

Introduction to .NET

What is .NET?

Ans: It is a product of Microsoft launched in the year 2002, which can be used for building various kinds of Applications like: Web, Mobile, Desktop, Micro services, Cloud, Machine Learning, Game Development and IoT (Internet of Things).

How to develop all the above applications by using .NET?

Ans: To develop the above applications, .NET provides with a set of Programming Languages, Technologies & Servers using which we can build any kind of Application.

What are the Programming Languages, .NET provides to us?

Ans: In .NET there are 30+ programming languages available for a developer to build applications and programmers have a chance of choosing any 1 language from the list.

Features of .NET: there are 2 important features in .NET, those are:

1. Language Independent
2. Platform Independent

1. Language Independent: .NET is a collection of programming Languages i.e.; it provides us multiple languages for building our applications and developers can choose any 1 language from the list to build their applications. At the time of launching .NET in 2002, Microsoft has given 30+ Languages like C#, VB.NET, Fortran.NET, Python.NET (Iron Python), Cobol.NET, VCPP.NET, Pascal.NET, J#.NET, etc. Most of these languages are extension to some existing languages, like:

C, CPP	=>	C#
Cobol	=>	Cobol.NET
Pascal	=>	Pascal.NET
Fortran	=>	Fortran.NET
Visual Basic	=>	VB.NET
Visual CPP	=>	VCPP.NET
Python	=>	Python.NET (Iron Python)
Java	=>	J#.NET
	=>	F#
	=>	ML.NET

Note: As of today, we don't have all these 30+ languages in usage, what we have is only 5 languages in usage like C#, VB.NET, F#.NET, Iron Python and ML.NET, and the most popular of all these languages is "C#". Because .NET is a collection of languages, programmers always have a choice to choose a language based on his previous experience or interest to build their applications, for example:

Task: Write a program for printing from 1 to 100 by using a for loop.

C# Source Code => Compiled by using C# Compiler => CIL Code

```
static void Main()
{
    for (int i = 1; i <= 100; i++)
```

```
{  
    Console.WriteLine(i);  
}  
}
```

VB Source Code => Compiled by using VB Compiler => CIL Code

```
Shared Sub Main()  
    For I As Integer = 1 To 100 Step 1  
        Console.WriteLine(i)  
    Next i  
End Sub
```

F# Source Code => Compiled by using F# Compiler => CIL Code

```
let main() =  
    for i = 1 to 100 do  
        printfn "%i" i  
main()
```

The output code that is generated after compilation of a program that is implemented by using a .NET Language is called CIL (Common Intermediate Language) Code or MSIL (Microsoft Intermediate Language).

COBOL, Pascal, FORTRAN, and C Languages are Procedural Programming Languages and the drawback in this approach is they don't provide security and re-usability of code. To overcome the drawbacks of Procedural Programming Language's in early 80's we are provided with a new approach known as **Object Oriented Programming** which provides security and re-usability.

All Object-Oriented Programming Languages have an important feature that is "**Code Re-usability**" i.e., the code we write in 1 program can be consumed from another program, for example:

C++ Source Code => Compiled by using C++ Compiler => Generates Object Code => Which can be consumed from another C++ Program.

Java Source Code => Compiled by using Java Compiler => Generates Byte Code => Which can be consumed from another Java Program.

C# Source Code => Compiled by using C# Compiler => Generates CIL Code => Which can be consumed from any .NET Language Program.

F# Source Code => Compiled by using F# Compiler => Generates CIL Code => Which can be consumed from any .NET Language Program.

VB Source Code => Compiled by using VB Compiler => Generates CIL Code => Which can be consumed from any .NET Language Program.

Note: Re-usability in CPP and Java Languages is only with-in that language whereas the same re-usability in .NET Languages is across all languages of .NET, and this is what we call as Language Independent.

If any 2 languages want to communicate or interoperate with each other they need to cross 2 hurdles:

1. There should not be any mismatch in compiled code.
2. There should not be any mismatch in data types.

Lang1 (int is 2 bytes) => Object Code

Lang2 (int is 4 bytes) => Object Code

Note: In .NET Languages we will not face compiled code mismatch because all languages are generating CIL or MSIL Code only after the compilation. They don't face data type mis-match problem also because all languages of .NET adopt a rule known as "Uniform Data Type Structure" i.e., similar types will always be same in size irrespective of their names.

