insight (<https://redis.io/downloads/>) app to view your redis cache db



image_299.png

Project setup

Create a new ASP.NET Core Web API project named `RedisCachingDemo`. Once you create the project, we need to add the following Packages for Entity Framework Core to work with the SQL Server database and Redis cache. You can install these packages using the following commands in the Package Manager Console.

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer  
Install-Package Microsoft.EntityFrameworkCore.Tools  
Install-Package Microsoft.Extensions.Caching.StackExchangeRedis
```

⚠ To connect and start caching data from .NET Core applications with Redis cache, we need `Microsoft.Extensions.Caching.StackExchangeRedis` package.

Define the Database Model and DbContext

`Product.cs`

```
namespace RedisCachingDemo.Models  
{  
    public class Product  
    {  
        public int Id { get; set; }  
        public string Name { get; set; }  
        public string Category { get; set; }  
        public int Price { get; set; }  
        public int Quantity { get; set; }  
    }  
}
```

Create Db Context:

Next, we need to create the `DbContext` class. First, create a new folder named `Data` in the project root directory. Then, inside the `Data` folder, create a class file named `ApplicationDbContext.cs`

`ApplicationDbContext.cs`

```
using Microsoft.EntityFrameworkCore;  
using RedisCachingDemo.Models;
```

```
namespace RedisCachingDemo.Data
{
    public class ApplicationDbContext : DbContext
    {
        public
        ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }
        // Override this method to configure the model and seed initial
        data
        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            // Seed initial data for testing purposes.
            // Each Product instance must have a unique Id.
            var initialProducts = new List<Product>
            {
                new Product
                {
                    Id = 1,
                    Name = "Apple iPhone 14",
                    Category = "Electronics",
                    Price = 999,
                    Quantity = 50
                },
                new Product
                {
                    Id = 2,
                    Name = "Samsung Galaxy S22",
                    Category = "Electronics",
                    Price = 899,
                    Quantity = 40
                },
                new Product
                {

```

```

        Id = 3,
        Name = "Sony WH-1000XM4 Headphones",
        Category = "Electronics",
        Price = 349,
        Quantity = 30
    },
    new Product
    {
        Id = 4,
        Name = "Nike Air Zoom Pegasus",
        Category = "Footwear",
        Price = 120,
        Quantity = 100
    },
    new Product
    {
        Id = 5,
        Name = "Adidas Ultraboost",
        Category = "Footwear",
        Price = 180,
        Quantity = 80
    },
    new Product
    {
        Id = 6,
        Name = "Organic Apples (1kg)",
        Category = "Groceries",
        Price = 4,
        Quantity = 200
    },
    new Product
    {
        Id = 7,
        Name = "Organic Bananas (1 Dozen)",
        Category = "Groceries",
        Price = 3,
        Quantity = 150
    }
}

```

```

    };
    // Instruct EF Core to seed this data into the 'Products'
    table
        modelBuilder.Entity<Product>().HasData(initialProducts);
    }
    // This DbSet maps to a Products table in the database
    public DbSet<Product> Products { get; set; }
}

```

Configure Connection String and Redis Cache Settings:

Next, please modify the `appsettings.json` file as follows. Here, the Entity Framework Core will create the database named `redis_caching_demo_db` if it has not already been created in the SQL server. Also, we have added a section specifying the Redis connection details (host and port) and an instance name (prefix) to avoid key collisions. Please remember that 6379 is the Port number on which our Redis Server is running.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=localhost;Initial Catalog=redis_caching_demo_db;Integrated Security=True;Pooling=False;Encrypt=True;Trust Server Certificate=True"
  },
  "RedisCacheOptions": {
    "Configuration": "localhost:6379",
    "InstanceName": "RedisCachingDemo"
  }
}
```

RedisCacheOptions

This is the section under which all Redis-related configurations are specified. We can name this section anything.

- “**Configuration**”: “localhost:6379”: This key specifies the connection string for the Redis server. Here, localhost is the host where the Redis server is running. localhost also indicates that Redis is running on the same machine as your application. If Redis were hosted elsewhere, we would specify the IP address or hostname of that server. 6379 is the port number on which the Redis server listens for incoming connections.
- “**InstanceName**”: “RedisCachingDemo”: The InstanceName is a prefix used for all entries created by our application’s instance. This can be useful when multiple applications use the same Redis server, and we want to avoid key collisions and make it easier to identify which ones belong to which application. In this case, every key stored in Redis by this application will be prefixed with RedisCachingDemo.

Configure the DbContext and Redis Cache in the Program Class:

Next, we need to configure our application to support Redis cache, and for this purpose, we need to register the **AddStackExchangeRedisCache** service. So, please modify the Program class as follows. The following code is self-explained, so please read the comment lines for a better understanding.

```
using Microsoft.EntityFrameworkCore;
using RedisCachingDemo.Data;
using StackExchange.Redis;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
builder.Services.AddOpenApi();

builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
```

```

        // This will use the property names as defined in the C#
model
            options.JsonSerializerOptions.PropertyNamingPolicy =
null;
        });

// Configure DbContext to Use SQL Server Database
builder.Services.AddDbContext<ApplicationContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultC
onnection")));

// Register Redis distributed cache
builder.Services.AddStackExchangeRedisCache(options =>
{
    //This property is set to specify the connection string for Redis
    //The value is fetched from the application's configuration system,
    i.e., appsettings.json file
    options.Configuration =
builder.Configuration["RedisCacheOptions:Configuration"];
    //This property helps in setting a logical name for the Redis cache
instance.
    //The value is also fetched from the appsettings.json file
    options.InstanceName =
builder.Configuration["RedisCacheOptions:InstanceName"];
});
// Register the Redis connection multiplexer as a singleton service
// This allows the application to interact directly with Redis for
advanced scenarios

// Establish a connection to the Redis server using the configuration
from appsettings.json
builder.Services.AddSingleton<IConnectionMultiplexer>(sp =>
ConnectionMultiplexer.Connect(builder.Configuration["RedisCacheOptions:C
onfiguration"]));

var app = builder.Build();

// Configure the HTTP request pipeline.

```

```
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Create and Apply Database Migration:

```
# Create migration
dotnet ef migrations add AddNewFeature

# Apply to database
dotnet ef database update
```

EF Core will:

- Create the database `redis_caching_demo_db` in our sql server.
- Create all tables.
- Insert the seed data defined in `AppDbContext`.

How to Use Redis for Cache in ASP.NET Core Web API:

Once the Redis server settings are configured in the `Program` class, we can inject the `IDistributedCache` service instance into our controllers or services to interact with Redis. The `IDistributedCache` interface provides `GetStringAsync`, `SetStringAsync`, and `RemoveAsync` methods in ASP.NET Core when Redis is used as the underlying cache provider.

`SetStringAsync`:

This method stores a string value in the cache with an associated key and optional caching options (such as expiration policies). When we call SetStringAsync, it converts our string data into a byte array and stores it in Redis under the specified key.

```
// Fetch products from the database.  
products = await _context.Products.AsNoTracking().ToListAsync();  
  
// Serialize your data to a string (e.g., JSON) before caching  
string serializedData = JsonSerializer.Serialize(products);  
  
// Define caching options (e.g., sliding expiration of 5 minutes)  
var cacheOptions = new DistributedCacheEntryOptions  
{  
    SlidingExpiration = TimeSpan.FromMinutes(5)  
};  
  
// Store the serialized data in the cache asynchronously  
await _cache.SetStringAsync("SomeCacheKey", serializedData, cacheOptions);
```

image_301.png

How It Works:

- We need to call **SetStringAsync(key, value, options)** to cache data.
- The method serializes the string value (if required) and sends it to Redis to be stored.
- The **DistributedCacheEntryOptions** allows us to set parameters like absolute or sliding expiration. This means the cached item can be automatically removed after a set duration or if it hasn't been accessed for a specified period.
- The cache is updated with the new value if the key already exists.

GetStringAsync:

This method retrieves a cached value (stored as a string) associated with the provided key. We need to use this method to quickly check if the data is available in the cache before querying a slower data source like a database. After retrieval, we must deserialize the string to its original object type. The syntax is given below:

```

// Attempt to retrieve the cached data by its key asynchronously
var cachedData = await _cache.GetStringAsync("SomeCacheKey");

// If the cache returns data, it can be deserialized and used directly
if (!string.IsNullOrEmpty(cachedData))
{
    // Process the cached data (e.g., deserialize it)
}
else
{
    // Cache miss: retrieve data from the database and cache it for future requests
}

```

image_302.png

How It Works:

- When we call GetStringAsync(key), it sends an asynchronous request to the Redis server to fetch the value associated with the key.
- If the key exists, it returns the corresponding string value (typically serialized data like JSON).
- If the key is not found (a cache miss), it returns null.

RemoveAsync:

This method deletes the cache entry in Redis based on the specified key. We need to use this method to maintain cache consistency. For instance, if a product is updated or deleted, remove its cache entry so that the next read operation fetches fresh data. The syntax is given below:

```

// Remove the cached data for a specific key asynchronously
await _cache.RemoveAsync("SomeCacheKey");

// After removal, a subsequent GetStringAsync call will return null (cache miss)

```

image_303.png

How It Works:

- When you call RemoveAsync(key), it instructs Redis to delete the data associated with the provided key. The removal is performed asynchronously, allowing your application to continue processing while the cache entry is being deleted.

- This is useful for cache invalidation when the underlying data has changed (e.g., after a database update or delete operation).

Using Redis Cache in ASP.NET Core Web API Controller:

To better understand, please create a new API Empty controller named **ProductsController** within the Controllers folder. The following code is self-explained, so please read the comment lines for a better understanding.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Caching.Distributed;
using RedisCachingDemo.Data;
using RedisCachingDemo.Models;
using System.Text.Json;

// For more information on enabling Web API for empty projects, visit
https://go.microsoft.com/fwlink/?LinkID=397860

namespace RedisCachingDemo.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        private readonly ApplicationDbContext _context;
        private readonly IDistributedCache _cache;
        // Inject ApplicationDbContext and IDistributedCache via
constructor.

        public ProductsController(ApplicationDbContext context,
IDistributedCache cache)
        {
            _context = context;
            _cache = cache;
        }

        // GET: api/products/all
        [HttpGet("all")]
    }
}
```

```

public async Task<IActionResult> GetAll()
{
    var cacheKey = "GET_ALL_PRODUCTS";
    List<Product> products;

    try
    {
        // Attempt to retrieve the product list from Redis
cache.
        var cachedData = await
_cache.GetStringAsync(cacheKey);
        if (!string.IsNullOrEmpty(cachedData))
        {
            // Deserialize JSON string back to
List<Product>.
            products =
JsonSerializer.Deserialize<List<Product>>(cachedData) ?? new
List<Product>();
        }
        else
        {
            // Cache miss: fetch products from the database.
            products = await
_context.Products.AsNoTracking().ToListAsync();
            if (products != null)
            {
                // Serialize the product list to a JSON
string.
                var serializedData =
JsonSerializer.Serialize(products);
                // Define cache options (using sliding
expiration).
                var cacheOptions = new
DistributedCacheEntryOptions()

                .SetSlidingExpiration(TimeSpan.FromMinutes(5));
                // Store the serialized data in Redis.
                await _cache.SetStringAsync(cacheKey,

```

```

        serializedData, cacheOptions);
    }
}
return Ok(products);
}
catch (Exception ex)
{
    // Return a 500 response if any error occurs.
    return StatusCode(500, new { message = "An error
occurred while retrieving products.", details = ex.Message });
}
}

// GET: api/products/Category?Category=Fruits
[HttpGet("Category")]
public async Task<IActionResult> GetProductByCategory(string
Category)
{
    // Cache key includes the category to ensure unique cache
entries.
    var cacheKey = $"PRODUCTS_{Category}";
    List<Product> products;
    try
    {
        var cachedData = await _cache.GetStringAsync(cacheKey);
        if (!string.IsNullOrEmpty(cachedData))
        {
            products = JsonSerializer.Deserialize<List<Product>>
(cachedData) ?? new List<Product>();
        }
        else
        {
            // Cache miss: fetch from the database by matching
category.
            products = await _context.Products
                .Where(prd => prd.Category.ToLower() ==
Category.ToLower())
                .AsNoTracking()
        }
    }
    catch (Exception ex)
    {
        // Return a 500 response if any error occurs.
        return StatusCode(500, new { message = "An error
occurred while retrieving products.", details = ex.Message });
    }
}
}

```

```

        .ToListAsync();
    if (products != null)
    {
        var serializedData =
JsonSerializer.Serialize(products);
        // Use absolute expiration so that the cache
entry expires after 5 minutes.
        var cacheOptions = new
DistributedCacheEntryOptions()
        .SetAbsoluteExpiration(TimeSpan.FromMinutes(5));
        await _cache.SetStringAsync(cacheKey,
serializedData, cacheOptions);
    }
}
return Ok(products);
}
catch (Exception ex)
{
    return StatusCode(500, new { message = "An error
occurred while retrieving products.", details = ex.Message });
}
}
// GET: api/products/{id}
[HttpGet("{id}")]
public async Task<IActionResult> GetProduct(int id)
{
    // Cache key for a single product.
    var cacheKey = $"Product_{id}";
    Product? product;
    try
    {
        var cachedData = await _cache.GetStringAsync(cacheKey);
        if (!string.IsNullOrEmpty(cachedData))
        {
            product = JsonSerializer.Deserialize<Product>
(cachedData) ?? new Product();
        }
    }
}