2. Platform Independent: it is an approach of executing an application that is developed on 1 platform, in other platforms.

What is a Platform?

Ans: A platform is an environment under which an application executes, and it is a combination of 2 things, those are Micro-Processor and Operating System.



Note: up to 1995, application that are developed by using programming languages that are present in the market (E.g., C, CPP, VB, Cobol, Pascal, Fortran, VCPP) are all platform dependent i.e., if we develop any application by using any of these languages on 1 platform, we can't run them on other platforms. For example, we can't install MS Office on Linux or Mac OS, so it is a platform dependent application.

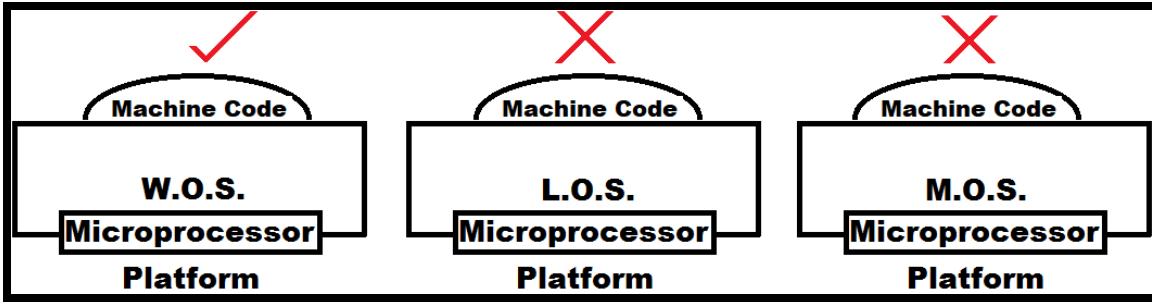
Why older programming languages are platform dependent?

Ans: Applications that are developed by using programming languages that are present in the market before 1995 are all platform dependent, because in all these languages when we compile the Source Code, they will generate Machine Code based on the O.S. where they are compiled, so Machine Code that is generated for 1 O.S is not understandable to other OS's.

Application developed by using C++ language on Windows OS:

Source Code => Compiled by C++ Compiler => Machine Code

Machine Code means operating system understandable code and this code has an advantage and dis-advantage. Advantage is to run the Machine Code we don't require to install CPP Software on client machines whereas dis-advantage is the above Machine Code runs only on Windows but not on any other OS.



Note: any application which directly sits on the top of OS is always a platform dependent application.

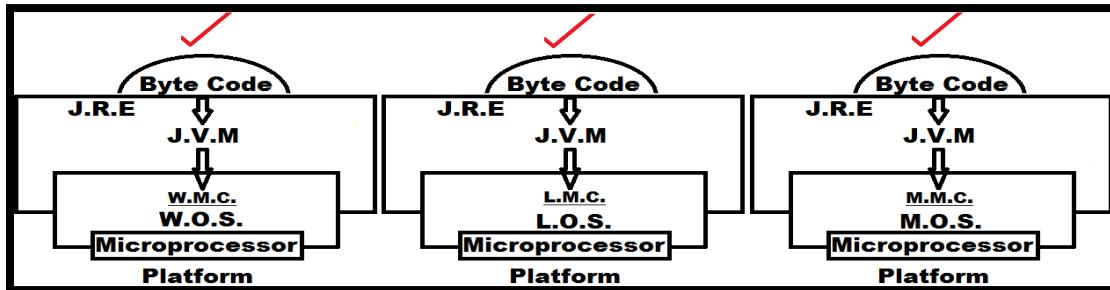
What is Platform Independent?

Ans: Applications that are developed by using Java and .NET Languages are Platform Independent i.e.,these applications once developed on a Platform can run on any other Platform (i.e., write once and run anywhere).

Application developed by using Java language on Windows OS:

Source Code => Compiled by Java Compiler => Byte Code

Byte Code is not OS understandable, so OS is not at all responsible to execute this code. We can run this Byte Code, we need to install a software provided by Java known as JRE (Java Runtime Environment) and if this software is installed on the Client's Computer we can run the Byte Code where ever we want, because inside of the JRE there is a component called as JVM(Java Virtual Machine) and that JVM contains a compiler called "JIT(Just In Time) Compiler" which will convert Byte Code into Machine Code based on the OS where it was executing.