```

```

        else
        {
            // Fetch from database if not present in cache.
            product = await _context.Products.FindAsync(id);
            if (product == null)
                return NotFound($"Product with ID {id} not
found.");
            var serializedData =
JsonSerializer.Serialize(product);
            await _cache.SetStringAsync(cacheKey,
serializedData, new DistributedCacheEntryOptions
{
            SlidingExpiration = TimeSpan.FromMinutes(5)
});
        }
        return Ok(product);
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = "An error
occurred while retrieving the product.", details = ex.Message });
    }
}
// PUT: api/products/{id}
[HttpPut("{id}")]
public async Task<IActionResult> UpdateProduct(int id,
[FromBody] Product updatedProduct)
{
    if (id != updatedProduct.Id)
    {
        return BadRequest("Product ID mismatch.");
    }
    try
    {
        var existingProduct = await
_context.Products.FindAsync(id);
        if (existingProduct == null)
            return NotFound($"Product with ID {id} not found.");

```

```

        // Update product details in the database.

_context.Entry(existingProduct).CurrentValues.SetValues(updatedProduct);
    await _context.SaveChangesAsync();
    // Update the cache for this product.
    var cacheKey = $"Product_{id}";
    var serializedData =
JsonSerializer.Serialize(updatedProduct);
    await _cache.SetStringAsync(cacheKey, serializedData,
new DistributedCacheEntryOptions
{
    SlidingExpiration = TimeSpan.FromMinutes(5)
});
    return Ok();
}
catch (Exception ex)
{
    return StatusCode(500, new { message = "An error
occurred while updating the product.", details = ex.Message });
}
}

// DELETE: api/products/{id}
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteProduct(int id)
{
    try
    {
        var product = await _context.Products.FindAsync(id);
        if (product == null)
            return NotFound($"Product with ID {id} not found.");
        // Remove product from the database.
        _context.Products.Remove(product);
        await _context.SaveChangesAsync();
        // Remove product from the cache.
        var cacheKey = $"Product_{id}";
        await _cache.RemoveAsync(cacheKey);
        return Ok();
    }
}

```

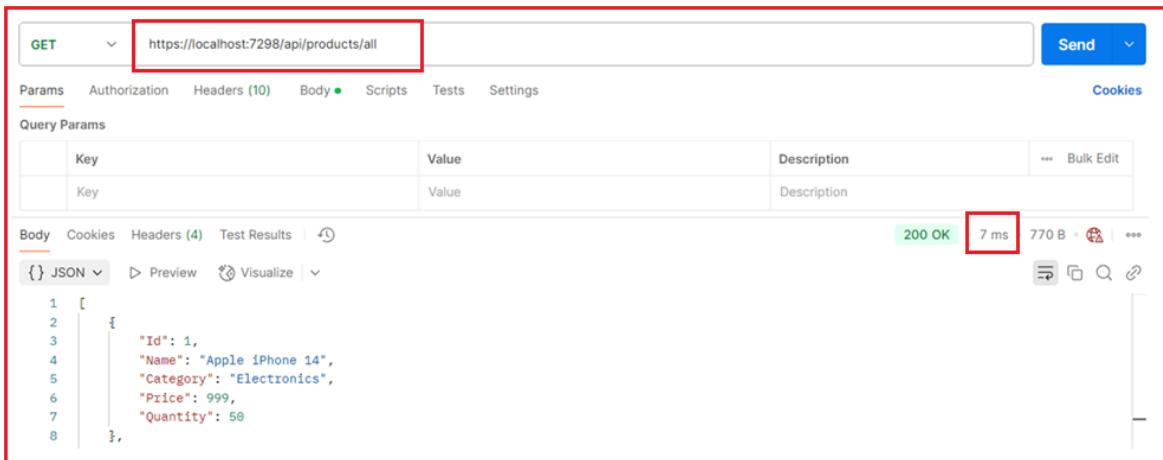
```

        catch (Exception ex)
    {
        return StatusCode(500, new { message = "An error
occurred while deleting the product.", details = ex.Message });
    }
}
}

```

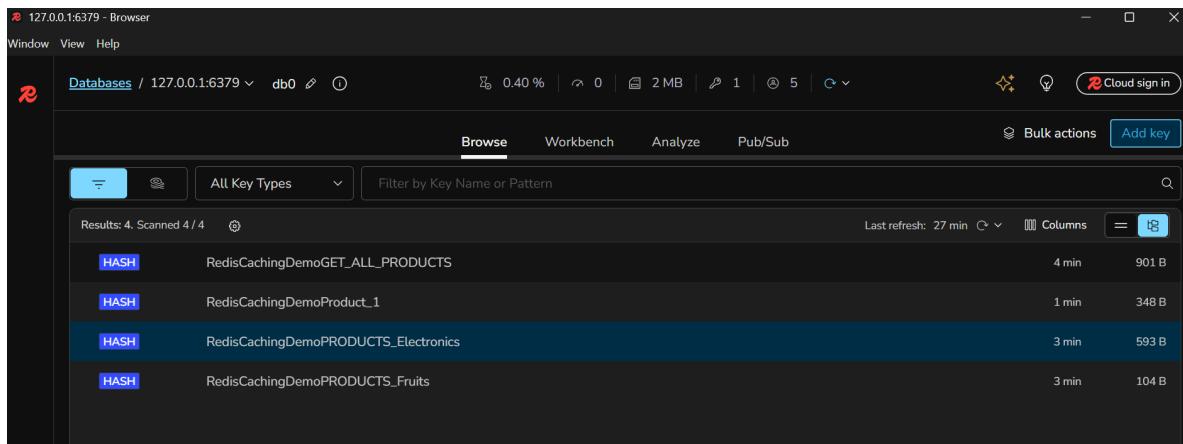
Testing

Run the application and test that the Redis Cache is working as expected. Let's try to access the API endpoint in Postman. The first request will take slightly longer to execute, but subsequent requests will improve the response time considerably. For me, it came down to just **7 milliseconds**, which is quite impressive.



image_304.png

check the redis Insight app to see the cached data



The screenshot shows the Redis desktop manager interface. At the top, there's a header bar with the title "127.0.0.1:6379 - Browser", a "Window" menu, and a "Help" option. Below the header is a navigation bar with tabs for "Databases" (selected), "127.0.0.1:6379", "db0", and a refresh icon. The main area has tabs for "Browse", "Workbench", "Analyze", and "Pub/Sub". On the right, there are buttons for "Cloud sign in", "Bulk actions", and "Add key". A search bar at the top of the main content area says "Filter by Key Name or Pattern". Below it, a message says "Results: 4. Scanned 4 / 4". The main content table lists four key types:

Type	Name	Last refresh	Size
HASH	RedisCachingDemoGET_ALL_PRODUCTS	4 min	901 B
HASH	RedisCachingDemoProduct_1	1 min	348 B
HASH	RedisCachingDemoPRODUCTS_Electronics	3 min	593 B
HASH	RedisCachingDemoPRODUCTS_Fruits	3 min	104 B

image_305.png

Stored Procedures in EFCore

Start typing here...

Fluent Validation

Installation

Using the NuGet package manager console within Visual Studio run the following command:

```
Install-Package FluentValidation
```

Or using the .net core CLI from a terminal window:

```
dotnet add package FluentValidation
```

Creating your first validator

To define a set of validation rules for a particular object, you will need to create a class that inherits from `AbstractValidator<T>`, where `T` is the type of class that you wish to validate.

For example, imagine that you have a `Customer` class:

```
public class Customer
{
    public int Id { get; set; }
    public string Surname { get; set; }
    public string Forename { get; set; }
    public decimal Discount { get; set; }
    public string Address { get; set; }
}
```

You would define a set of validation rules for this class by inheriting from `AbstractValidator<Customer>`:

```
using FluentValidation;

public class CustomerValidator : AbstractValidator<Customer>
```

```
{  
}
```

The validation rules themselves should be defined in the validator class's constructor.

To specify a validation rule for a particular property, call the `RuleFor` method, passing a lambda expression that indicates the property that you wish to validate. For example, to ensure that the `Surname` property is not null, the validator class would look like this:

```
using FluentValidation;  
  
public class CustomerValidator : AbstractValidator<Customer>  
{  
    public CustomerValidator()  
    {  
        RuleFor(customer => customer.Surname).NotNull();  
    }  
}
```

To run the validator, instantiate the validator object and call the `Validate` method, passing in the object to validate.

```
Customer customer = new Customer();  
CustomerValidator validator = new CustomerValidator();  
  
ValidationResult result = validator.Validate(customer);
```

The `Validate` method returns a `ValidationResult` object. This contains two properties:

- `IsValid` - a boolean that says whether the validation succeeded.
- `Errors` - a collection of `ValidationFailure` objects containing details about any validation failures.

The following code would write any validation failures to the console:

```
using FluentValidation.Results;
```

```
Customer customer = new Customer();
CustomerValidator validator = new CustomerValidator();

ValidationResult results = validator.Validate(customer);

if(! results.IsValid)
{
    foreach(var failure in results.Errors)
    {
        Console.WriteLine("Property " + failure.PropertyName + " failed validation. Error was: " + failure.ErrorMessage);
    }
}
```

Chaining validators

You can chain multiple validators together for the same property:

```
using FluentValidation;

public class CustomerValidator : AbstractValidator<Customer>
{
    public CustomerValidator()
    {
        RuleFor(customer => customer.Surname).NotNull().NotEqual("foo");
    }
}
```

This would ensure that the surname is not null and is not equal to the string ‘foo’.

Manual Validations – ASP.NET Core Web API

FluentValidation can be used within ASP.NET Core web applications to validate incoming models. There are several approaches for doing this:

- Manual validation
- Automatic validation (using the ASP.NET validation pipeline)

- Automatic validation (using a filter)

With manual validation, you inject the validator into your controller (or api endpoint), invoke the validator and act upon the result. This is the most straightforward approach and also the easiest to see what's happening.

With automatic validation, FluentValidation is invoked automatically by ASP.NET earlier in the pipeline which allows models to be validated before a controller action is invoked.

Getting started

The following examples will make use of a `Person` object which is validated using a `PersonValidator`. These classes are defined as follows:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public int Age { get; set; }
}

public class PersonValidator : AbstractValidator<Person>
{
    public PersonValidator()
    {
        RuleFor(x => x.Id).NotNull();
        RuleFor(x => x.Name).Length(0, 10);
        RuleFor(x => x.Email).EmailAddress();
        RuleFor(x => x.Age).InclusiveBetween(18, 60);
    }
}
```

If you're using MVC, Web Api or Razor Pages you'll need to register your validator with the Service Provider in the `ConfigureServices` method of your application's `Startup` class.

```
public void ConfigureServices(IServiceCollection services)
{
```

```
// If you're using MVC or WebApi you'll probably have  
// a call to AddMvc() or AddControllers() already.  
services.AddMvc();  
  
// ... other configuration ...  
  
services.AddScoped<IValidator<Person>, PersonValidator>();  
}
```

⚠ Note that you must register each validator as `IValidator<T>` where `T` is the type being validated. So if you have a `PersonValidator` that inherits from `AbstractValidator<Person>` then you should register it as `IValidator<Person>`

Here we use the `AddValidatorsFromAssemblyContaining` method from the `FluentValidation.DependencyInjectionExtension` package to automatically register all validators in the same assembly as `PersonValidator` with the service provider.

Now that the validators are registered with the service provider you can start working with either manual validation or automatic validation.

⚠ The auto-registration method used above uses reflection to scan one or more assemblies for validators. An alternative approach would be to use a source generator such as `AutoRegisterInject` to set up registrations.

Manual Validation

With the manual validation approach, you'll inject the validator into your controller (or Razor page) and invoke it against the model.

For example, you might have a controller that looks like this:

```
public class PeopleController : Controller  
{  
    private IValidator<Person> _validator;  
    private IPersonRepository _repository;  
  
    public PeopleController(IValidator<Person> validator,
```

```
IPersonRepository repository)
{
    // Inject our validator and also a DB context for storing our person
    // object.
    _validator = validator;
    _repository = repository;
}

public ActionResult Create()
{
    return View();
}

[HttpPost]
public async Task<IActionResult> Create(Person person)
{
    ValidationResult result = await _validator.ValidateAsync(person);

    if (!result.IsValid)
    {
        // Copy the validation results into ModelState.
        // ASP.NET uses the ModelState collection to populate
        // error messages in the View.
        result.AddModelError(this.ModelState);
    }

    // re-render the view when validation failed.
    return View("Create", person);
}

_repository.Save(person); //Save the person to the database, or some
other logic

TempData["notice"] = "Person successfully created";
return RedirectToAction("Index");
}
```

Because our validator is registered with the Service Provider, it will be injected into our controller via the constructor. We can then make use of the validator inside the `Create` action by invoking it with `ValidateAsync`.

If validation fails, we need to pass the error messages back down to the view so they can be displayed to the end user. We can do this by defining an extension method for FluentValidation's `ValidationResult` type that copies the error messages into ASP.NET's `ModelState` dictionary:

```
public static class Extensions
{
    public static void AddToModelState(this ValidationResult result,
    ModelStateDictionary ModelState)
    {
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(error.PropertyName, error.ErrorMessage);
        }
    }
}
```

This method is invoked inside the controller action in the example above.

Automatic Validation

Automatic validation instantiates and invokes a validator before the controller action is executed, meaning the `ModelState` will already be populated with validation results by the time your controller action is invoked. There are 2 implementations for this approach:

- Using ASP.NET's validation pipeline (no longer recommended)
- Using an Action Filter (supported by a 3rd party package)

Using the ASP.NET Validation Pipeline

The `FluentValidation.AspNetCore` package provides auto-validation for ASP.NET Core MVC projects by plugging into ASP.NET's validation pipeline.

With automatic validation using the validation pipeline, FluentValidation plugs into ASP.NET's built-in validation process that's part of ASP.NET Core MVC and allows models

to be validated before a controller action is invoked (during model-binding). This approach to validation is more seamless but has several downsides:

- **The ASP.NET validation pipeline is not asynchronous:** If your validator contains asynchronous rules then your validator will not be able to run. You will receive an exception at runtime if you attempt to use an asynchronous validator with auto-validation.
- **It is MVC-only:** This approach for auto-validation only works with MVC Controllers and Razor Pages. It does not work with the more modern parts of ASP.NET such as Minimal APIs or Blazor.
- **It is harder to debug:** The ‘magic’ nature of auto-validation makes it hard to debug/troubleshoot if something goes wrong as so much is done behind the scenes.



We no longer recommend using this approach for new projects but it is still available for legacy implementations.

Instructions for this approach can be found in the `FluentValidation.AspNetCore` package can be found on its project page here

(<https://github.com/FluentValidation/FluentValidation.AspNetCore#aspnet-core-integration-for-fluentvalidation>).

Minimal APIs

When using FluentValidation with minimal APIs, you can still register the validators with the service provider, (or you can instantiate them directly if they don’t have dependencies) and invoke them inside your API endpoint.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// Register validator with service provider (or use one of the automatic
// registration methods)
builder.Services.AddScoped<IValidator<Person>, PersonValidator>();

// Also registering a DB access repository for demo purposes
```

```

// replace this with whatever you're using in your application.
builder.Services.AddScoped<IPersonRepository, PersonRepository>();

app.MapPost("/person", async (IValidator<Person> validator,
IPersonRepository repository, Person person) =>
{
    ValidationResult validationResult = await
validator.ValidateAsync(person);

    if (!validationResult.IsValid)
    {
        return Results.ValidationProblem(validationResult.ToDictionary());
    }

    repository.Save(person);
    return Results.Created($"/{person.Id}", person);
});

```

Built-in Validators

FluentValidation ships with several built-in validators. The error message for each validator can contain special placeholders that will be filled in when the error message is constructed.

NotNull Validator

Ensures that the specified property is not null.

Example:

```
RuleFor(customer => customer.Surname).NotNull();
```

Example error: ‘Surname’ must not be empty.

String format args:

- {PropertyName} – Name of the property being validated
- {PropertyValue} – Current value of the property

- {PropertyPath} - The full path of the property

NotEmpty Validator

Ensures that the specified property is not null, an empty string or whitespace (or the default value for value types, e.g., 0 for int). When used on an IEnumerable (such as arrays, collections, lists, etc.), the validator ensures that the IEnumerable is not empty.

Example:

`RuleFor(customer => customer.Surname).NotEmpty();` Example error: ‘Surname’ should not be empty. String format args:

- {PropertyName} – Name of the property being validated
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

NotEqual Validator

Ensures that the value of the specified property is not equal to a particular value (or not equal to the value of another property).

Example:

`//Not equal to a particular value`

```
RuleFor(customer => customer.Surname).NotEqual("Foo");

//Not equal to another property

RuleFor(customer => customer.Surname).NotEqual(customer =>
customer.Forename);
```

Example error: ‘Surname’ should not be equal to ‘Foo’

String format args:

- {PropertyName} – Name of the property being validated
- {ComparisonValue} – Value that the property should not equal

- {ComparisonProperty} – Name of the property being compared against (if any)
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property
-

Optionally, a comparer can be provided to ensure a specific type of comparison is performed:

```
RuleFor(customer => customer.Surname).NotEqual("Foo",
StringComparer.OrdinalIgnoreCase);
```

An ordinal comparison will be used by default. If you wish to do a culture-specific comparison instead, you should pass StringComparer.CurrentCulture as the second parameter.

Equal Validator

Ensures that the value of the specified property is equal to a particular value (or equal to the value of another property).

Example:

```
//Equal to a particular value
RuleFor(customer => customer.Surname).Equal("Foo");

//Equal to another property
RuleFor(customer => customer.Password).Equal(customer =>
customer.PasswordConfirmation);
```

Example error: ‘Surname’ should be equal to ‘Foo’ String format args:

- {PropertyName} – Name of the property being validated
- {ComparisonValue} – Value that the property should equal
- {ComparisonProperty} – Name of the property being compared against (if any)
- {PropertyValue} – Current value of the property

- {PropertyName} – Name of the property

```
RuleFor(customer => customer.Surname).Equal("Foo",  
StringComparer.OrdinalIgnoreCase);
```

An ordinal comparison will be used by default. If you wish to do a culture-specific comparison instead, you should pass StringComparer.CurrentCulture as the second parameter.

Length Validator

Ensures that the length of a particular string property is within the specified range. However, it doesn't ensure that the string property isn't null.

Example:

```
RuleFor(customer => customer.Surname).Length(1, 250); //must be between  
1 and 250 chars (inclusive)
```

Example error: 'Surname' must be between 1 and 250 characters. You entered 251 characters.

Note: Only valid on string properties.

String format args:

- {PropertyName} – Name of the property being validated
- {MinLength} – Minimum length
- {MaxLength} – Maximum length
- {TotalLength} – Number of characters entered
- {PropertyValue} – Current value of the property
- {PropertyPath} – The full path of the property

MaxLength Validator

Ensures that the length of a particular string property is no longer than the specified value.

Example:

```
RuleFor(customer => customer.Surname).MaximumLength(250); //must be 250  
chars or fewer
```

Example error: The length of ‘Surname’ must be 250 characters or fewer. You entered 251 characters.

Note: Only valid on string properties.

String format args:

- {PropertyName} – Name of the property being validated
- {MaxLength} – Maximum length
- {TotalLength} – Number of characters entered
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

MinLength Validator

Ensures that the length of a particular string property is longer than the specified value.

Example:

```
RuleFor(customer => customer.Surname).MinimumLength(10); //must be 10  
chars or more
```

Example error: The length of ‘Surname’ must be at least 10 characters. You entered 5 characters.

Note: Only valid on string properties.

String format args:

- {PropertyName} – Name of the property being validated

- {MinLength} – Minimum length
- {TotalLength} – Number of characters entered
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

Less Than Validator

Ensures that the value of the specified property is less than a particular value (or less than the value of another property).

Example:

```
//Less than a particular value
RuleFor(customer => customer.CreditLimit).LessThan(100);

//Less than another property
RuleFor(customer => customer.CreditLimit).LessThan(customer =>
customer.MaxCreditLimit);
```

Example error: ‘Credit Limit’ must be less than 100.

Notes: Only valid on types that implement IComparable

String format args:

- {PropertyName} – Name of the property being validated
- {ComparisonValue} – Value to which the property was compared
- {ComparisonProperty} – Name of the property being compared against (if any)
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

Less Than Or Equal Validator

Ensures that the value of the specified property is less than or equal to a particular value (or less than or equal to the value of another property).

Example:

```
//Less than a particular value  
RuleFor(customer => customer.CreditLimit).LessThanOrEqualTo(100);  
  
//Less than another property  
RuleFor(customer => customer.CreditLimit).LessThanOrEqualTo(customer =>  
customer.MaxCreditLimit);
```

Example error: ‘Credit Limit’ must be less than or equal to 100. Notes: Only valid on types that implement IComparable

String format args:

- {PropertyName} – Name of the property being validated
- {ComparisonValue} – Value to which the property was compared
- {ComparisonProperty} – Name of the property being compared against (if any)
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

Greater Than Validator

Ensures that the value of the specified property is greater than a particular value (or greater than the value of another property).

Example:

```
//Greater than a particular value  
RuleFor(customer => customer.CreditLimit).GreaterThan(0);  
  
//Greater than another property  
RuleFor(customer => customer.CreditLimit).GreaterThan(customer =>  
customer.MinimumCreditLimit);
```

Example error: ‘Credit Limit’ must be greater than 0. Notes: Only valid on types that implement IComparable

String format args:

- {PropertyName} – Name of the property being validated
- {ComparisonValue} – Value to which the property was compared
- {ComparisonProperty} – Name of the property being compared against (if any)
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

Greater Than Or Equal Validator

Ensures that the value of the specified property is greater than or equal to a particular value (or greater than or equal to the value of another property).

Example:

```
//Greater than a particular value
RuleFor(customer => customer.CreditLimit).GreaterThanOrEqualTo(1);

//Greater than another property
RuleFor(customer => customer.CreditLimit).GreaterThanOrEqualTo(customer
=> customer.MinimumCreditLimit);
```

Example error: ‘Credit Limit’ must be greater than or equal to 1. Notes: Only valid on types that implement IComparable

String format args:

- {PropertyName} – Name of the property being validated
- {ComparisonValue} – Value to which the property was compared
- {ComparisonProperty} – Name of the property being compared against (if any)
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

Predicate Validator

(Also known as Must)

Passes the value of the specified property into a delegate that can perform custom validation logic on the value.

Example:

```
RuleFor(customer => customer.Surname).Must(surname => surname == "Foo");
```

Example error: The specified condition was not met for ‘Surname’

String format args:

- {PropertyName} – Name of the property being validated
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

Note that there is an additional overload for Must that also accepts an instance of the parent object being validated. This can be useful if you want to compare the current property with another property from inside the predicate:

```
RuleFor(customer => customer.Surname).Must((customer, surname) =>  
surname != customer.Forename)
```

Note that in this particular example, it would be better to use the cross-property version of NotEqual.

Regular Expression Validator

Ensures that the value of the specified property matches the given regular expression.

Example:

```
RuleFor(customer => customer.Surname).Matches("some regex here");
```

Example error: ‘Surname’ is not in the correct format. String format args:

- {PropertyName} – Name of the property being validated

- `{PropertyValue}` – Current value of the property
- `{RegularExpression}` – Regular expression that was not matched
- `{PropertyPath}` - The full path of the property

Email Validator

Ensures that the value of the specified property is a valid email address format.

Example:

```
RuleFor(customer => customer.Email).EmailAddress();
```

Example error: ‘Email’ is not a valid email address.

String format args:

- `{PropertyName}` – Name of the property being validated
- `{PropertyValue}` – Current value of the property
- `{PropertyPath}` - The full path of the property

The email address validator can work in 2 modes. The default mode just performs a simple check that the string contains an “@” sign which is not at the beginning or the end of the string. This is an intentionally naive check to match the behaviour of ASP.NET Core’s `EmailAddressAttribute`, which performs the same check. For the reasoning behind this, see this post:

From the comments:

“The check is intentionally naive because doing something infallible is very hard. The email really should be validated in some other way, such as through an email confirmation flow where an email is actually sent. The validation attribute is designed only to catch egregiously wrong values such as for a UI.” Alternatively, you can use the old email validation behaviour that uses a regular expression consistent with the .NET 4.x version of the ASP.NET `EmailAddressAttribute`. You can use this behaviour in `FluentValidation` by calling `RuleFor(x => x.Email).EmailAddress(EmailValidationMode.Net4xRegex)`. Note that this approach is

deprecated and will generate a warning as regex-based email validation is not recommended.

Note:

- ⚠** In FluentValidation 9, the ASP.NET Core-compatible “simple” check is the default mode. In FluentValidation 8.x (and older), the Regex mode is the default.

Credit Card Validator

Checks whether a string property could be a valid credit card number.

Example:

```
RuleFor(x => x.CreditCard).CreditCard();
```

Example error: ‘Credit Card’ is not a valid credit card number.

String format args:

- {PropertyName} – Name of the property being validated
- {PropertyValue} – Current value of the property
- {PropertyPath} - The full path of the property

Enum Validator

Checks whether a numeric value is valid to be in that enum. This is used to prevent numeric values from being cast to an enum type when the resulting value would be invalid. For example, the following is possible:

```
public enum ErrorLevel
{
    Error = 1,
    Warning = 2,
    Notice = 3
}
```

```
public class Model
{
    public ErrorLevel ErrorLevel { get; set; }

    var model = new Model();
    model.ErrorLevel = (ErrorLevel)4;
```

The compiler will allow this, but a value of 4 is technically not valid for this enum. The Enum validator can prevent this from happening.

```
RuleFor(x => x.ErrorLevel).IsInEnum();
```

Example error: ‘Error Level’ has a range of values which does not include ‘4’.

String format args:

- Name of the property being validated – Current value of the property – The full path of the property

Enum Name Validator

Checks whether a string is a valid enum name.

Example:

```
// For a case sensitive comparison
RuleFor(x => x.ErrorLevelName).IsEnumName(typeof(ErrorLevel));

// For a case-insensitive comparison
RuleFor(x => x.ErrorLevelName).IsEnumName(typeof(ErrorLevel),
caseSensitive: false);
```

Example error: ‘Error Level’ has a range of values which does not include ‘Foo’.

String format args:

- {PropertyName} – Name of the property being validated
- {PropertyValue} – Current value of the property

- `{PropertyPath}` - The full path of the property

Empty Validator

Opposite of the NotEmpty validator. Checks if a property value is null, or is the default value for the type. When used on an IEnumerable (such as arrays, collections, lists, etc.), the validator ensures that the IEnumerable is empty.

Example:

```
RuleFor(x => x.Surname).Empty();
```

Example error: ‘Surname’ must be empty.

String format args:

- `{PropertyName}` – Name of the property being validated
- `{PropertyValue}` – Current value of the property
- `{PropertyPath}` - The full path of the property

Null Validator

Opposite of the NotNull validator. Checks if a property value is null.

Example:

```
RuleFor(x => x.Surname).Null();
```

Example error: ‘Surname’ must be empty.

String format args:

- `{PropertyName}` – Name of the property being validated
- `{PropertyValue}` – Current value of the property
- `{PropertyPath}` - The full path of the property

ExclusiveBetween Validator

Checks whether the property value is in a range between the two specified numbers (exclusive).

Example:

```
RuleFor(x => x.Id).ExclusiveBetween(1,10);
```

Example error: 'Id' must be between 1 and 10 (exclusive). You entered 1.

String format args:

- {PropertyName} – Name of the property being validated
- {PropertyValue} – Current value of the property
- {From} – Lower bound of the range
- {To} – Upper bound of the range
- {PropertyPath} - The full path of the property

InclusiveBetween Validator

Checks whether the property value is in a range between the two specified numbers (inclusive).

Example:

```
RuleFor(x => x.Id).InclusiveBetween(1,10);
```

Example error: 'Id' must be between 1 and 10. You entered 0.

String format args:

- {PropertyName} – Name of the property being validated
- {PropertyValue} – Current value of the property
- {From} – Lower bound of the range
- {To} – Upper bound of the range
- {PropertyPath} - The full path of the property

PrecisionScale Validator

Checks whether a decimal value has the specified precision and scale.

Example:

```
RuleFor(x => x.Amount).PrecisionScale(4, 2, false);
```

Example error: ‘Amount’ must not be more than 4 digits in total, with allowance for 2 decimals. 5 digits and 3 decimals were found.

String format args:

- {PropertyName} – Name of the property being validated
- {PropertyValue} – Current value of the property
- {ExpectedPrecision} – Expected precision
- {ExpectedScale} – Expected scale
- {Digits} – Total number of digits in the property value
- {ActualScale} – Actual scale of the property value
- {PropertyPath} - The full path of the property



Note that the 3rd parameter of this method is ignoreTrailingZeros. When set to true, trailing zeros after the decimal point will not count towards the expected number of decimal places.

Example:

- When ignoreTrailingZeros is false then the decimal 123.4500 will be considered to have a precision of 7 and scale of 4
- When ignoreTrailingZeros is true then the decimal 123.4500 will be considered to have a precision of 5 and scale of 2.

Please also note that this method implies certain range of values that will be accepted. For example in case of `.PrecisionScale(3, 1)`, the method will accept values between -99.9 and 99.9, inclusive. Which means that integer part is always controlled to contain at most 3 - 1 digits, independently from `ignoreTrailingZeros` parameter.

⚠ Note that prior to FluentValidation 11.4, this method was called `ScalePrecision` instead and had its parameters reversed.

Custom Validators

There are several ways to create a custom, reusable validator. The recommended way is to make use of the Predicate Validator to write a custom validation function, but you can also use the Custom method to take full control of the validation process.

For these examples, we'll imagine a scenario where you want to create a reusable validator that will ensure a List object contains fewer than 10 items.

Predicate Validator

The simplest way to implement a custom validator is by using the Must method, which internally uses the `PredicateValidator`.

Imagine we have the following class:

```
public class Person {  
    public IList<Pet> Pets {get;set;} = new List<Pet>();  
}
```

To ensure our list property contains fewer than 10 items, we could do this:

```
public class PersonValidator : AbstractValidator<Person> {  
    public PersonValidator() {  
        RuleFor(x => x.Pets).Must(list => list.Count < 10)  
            .WithMessage("The list must contain fewer than 10 items");  
    }  
}
```

To make this logic reusable, we can wrap it an extension method that acts upon any List

type.

```
public static class MyCustomValidators {
    public static IRuleBuilderOptions<T, IList<TElement>>
ListMustContainFewerThan<T, TElement>(this IRuleBuilder<T,
IList<TElement>> ruleBuilder, int num) {
    return ruleBuilder.Must(list => list.Count < num).WithMessage("The
list contains too many items");
}
}
```

Here we create an extension method on `IRuleBuilder<T,TProperty>`, and we use a generic type constraint to ensure this method only appears in intellisense for List types. Inside the method, we call the Must method in the same way as before but this time we call it on the passed-in `RuleBuilder` instance. We also pass in the number of items for comparison as a parameter. Our rule definition can now be rewritten to use this method:

```
RuleFor(x => x.Pets).ListMustContainFewerThan(10);
```

Custom message placeholders

We can extend the above example to include a more useful error message. At the moment, our custom validator always returns the message “The list contains too many items” if validation fails. Instead, let’s change the message so it returns “‘Pets’ must contain fewer than 10 items.” This can be done by using custom message placeholders. FluentValidation supports several message placeholders by default including `{PropertyName}` and `{PropertyValue}`

We need to modify our extension method slightly to use a different overload of the Must method, one that accepts a `ValidationContext<T>` instance. This context provides additional information and methods we can use when performing validation:

```
public static IRuleBuilderOptions<T, IList<TElement>>
ListMustContainFewerThan<T, TElement>(this IRuleBuilder<T,
IList<TElement>> ruleBuilder, int num) {

    return ruleBuilder.Must((rootObject, list, context) => {
        context.MessageFormatter.AppendArgument("MaxElements", num);
    });
}
```

```

        return list.Count < num;
    })
    .WithMessage("{PropertyName} must contain fewer than {MaxElements}
items.");
}

```

Note that the overload of `Must` that we're using now accepts 3 parameters: the root (parent) object, the property value itself, and the context. We use the context to add a custom message replacement value of `MaxElements` and set its value to the number passed to the method. We can now use this placeholder as `{MaxElements}` within the call to `WithMessage`.

The resulting message will now be '`Pets`' must contain fewer than 10 items. We could even extend this further to include the number of elements that the list contains like this:

```

public static IRuleBuilderOptions<T, IList<TElement>>
ListMustContainFewerThan<T, TElement>(this IRuleBuilder<T,
IList<TElement>> ruleBuilder, int num) {

    return ruleBuilder.Must((rootObject, list, context) => {
        context.MessageFormatter
            .AppendArgument("MaxElements", num)
            .AppendArgument("TotalElements", list.Count);

        return list.Count < num;
    })
    .WithMessage("{PropertyName} must contain fewer than {MaxElements}
items. The list contains {TotalElements} element");
}

```

Writing a Custom Validator

If you need more control of the validation process than is available with `Must`, you can write a custom rule using the `Custom` method. This method allows you to manually create the `ValidationFailure` instance associated with the validation error. Usually, the framework does this for you, so it is more verbose than using `Must`.

```
public class PersonValidator : AbstractValidator<Person> {
    public PersonValidator() {
        RuleFor(x => x.Pets).Custom((list, context) => {
            if(list.Count > 10) {
                context.AddFailure("The list must contain 10 items or fewer");
            }
        });
    }
}
```

The advantage of this approach is that it allows you to return multiple errors for the same rule (by calling the `context.AddFailure` method multiple times). In the above example, the property name in the generated error will be inferred as “Pets”, although this could be overridden by calling a different overload of `AddFailure`:

```
context.AddFailure("SomeOtherProperty", "The list must contain 10 items
or fewer");
// Or you can instantiate the ValidationFailure directly:
context.AddFailure(new ValidationFailure("SomeOtherProperty", "The list
must contain 10 items or fewer"));
```

As before, this could be wrapped in an extension method to simplify the consuming code.

```
public static IRuleBuilderOptionsConditions<T, IList<TElement>>
ListMustContainFewerThan<T, TElement>(this IRuleBuilder<T,
IList<TElement>> ruleBuilder, int num) {

    return ruleBuilder.Custom((list, context) => {
        if(list.Count > 10) {
            context.AddFailure("The list must contain 10 items or fewer");
        }
    });
}
```

[Workshop] Fluent validation with manual validation

Let's Understand How to Implement Fluent API Validation in ASP.NET Core Web API. We will create a simple application to validate a Product model using the EF Core Database First approach.

Create a New ASP.NET Core Web API Project and Install Required NuGet Packages

First, create a new ASP.NET Core Web API Project named FluentAPIValidation and install the following Packages required for Fluent API validation and Entity Framework core. You can install the packages using the Package Manager Console by executing the following commands:

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer  
Install-Package Microsoft.EntityFrameworkCore.Tools  
Install-Package FluentValidation  
Install-Package Microsoft.EntityFrameworkCore.Design  
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

Create the Product Model

First, create a folder named Models in the project root directory where we will create our Models.

Product Model

Create a class file named `Product.cs`

```
using System.ComponentModel.DataAnnotations.Schema;  
  
namespace Fluent_API_Validation.Model  
{  
    public class Product  
    {  
        public int ProductId { get; set; }
```

```

// Stock Keeping Unit following a specific pattern (e.g., 8
uppercase letters/digits)
public string SKU { get; set; }

public string Name { get; set; }

[Column(TypeName = "decimal(8,2)")]
public decimal Price { get; set; }

public int Stock { get; set; }

public int CategoryId { get; set; }

public string? Description { get; set; }

// Discount percentage (0-100)
[Column(TypeName = "decimal(18,2)")]
public decimal Discount { get; set; }

// Manufacturing date (should not be in the future)
public DateTime ManufacturingDate { get; set; }

// Expiry date (must be after the manufacturing date)
public DateTime ExpiryDate { get; set; }

// Many-to-many relationship with Tag
public ICollection<Tag> Tags { get; set; } = new List<Tag>();
}

}

```

Tag Model

Each tag is stored as a separate entity. Products and Tags are connected via many-to-many relationships. So, create a class file named Tag.cs within the Models folder and copy and paste the following code.

```

namespace Fluent_API_Validation.Model
{
    public class Tag
    {
        public int TagId { get; set; }
        public string Name { get; set; }
        // Many-to-many relationship with Product
        public ICollection<Product> Products { get; set; } = new
List<Product>();
    }
}

```

DbContext Class:

First, create a folder named Data in the project root directory. Then, add a class file named `ECommerceDbContext.cs`

```

using Fluent_API_Validation.Model;
using Microsoft.EntityFrameworkCore;

namespace Fluent_API_Validation.Data
{
    public class ECommerceDbContext : DbContext
    {
        public ECommerceDbContext(DbContextOptions<ECommerceDbContext>
options) : base(options) { }

        public DbSet<Product> Products { get; set; }
        public DbSet<Tag> Tags { get; set; }
        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            // Configure many-to-many relationship between Product and
            Tag using an implicit join table "ProductTag"
            modelBuilder.Entity<Product>()
                .HasMany(p => p.Tags)
                .WithMany(t => t.Products)
                // Tells EFCore, create a table named ProductTag with
}
}

```

```

    TagId and ProductId as columns,
        //but don't bother me with an explicit class.
        .UsingEntity<Dictionary<string, object>>(
            "ProductTag",
            pt => pt.HasOne<Tag>
        ).WithMany().HasForeignKey("TagId"),
            pt => pt.HasOne<Product>
        ).WithMany().HasForeignKey("ProductId"));
        // Seed Tags (one record per line)
        modelBuilder.Entity<Tag>().HasData(
            new Tag { TagId = 1, Name = "electronics" },
            new Tag { TagId = 2, Name = "gaming" },
            new Tag { TagId = 3, Name = "office" },
            new Tag { TagId = 4, Name = "accessories" },
            new Tag { TagId = 5, Name = "home" }
        );
        // Seed Products (one record per line)
        modelBuilder.Entity<Product>().HasData(
            new Product { ProductId = 1, SKU = "GAM12345", Name =
"Gaming Laptop", Price = 1500.00m, Stock = 10, CategoryId = 1,
Description = "High performance gaming laptop.", Discount = 10,
ManufacturingDate = new DateTime(2023, 1, 1), ExpiryDate = new
DateTime(2024, 1, 1) },
            new Product { ProductId = 2, SKU = "OFF12345", Name =
"Office Desktop", Price = 800.00m, Stock = 20, CategoryId = 1,
Description = "Efficient desktop for office work.", Discount = 5,
ManufacturingDate = new DateTime(2023, 1, 1), ExpiryDate = new
DateTime(2024, 1, 1) },
            new Product { ProductId = 3, SKU = "SMA12345", Name =
"Smartphone", Price = 700.00m, Stock = 50, CategoryId = 2, Description =
"Latest model smartphone.", Discount = 0, ManufacturingDate = new
DateTime(2023, 1, 1), ExpiryDate = new DateTime(2024, 1, 1) },
            new Product { ProductId = 4, SKU = "WIR12345", Name =
"Wireless Mouse", Price = 50.00m, Stock = 100, CategoryId = 3,
Description = "Ergonomic wireless mouse.", Discount = 15,
ManufacturingDate = new DateTime(2023, 1, 1), ExpiryDate = new
DateTime(2024, 1, 1) },
            new Product { ProductId = 5, SKU = "MEC12345", Name =

```

```

    "Mechanical Keyboard", Price = 120.00m, Stock = 75, CategoryId = 3,
    Description = "RGB mechanical keyboard.", Discount = 20,
    ManufacturingDate = new DateTime(2023, 1, 1), ExpiryDate = new
    DateTime(2024, 1, 1) },
        new Product { ProductId = 6, SKU = "4KMON12", Name = "4K
Monitor", Price = 400.00m, Stock = 30, CategoryId = 4, Description =
"Ultra HD 4K monitor.", Discount = 5, ManufacturingDate = new
DateTime(2023, 1, 1), ExpiryDate = new DateTime(2024, 1, 1) },
        new Product { ProductId = 7, SKU = "GAMCHAIR", Name =
"Gaming Chair", Price = 300.00m, Stock = 15, CategoryId = 4, Description
= "Ergonomic gaming chair.", Discount = 10, ManufacturingDate = new
DateTime(2023, 1, 1), ExpiryDate = new DateTime(2024, 1, 1) },
        new Product { ProductId = 8, SKU = "BLU12345", Name =
"Bluetooth Speaker", Price = 150.00m, Stock = 40, CategoryId = 5,
Description = "Portable Bluetooth speaker.", Discount = 0,
ManufacturingDate = new DateTime(2023, 1, 1), ExpiryDate = new
DateTime(2024, 1, 1) },
        new Product { ProductId = 9, SKU = "SMAW1234", Name =
"Smartwatch", Price = 250.00m, Stock = 25, CategoryId = 2, Description =
"Feature-packed smartwatch.", Discount = 5, ManufacturingDate = new
DateTime(2023, 1, 1), ExpiryDate = new DateTime(2024, 1, 1) },
        new Product { ProductId = 10, SKU = "HOMECAM1", Name =
"Home Security Camera", Price = 100.00m, Stock = 60, CategoryId = 5,
Description = "HD home security camera.", Discount = 10,
ManufacturingDate = new DateTime(2023, 1, 1), ExpiryDate = new
DateTime(2024, 1, 1) }
    );
    // Seed join table for ProductTag (each record provided in
the same line)
    modelBuilder.Entity("ProductTag").HasData(
        new { ProductId = 1, TagId = 1 },
        new { ProductId = 1, TagId = 2 },
        new { ProductId = 2, TagId = 1 },
        new { ProductId = 2, TagId = 3 },
        new { ProductId = 3, TagId = 1 },
        new { ProductId = 4, TagId = 4 },
        new { ProductId = 5, TagId = 4 },
        new { ProductId = 5, TagId = 3 },

```

```

        new { ProductId = 6, TagId = 1 },
        new { ProductId = 6, TagId = 3 },
        new { ProductId = 7, TagId = 2 },
        new { ProductId = 7, TagId = 3 },
        new { ProductId = 8, TagId = 1 },
        new { ProductId = 8, TagId = 4 },
        new { ProductId = 9, TagId = 1 },
        new { ProductId = 9, TagId = 4 },
        new { ProductId = 10, TagId = 1 },
        new { ProductId = 10, TagId = 5 }
    );
}
}

}

```

Why Use Dictionary<string, object> while Configuring the Joining table?

We use Dictionary<string, object> here to define a shadow entity for the join table without creating an explicit CLR class for it. By specifying Dictionary<string, object>, we indicate that the join table (named “ProductTag”) doesn’t have its own dedicated entity class. It’s just a table with the foreign keys (ProductId and TagId).

- **No Explicit Entity Needed:** It allows us to configure the join table without writing a separate ProductTag class.
- **Shadow Properties:** The keys (e.g., “ProductId”, “TagId”) become shadow properties that EF manages internally.

Creating DTOs

Create a folder named DTOs in the Project root directory. A DTO (Data Transfer Object) transfers data between the client and server. It often contains a subset of the entity’s properties or additional properties for validation that should not be persisted directly. In our example, we will define:

- **ProductCreateDTO** for creation,
- **ProductUpdateDTO** for updating, and

- **ProductResponseDTO** for sending data back to clients.
- **ProductBaseDTO** for common properties and methods

ProductBaseDTO

```
namespace Fluent_API_Validation.DTOs
{
    public class ProductBaseDTO
    {
        public string SKU { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public int Stock { get; set; }
        public int CategoryId { get; set; }
        public string? Description { get; set; }
        public decimal Discount { get; set; }
        public DateTime ManufacturingDate { get; set; }
        public DateTime ExpiryDate { get; set; }
        public List<string>? Tags { get; set; }
    }
}
```

ProductCreateDTO

```
namespace Fluent_API_Validation.DTOs
{
    // ProductCreateDTO inherits all common properties from
    ProductBaseDTO

    public class ProductCreateDTO : ProductBaseDTO
    {
    }
}
```

ProductUpdateDTO

```
namespace Fluent_API_Validation.DTOs
{
```

```

// ProductUpdateDTO inherits all common properties from
ProductBaseDTO

// and adds the ProductId for identifying the product to update
public class ProductUpdateDTO : ProductBaseDTO
{
    public int ProductId { get; set; }
}

```

ProductResponseDTO

```

namespace Fluent_API_Validation.DTOs
{
    public class ProductResponseDTO
    {
        public int ProductId { get; set; }
        public string SKU { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public int Stock { get; set; }
        public int CategoryId { get; set; }
        public string? Description { get; set; }
        public decimal Discount { get; set; }
        public DateTime ManufacturingDate { get; set; }
        public DateTime ExpiryDate { get; set; }
        public List<string>? Tags { get; set; }
    }
}

```

Add Fluent Validation for the Product Model:

First, create a new folder named `Validators` in the project root directory. This folder contains classes inherited from `AbstractValidator<T>` to define validation rules separate from the model classes.

ProductBaseDTOValidator

```

using Fluent_API_Validation.Data;
using Fluent_API_Validation.DTOs;

```

```

using FluentValidation;
using Microsoft.EntityFrameworkCore;

namespace Fluent_API_Validation.Validators
{
    /// <summary>
    /// Base validator for ProductBaseDTO containing all common
    validation rules.
    /// Supports async validation for database checks.
    /// </summary>
    public class ProductBaseDTOValidator<T> : AbstractValidator<T> where
T : ProductBaseDTO
    {
        protected readonly ECommerceDbContext _context;

        public ProductBaseDTOValidator(ECommerceDbContext context)
        {
            _context = context;

            // SKU validation: must not be empty, match regex, and be
unique
            RuleFor(p => p.SKU)
                .NotEmpty().WithMessage("SKU is required.")
                .Matches("^[A-Z0-9]{8}$").WithMessage("SKU must be 8
characters long and contain only uppercase letters and digits.")
                .MustAsync(BeUniqueSKUAsync).WithMessage("SKU must be
unique.");

            // Name validation: must not be empty, proper length, and be
unique
            RuleFor(p => p.Name)
                .NotEmpty().WithMessage("Product name is required.")
                .Length(3, 50).WithMessage("Product name must be between
3 and 50 characters.")
                .MustAsync(BeUniqueNameAsync).WithMessage("Product name
must be unique.");

            // Price validation: must be greater than 0 and conform to
        }
    }
}

```

```

precision and scale
    RuleFor(p => p.Price)
        .GreaterThan(0).WithMessage("Price must be greater than
0.")
        .PrecisionScale(8, 2, true).WithMessage("Price must have
at most 8 digits in total and 2 decimals.");

    // Stock validation: must be zero or positive
    RuleFor(p => p.Stock)
        .GreaterThanOrEqualTo(0).WithMessage("Stock cannot be
negative.");

    // CategoryId validation: must be a positive number
    RuleFor(p => p.CategoryId)
        .GreaterThan(0).WithMessage("Category ID is required.");

    // Description validation: if provided, must not exceed 500
characters
    RuleFor(p => p.Description)
        .MaximumLength(500).WithMessage("Description cannot
exceed 500 characters.")
        .When(p => !string.IsNullOrEmpty(p.Description));

    // Discount validation: must be between 0 and 100
    RuleFor(p => p.Discount)
        .InclusiveBetween(0, 100).WithMessage("Discount must be
between 0 and 100.");

    // Manufacturing date validation: must not be a future date
    RuleFor(p => p.ManufacturingDate)

.LessThanOrEqualTo(DateTime.Now).WithMessage("Manufacturing date cannot
be in the future.");

    // Expiry date validation: must be later than the
manufacturing date
    RuleFor(p => p.ExpiryDate)
        .GreaterThan(p =>

```

```

p.ManufacturingDate).WithMessage("Expiry date must be after the
manufacturing date.");

        // Tags validation: each tag must not be empty and not
        exceed 20 characters
        RuleForEach(p => p.Tags).ChildRules(tag =>
        {
            tag.RuleFor(t => t)
                .NotEmpty().WithMessage("Tag cannot be empty.")
                .MaximumLength(20).WithMessage("Tag cannot exceed 20
characters.");
        });
    }

    // Checks asynchronously that the SKU is unique in the database.
    // Override in derived classes to exclude current product during
    updates.
    protected virtual async Task<bool> BeUniqueSKUAsync(T dto,
string sku, CancellationToken cancellationToken)
    {
        return !await _context.Products
            .AsNoTracking()
            .AnyAsync(p => p.SKU == sku, cancellationToken);
    }

    // Checks asynchronously that the product name is unique in the
    database.
    // Override in derived classes to exclude current product during
    updates.
    protected virtual async Task<bool> BeUniqueNameAsync(T dto,
string name, CancellationToken cancellationToken)
    {
        return !await _context.Products
            .AsNoTracking()
            .AnyAsync(p => p.Name == name, cancellationToken);
    }
}

```

```
    }  
}
```

ProductCreateDTOValidator

The following class is inherited from the `AbstractValidator<T>` class where `T` is the model being validated and, in our example, it is the `ProductCreateDTO`.

```
using Fluent_API_Validation.Data;  
using Fluent_API_Validation.DTOs;  
using FluentValidation;  
  
namespace Fluent_API_Validation.Validators  
{  
    // Validator for ProductCreateDTO, inherits all common rules from  
    ProductBaseDTOValidator.  
    public class ProductCreateDTOValidator :  
ProductBaseDTOValidator<ProductCreateDTO>  
    {  
        public ProductCreateDTOValidator(ECommerceDbContext context) :  
base(context)  
        {  
        }  
    }  
}
```

ProductUpdateDTOValidator

```
using Fluent_API_Validation.Data;  
using Fluent_API_Validation.DTOs;  
using FluentValidation;  
using Microsoft.EntityFrameworkCore;  
  
namespace Fluent_API_Validation.Validators  
{  
    // Validator for ProductUpdateDTO, inherits all common rules from  
    ProductBaseDTOValidator.
```

```

    // Includes async validation to check if ProductId exists in
database.

    public class ProductUpdateDTOValidator :
ProductBaseDTOValidator<ProductUpdateDTO>
    {
        public ProductUpdateDTOValidator(ECommerceDbContext context) :
base(context)
        {
            // ProductId validation: must be greater than 0 and exist in
database
            RuleFor(p => p.ProductId)
                .GreaterThan(0).WithMessage("ProductId must be greater
than 0.")
                .MustAsync(ProductExistsAsync).WithMessage("Product does
not exist.");
        }

        // Checks asynchronously that the ProductId exists in the
database.

        private async Task<bool> ProductExistsAsync(int productId,
CancellationToken cancellationToken)
        {
            return await _context.Products
                .AsNoTracking()
                .AnyAsync(p => p.ProductId == productId,
cancellationToken);
        }

        // Override to exclude the current product when checking SKU
uniqueness during updates.

        protected override async Task<bool>
BeUniqueSKUAsync(ProductUpdateDTO dto, string sku, CancellationToken
cancellationToken)
        {
            return !await _context.Products
                .AsNoTracking()

```

```

        .AnyAsync(p => p.SKU == sku && p.ProductId !=  

dto.ProductId, cancellationToken);  

    }  
  

        // Override to exclude the current product when checking Name  

uniqueness during updates.  

    protected override async Task<bool>  

BeUniqueNameAsync(ProductUpdateDTO dto, string name, CancellationToken  

cancellationToken)  

    {  

        return !await _context.Products  

            .AsNoTracking()  

            .AnyAsync(p => p.Name == name && p.ProductId !=  

dto.ProductId, cancellationToken);  

    }  

}
}

```

Create the Products Controller

The ProductsController handles HTTP requests related to product operations (such as creating, retrieving, and updating products). It is the intermediary between client requests and the underlying business logic and data access layer. The controller demonstrates:

- **Create** and **Update** endpoints that use DTOs and FluentValidation.
- A **GET endpoint** that filters products based on a comma-separated list of tags.
- Proper mapping between DTOs and domain models with tag processing.

```

using Fluent_API_Validation.Data;
using Fluent_API_Validation.DTOs;
using Fluent_API_Validation.Model;
using FluentValidation;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

```

```

namespace Fluent_API_Validation.Controllers
{
    // The ProductsController handles all product-related API endpoints.
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        // The database context used to interact with the underlying
        database.

        private readonly ECommerceDbContext _context;
        private readonly IValidator<ProductCreateDTO> _createValidator;
        private readonly IValidator<ProductUpdateDTO> _updateValidator;
        // Constructor injection of the database context and validators.
        public ProductsController(
            ECommerceDbContext context,
            IValidator<ProductCreateDTO> createValidator,
            IValidator<ProductUpdateDTO> updateValidator)
        {
            _context = context;
            _createValidator = createValidator;
            _updateValidator = updateValidator;
        }

        // GET: api/products?tags=tag1,tag2
        // Retrieves all products, optionally filtered by any matching
        tags.

        [HttpGet]
        public async Task<ActionResult<IEnumerable<ProductResponseDTO>>>
GetProducts([FromQuery] string? tags)
        {
            // Build the query and include the related Tags collection
            for each product.

            IQueryable<Product> query = _context.Products
                .AsNoTracking()
                .Include(p => p.Tags);
            if (!string.IsNullOrEmpty(tags))
            {

```

```

        // Split the comma-separated tags string into a list.
        // Trim spaces and convert each tag to lower case for
        case-insensitive comparison.
        var tagList = tags.Split(',',
StringSplitOptions.RemoveEmptyEntries)
                    .Select(t => t.Trim().ToLower())
                    .ToList();
        // Filter products that contain ANY of the specified
tags.
        // If a product has at least one matching tag, it is
        included in the result.
        query = query.Where(p => p.Tags.Any(t =>
tagList.Contains(t.Name.ToLower())));
    }
    // Execute the query and retrieve the list of products from
the database.
    var products = await query.ToListAsync();
    // Map each Product entity to a ProductResponseDTO.
    var result = products.Select(p => new ProductResponseDTO
{
    ProductId = p.ProductId,
    SKU = p.SKU,
    Name = p.Name,
    Price = p.Price,
    Stock = p.Stock,
    CategoryId = p.CategoryId,
    Description = p.Description,
    Discount = p.Discount,
    ManufacturingDate = p.ManufacturingDate,
    ExpiryDate = p.ExpiryDate,
    // Map the Tags collection to a list of tag names.
    Tags = p.Tags.Select(t => t.Name).ToList()
}).ToList();
    // Return the filtered list of products with an HTTP 200 OK
status.
    return Ok(result);
}

```

```

// GET: api/products/{id}
// Retrieves a single product by its ID.
[HttpGet("{id}")]
public async Task<ActionResult<ProductResponseDTO>>
GetProduct(int id)
{
    // Retrieve the product with the given ID, including its
Tags.
    // AsNoTracking() is used here since no update is needed
(improves performance).
    var product = await _context.Products
        .AsNoTracking()
        .Include(p => p.Tags)
        .FirstOrDefaultAsync(p =>
p.ProductId == id);

    // If the product is not found, return a 404 Not Found
response.
    if (product == null)
        return NotFound();
    // Map the Product entity to a ProductResponseDTO.
    var response = new ProductResponseDTO
    {
        ProductId = product.ProductId,
        SKU = product.SKU,
        Name = product.Name,
        Price = product.Price,
        Stock = product.Stock,
        CategoryId = product.CategoryId,
        Description = product.Description,
        Discount = product.Discount,
        ManufacturingDate = product.ManufacturingDate,
        ExpiryDate = product.ExpiryDate,
        Tags = product.Tags.Select(t => t.Name).ToList()
    };
    // Return the product details with an HTTP 200 OK status.
    return Ok(response);
}

```

```

// POST: api/products
// Creates a new product based on the data provided in
ProductCreateDTO.
[HttpPost]
public async Task<ActionResult<ProductResponseDTO>>
CreateProduct([FromBody] ProductCreateDTO productDto)
{
    // Validate asynchronously using FluentValidation
    var validationResult = await
_createValidator.ValidateAsync(productDto);

    if (!validationResult.IsValid)
    {
        var errors = validationResult.Errors.Select(e => new
        {
            Field = e.PropertyName,
            Error = e.ErrorMessage
        });
        return BadRequest(new { Errors = errors });
    }

    // Map the incoming DTO to a new Product entity.
    var product = new Product
    {
        SKU = productDto.SKU,
        Name = productDto.Name,
        Price = productDto.Price,
        Stock = productDto.Stock,
        CategoryId = productDto.CategoryId,
        Description = productDto.Description,
        Discount = productDto.Discount,
        ManufacturingDate = productDto.ManufacturingDate,
        ExpiryDate = productDto.ExpiryDate
    };
    // Process Tags: For each tag provided in the DTO, check if
it exists.
    // If the tag exists, use the existing Tag; otherwise,

```

```

create a new Tag.

    if (productDto.Tags != null && productDto.Tags.Any())
    {
        foreach (var tagName in productDto.Tags)
        {
            // Normalize the tag by trimming whitespace and
            converting to lower case.

            var normalizedTagName = tagName.Trim().ToLower();
            var existingTag = await
_context.Tags.FirstOrDefaultAsync(t => t.Name.ToLower() ==
normalizedTagName);

            if (existingTag != null)
                product.Tags.Add(existingTag); // Associate the
existing tag with the product.

            else
                product.Tags.Add(new Tag { Name =
normalizedTagName }); // Create a new tag and associate it.

        }
    }
    // Add the new product to the database context.
    _context.Products.Add(product);
    // Save changes to persist the new product (and associated
tags) to the database.

    await _context.SaveChangesAsync();
    // Map the newly created Product entity to a
ProductResponseDTO.

    var response = new ProductResponseDTO
    {
        ProductId = product.ProductId,
        SKU = product.SKU,
        Name = product.Name,
        Price = product.Price,
        Stock = product.Stock,
        CategoryId = product.CategoryId,
        Description = product.Description,
        Discount = product.Discount,
        ManufacturingDate = product.ManufacturingDate,
        ExpiryDate = product.ExpiryDate,
    }
}

```

```

        Tags = product.Tags.Select(t => t.Name).ToList()
    };
    // Return the created product with an HTTP 200 OK (or 201
Created) status.
    return Ok(response);
}

// PUT: api/products/{id}
// Updates an existing product based on the data provided in
ProductUpdateDTO.
[HttpPut("{id}")]
public async Task<ActionResult<ProductResponseDTO>>
UpdateProduct(int id, [FromBody] ProductUpdateDTO productDto)
{
    // Verify that the ID provided in the URL matches the ID in
the request body.
    if (id != productDto.ProductId)
        return BadRequest(new { error = "Product ID in URL and
body do not match." });
    // Validate asynchronously using FluentValidation
    var validationResult = await
_updateValidator.ValidateAsync(productDto);

    if (!validationResult.IsValid)
    {
        var errors = validationResult.Errors.Select(e => new
{
    Field = e.PropertyName,
    Error = e.ErrorMessage
});
        return BadRequest(new { Errors = errors });
    }
    // Retrieve the existing product from the database,
including its associated Tags.
    var product = await _context.Products
        .Include(p => p.Tags)
        .FirstOrDefaultAsync(p =>
p.ProductId == id);

```

```

    // If the product does not exist, return a 404 Not Found
    response.

        if (product == null)
            return NotFound();
        // Update the product properties with the new values from
        the DTO.

            product.SKU = productDto.SKU;
            product.Name = productDto.Name;
            product.Price = productDto.Price;
            product.Stock = productDto.Stock;
            product.CategoryId = productDto.CategoryId;
            product.Description = productDto.Description;
            product.Discount = productDto.Discount;
            product.ManufacturingDate = productDto.ManufacturingDate;
            product.ExpiryDate = productDto.ExpiryDate;
            // Clear the existing Tags associated with the product.
            product.Tags.Clear();
            // Process Tags: For each tag provided in the DTO, normalize
            and add it.

                if (productDto.Tags != null && productDto.Tags.Any())
                {
                    foreach (var tagName in productDto.Tags)
                    {
                        // Normalize the tag by trimming whitespace and
                        converting to lower case.

                            var normalizedTagName = tagName.Trim().ToLower();
                            var existingTag = await
                            _context.Tags.FirstOrDefaultAsync(t => t.Name.ToLower() ==
                            normalizedTagName);
                            if (existingTag != null)
                                product.Tags.Add(existingTag); // Associate the
                                existing tag with the product.
                            else
                                product.Tags.Add(new Tag { Name =
                                normalizedTagName }); // Create and add a new tag.
                    }
                }

```

```

        // Mark the product entity as updated in the database
        context.
            _context.Products.Update(product);
        // Save changes to persist the updated product (and tag
        associations) to the database.
        await _context.SaveChangesAsync();
        // Map the updated product entity to a ProductResponseDTO.
        var response = new ProductResponseDTO
        {
            ProductId = product.ProductId,
            SKU = product.SKU,
            Name = product.Name,
            Price = product.Price,
            Stock = product.Stock,
            CategoryId = product.CategoryId,
            Description = product.Description,
            Discount = product.Discount,
            ManufacturingDate = product.ManufacturingDate,
            ExpiryDate = product.ExpiryDate,
            Tags = product.Tags.Select(t => t.Name).ToList()
        };
        // Return the updated product with an HTTP 200 OK status.
        return Ok(response);
    }
}

```

Configure the Database Connection String in the appsettings.json file

To connect our DbContext to a database, we need to add a connection string in our `appsettings.json` file. So, please modify the `appsettings.json` file as follows:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
}
```

```
},
"AllowedHosts": "*",
"ConnectionStrings": {
    "DefaultConnection": "Data Source=localhost;Initial Catalog=fluent_validation_db;Integrated Security=True;Pooling=False;Encrypt=True;Trust Server Certificate=True"
}
}
```

Register FluentValidation and DbContext Services in Program Class

Next, we need to register the DbContext and FluentValidation services to the Dependency Injection container.

```
using Fluent_API_Validation.Data;
using Fluent_API_Validation.DTOs;
using Fluent_API_Validation.Validators;
using FluentValidation;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add controllers with JSON options
builder.Services.AddControllers()
    .AddJsonOptions(options =>
{
    options.JsonSerializerOptions.PropertyNamingPolicy = null;
});

// Configure OpenAPI
builder.Services.AddOpenApi();

// Register the ECommerceDbContext with dependency injection
builder.Services.AddDbContext<ECommerceDbContext>(options =>

    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")),
        sqlServerOptions => sqlServerOptions.EnableRetryOnFailure(
```

```

        maxRetryCount: 5,
        maxRetryDelay: TimeSpan.FromSeconds(30),
        errorNumbersToAdd: null)));
}

// Register validators for specific DTOs
builder.Services.AddScoped<IValidator<ProductCreateDTO>,
ProductCreateDTOValidator>();
builder.Services.AddScoped<IValidator<ProductUpdateDTO>,
ProductUpdateDTOValidator>();

var app = builder.Build();

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();

```

Generate and Apply Database Migration:

```

# 1. Make model changes
# 2. Create migration
dotnet ef migrations add AddNewFeature

# 3. Review migration files
# 4. Apply to database
dotnet ef database update

# 5. If needed, rollback
dotnet ef database update PreviousMigration

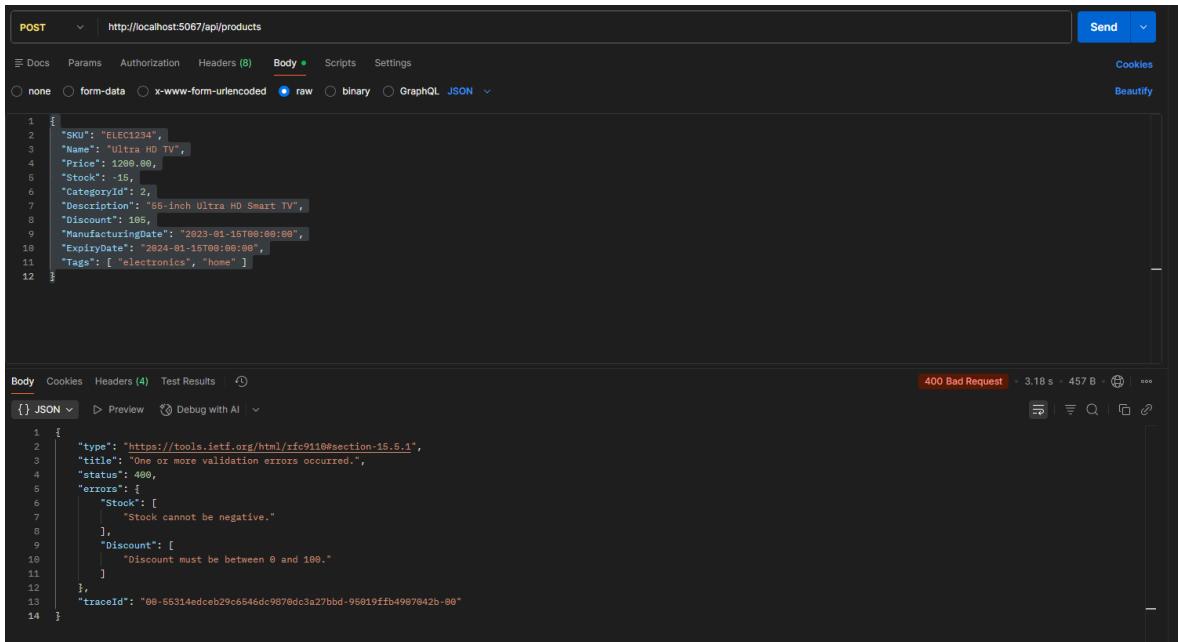
```

Test the POST Endpoint (Create Product)

Create a New Product Method: POST URL: <https://localhost:5001/api/products> Headers: Content-Type: application/json Body: Use a sample JSON like the one below (adjust values as needed)

```
{  
    "SKU": "ELEC1234",  
    "Name": "Ultra HD TV",  
    "Price": 1200.00,  
    "Stock": -15,  
    "CategoryId": 2,  
    "Description": "55-inch Ultra HD Smart TV",  
    "Discount": 105,  
    "ManufacturingDate": "2023-01-15T00:00:00",  
    "ExpiryDate": "2024-01-15T00:00:00",  
    "Tags": [ "electronics", "home" ]  
}
```

You should get the following error message.



image_306.png

Differences Between Fluent API and Data Annotation Validation in ASP.NET Core Web API:

Feature	Fluent API Validation	Data Annotation Validation
Flexibility	Highly flexible with support for complex logic.	Limited to predefined attributes.
Separation of Concerns	Validation logic is separate from models.	Validation logic is tightly coupled with models.
Conditional Validation	Easily supports dynamic or runtime conditions.	Challenging to implement.
Reusability	Rules are reusable across multiple models.	Not reusable; tied to individual properties.
Customization	Allows custom messages and rules with ease.	Limited customization options.
Ease of Use	Requires additional setup and understanding.	Simple and built into the framework.

image_307.png

When to Use Which Approach

- Use **Data Annotations** for simple scenarios where validation rules are straightforward. There is no need to reuse them across different models. They are easy to implement and understand.
- Use **Fluent API Validation** for complex, reusable, or dynamic validation scenarios where separating the validation logic from the model helps maintain cleaner code. It's also ideal when the same validation logic is needed across multiple models.



Note: The FluentValidation.AspNetCore package is no longer maintained. Microsoft recommends to use the core FluentValidation package and adopting a manual validation approach.

Filters – ASP.NET Core Web API

Start typing here...

Security – ASP.NET Core Web API

Start typing here...

JWT – ASP.NET Core Web API

Start typing here...

API Versioning in ASP.NET Core

Start typing here...

Repository Pattern in ASP.NET Core Web API

Start typing here...

Unit Testing – ASP.NET Core Web API

Start typing here...

Minimal API – ASP.NET Core

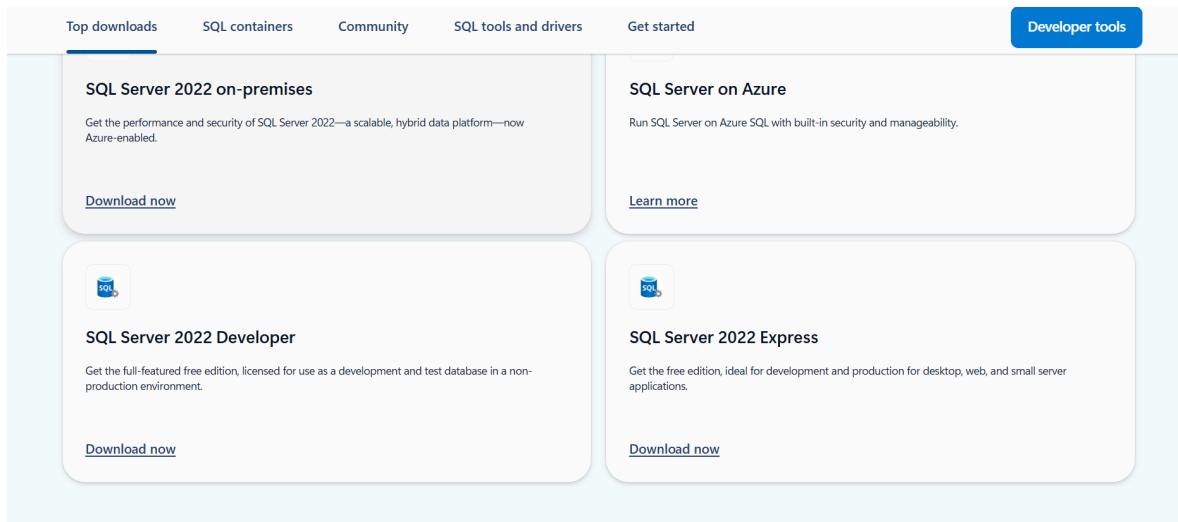
Start typing here...

Microservices in ASP.NET Core

Start typing here...

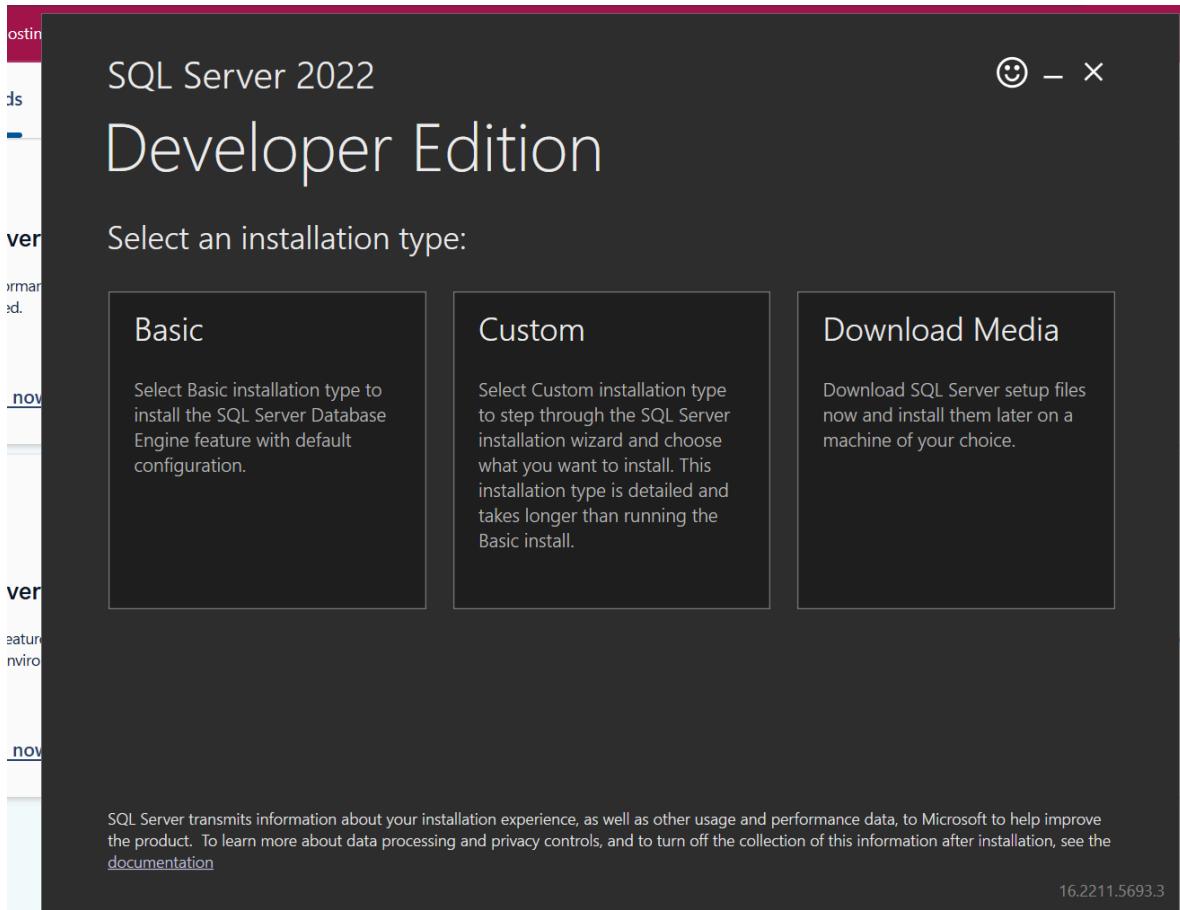
SQL Server

Visit <https://www.microsoft.com/en-us/sql-server/sql-server-downloads>
and download



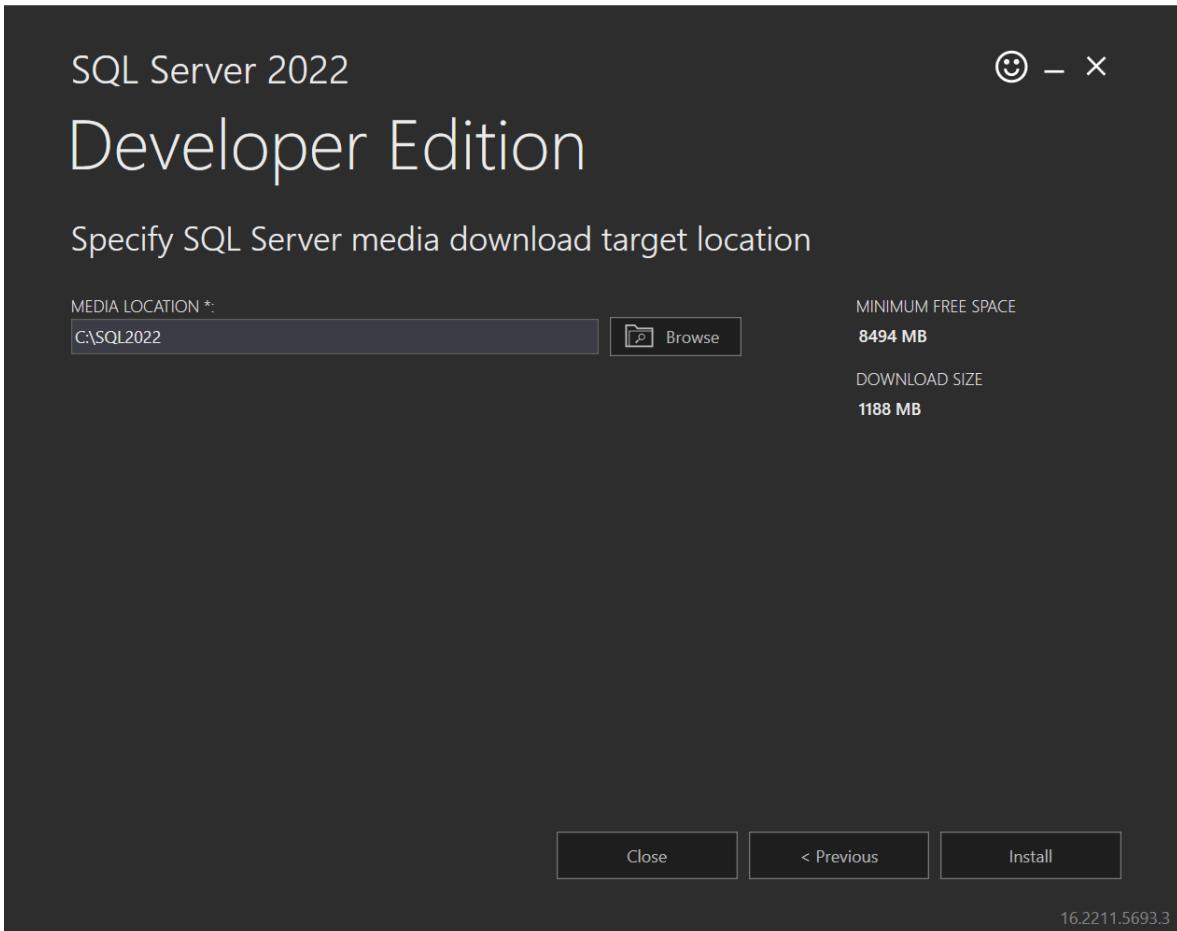
image_71.png

click custom



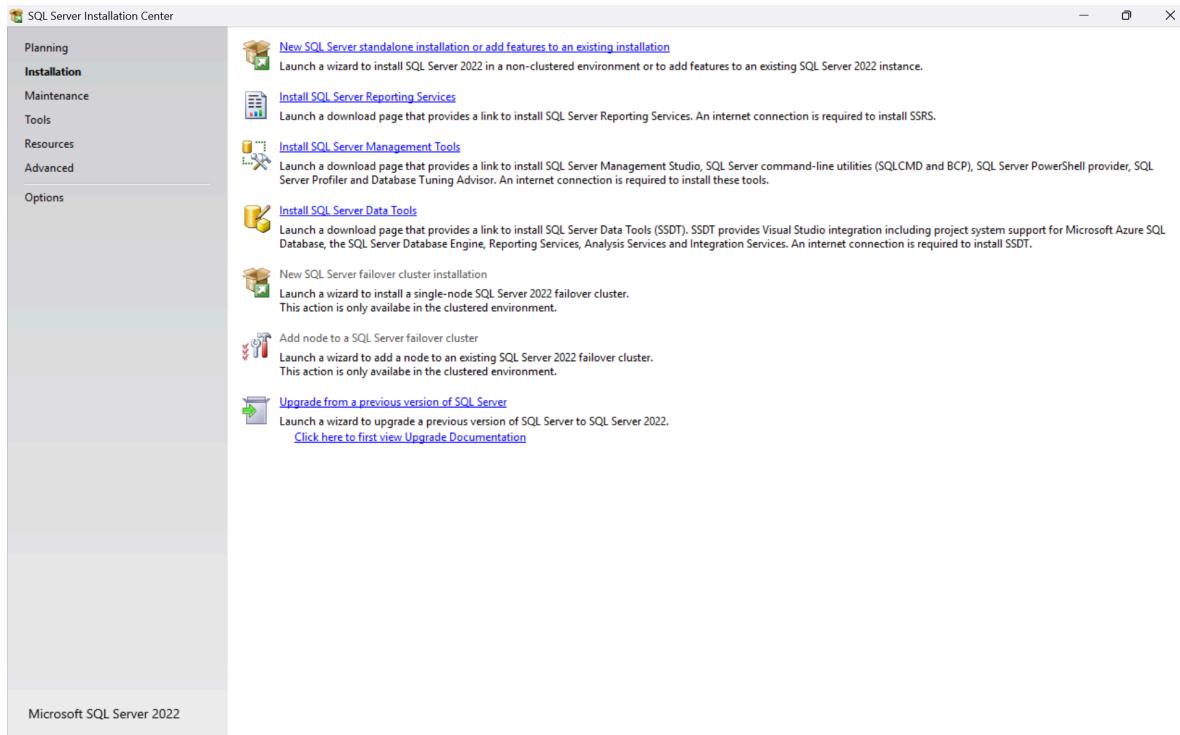
image_72.png

click install



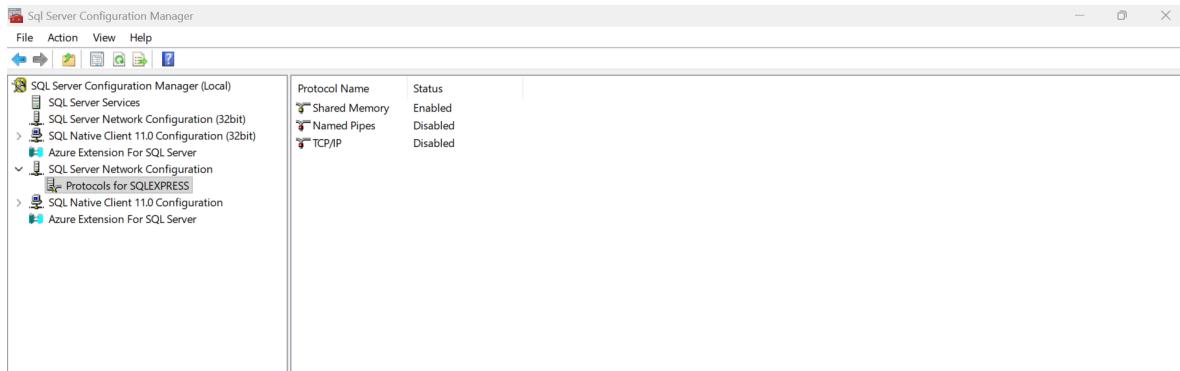
image_73.png

click new sql server standalone installation



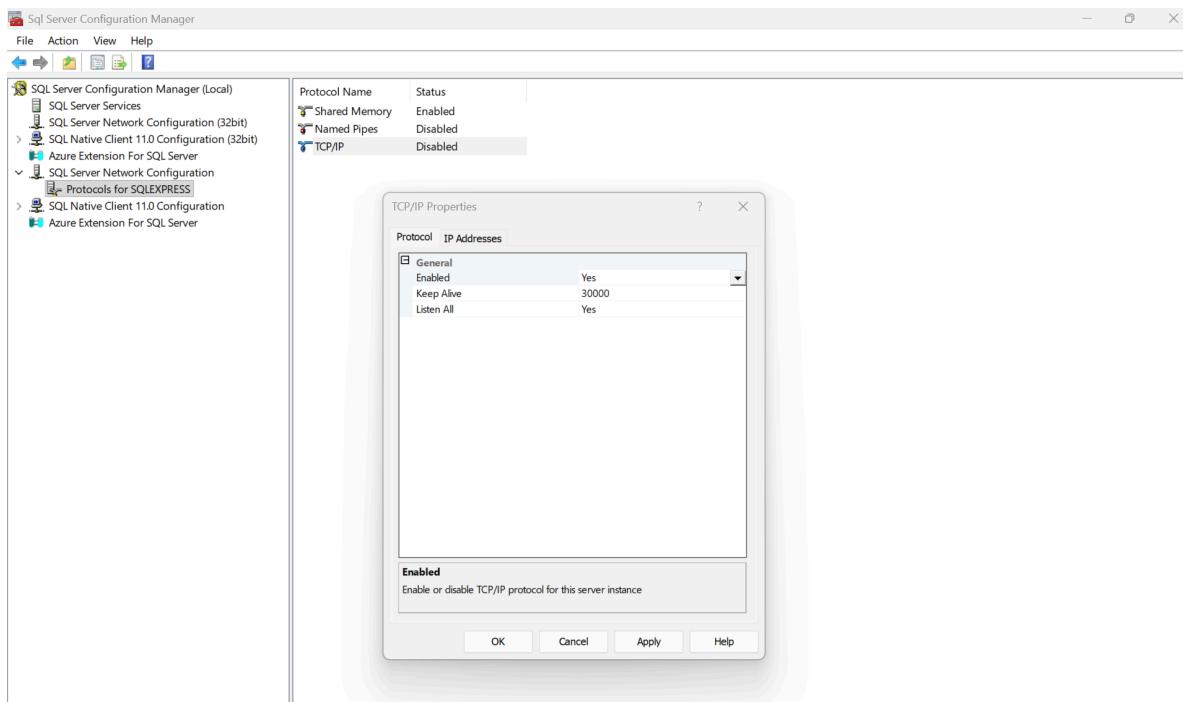
image_74.png

after installation open Sql server Configuration Manager and go to SQL Server Network Configuration, under protocols



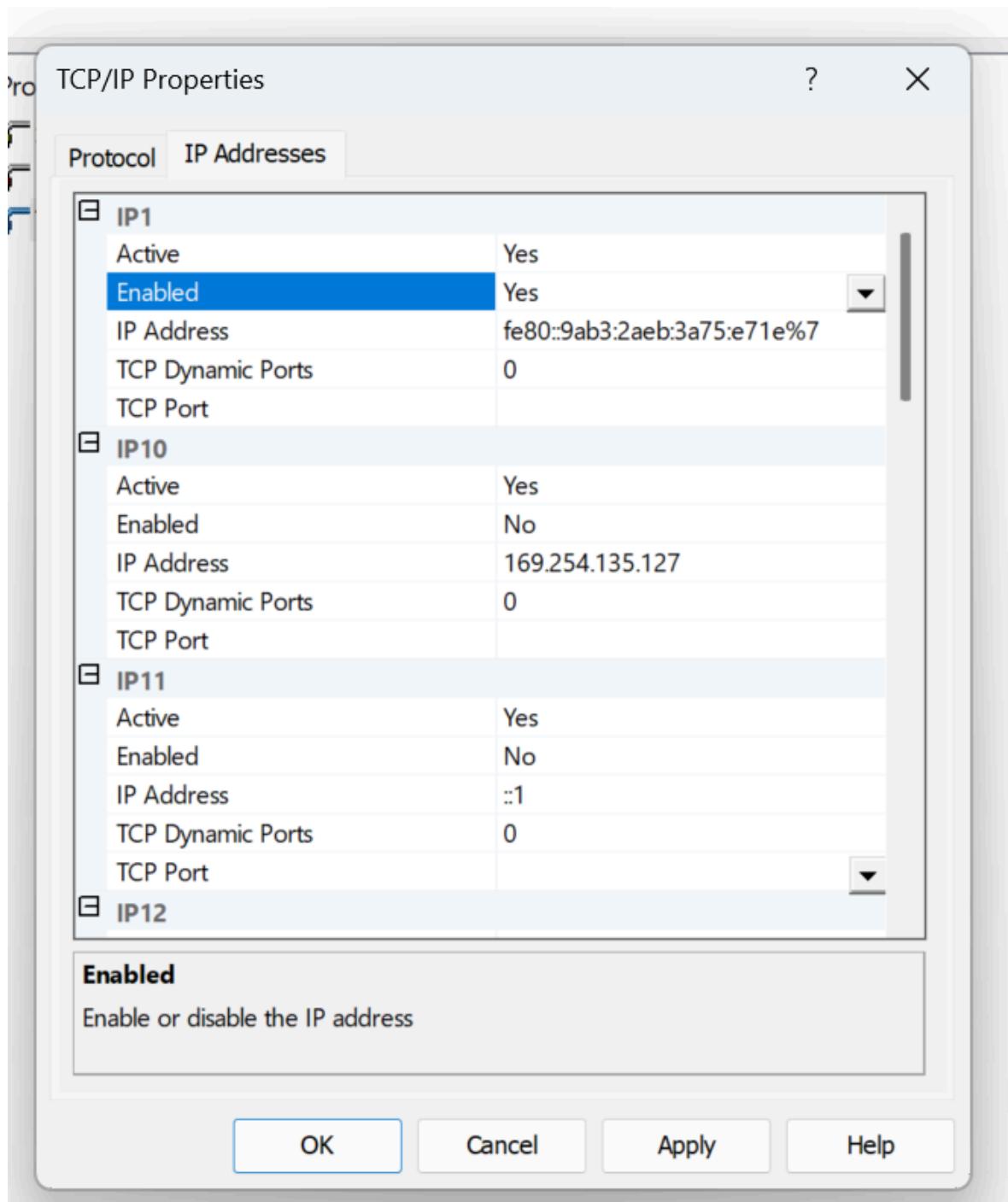
image_75.png

right click and go to properties to enable,



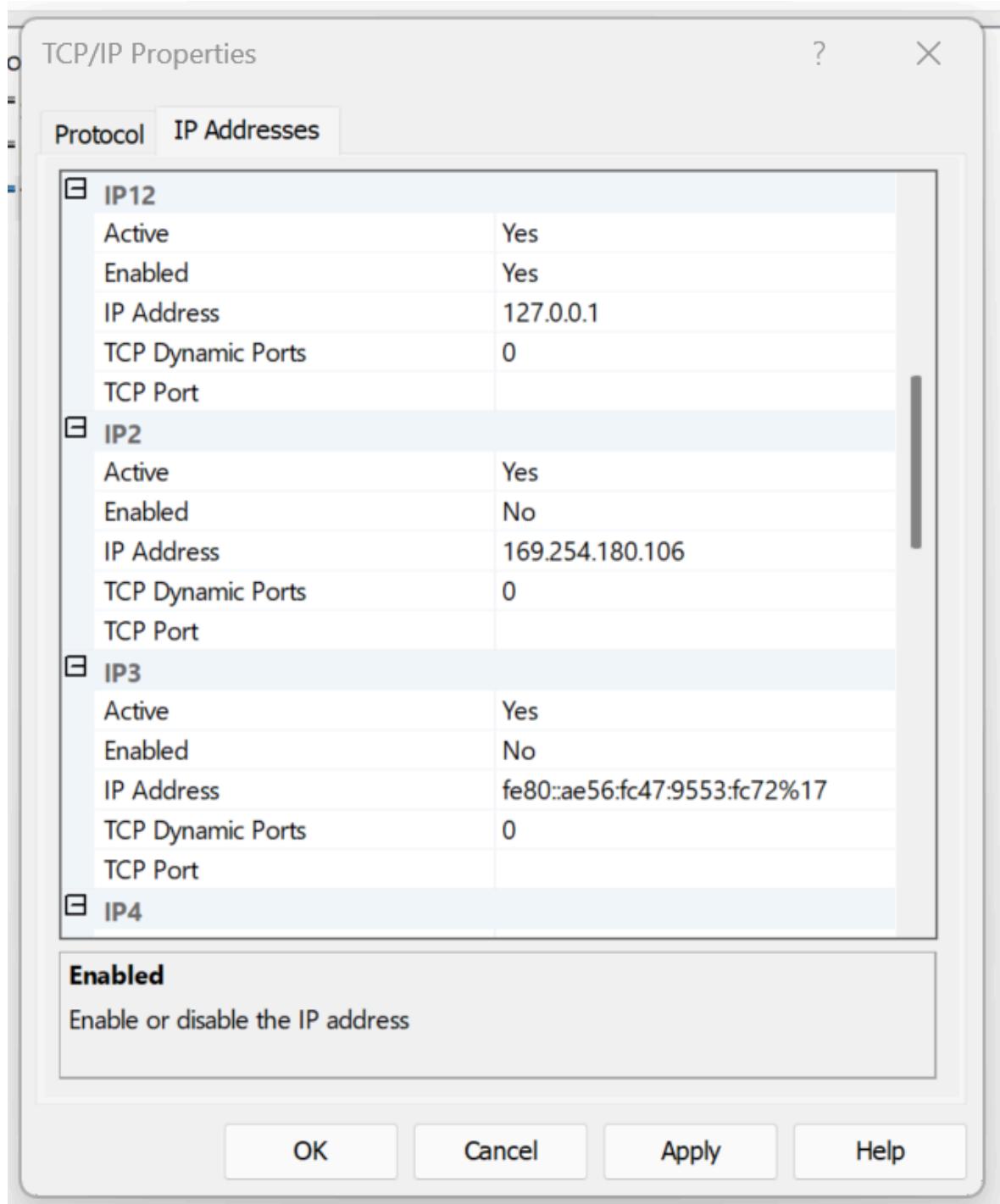
image_76.png

switch to Ip address tab and enable ip1



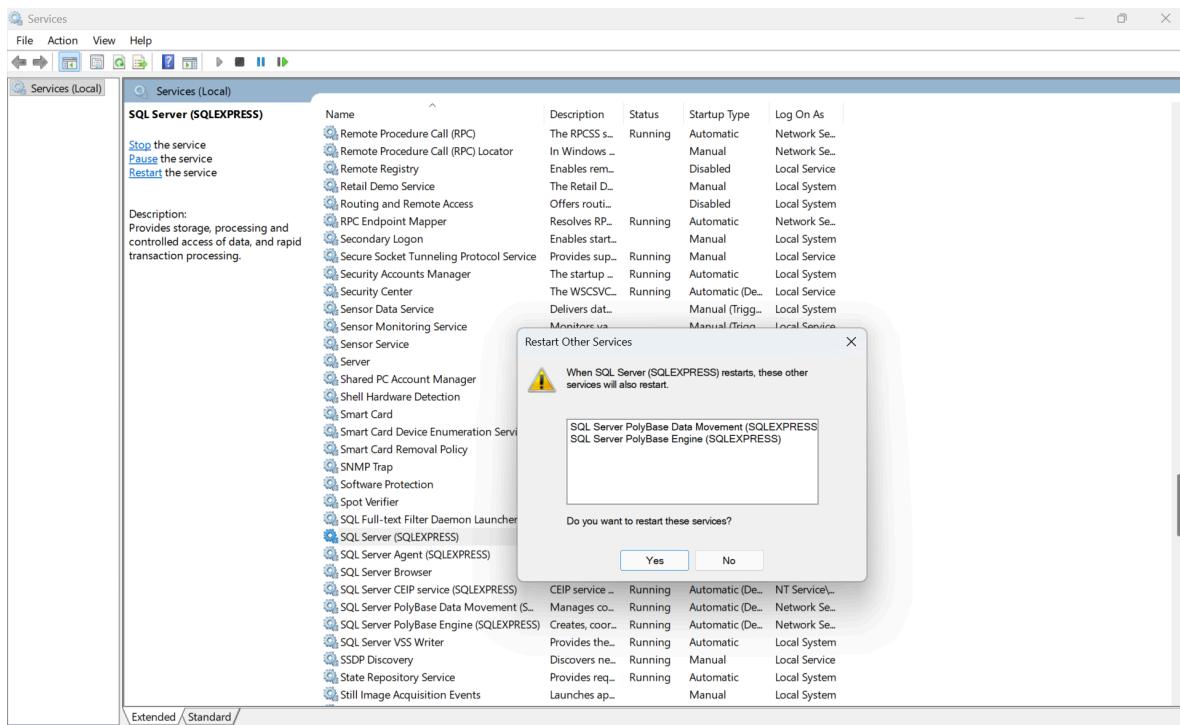
image_77.png

and ip12 (127.0.0.1 == localhost)



image_78.png

click window key and search for services, then scroll down to a service called SQL Server, we need to restart it to make sure the changes we made take effect. Right click and press restart



image_79.png