Note: JRE software is platform dependent i.e., we are provided with this JRE separately for each OS and this makes the Byte Code platform independent.

Windows Machine installed with Windows JRE:

Byte Code => JVM => Converts into Windows Machine Code

Linux Machine installed with Linux JRE:

Byte Code => JVM => Converts into Linux Machine Code

Mac Machine installed with Mac JRE:

Byte Code => JVM => Converts into Mac Machine Code

Solaris Machine installed with Solaris JRE:

Byte Code => JVM => Converts into Solaris Machine Code

We can download JRE from the below sites:

<https://www.java.com/en/download/manual.jsp>

<https://www.oracle.com/in/java/technologies/javase-jre8-downloads.html>

.NET: Microsoft launched .NET in the year 2002.

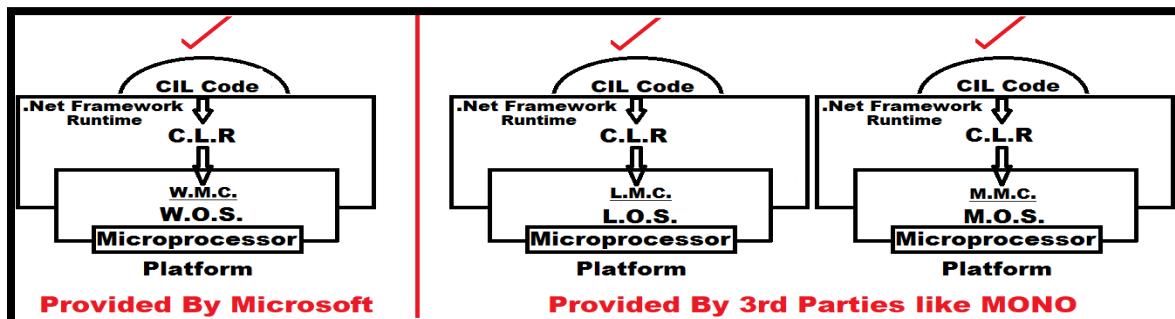
Application developed by using .NET languages on Windows OS:

Source Code => Compiled by a Language Compiler => CIL Code

Note: as said earlier, .NET is a collection of programming languages so with whatever .NET Language we develop the application and compile the source code by using an appropriate language compiler, the outcome will be "CIL" (Common Intermediate Language) code only.

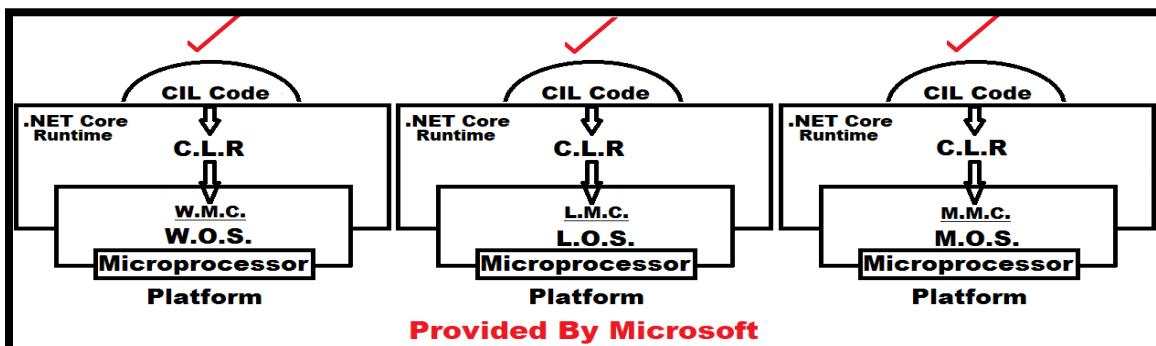
We will install CIL Code on Client Machines and to run that code we need to install software known as ".NET Runtime" and inside of this Runtime there will be a component called CLR (Common Language Runtime) which will convert CIL Code into Native Machine Code.

In the year 2002 when Microsoft launched .NET in the market, they provided their first Runtime for Windows O.S. only but not for any other O.S.'s, but they made the specifications to develop the Runtime as open, so 3rd party companies came forward and developed the Runtime's for other O.S.'s also and the name of that runtime is ".NET Framework". The first version of .NET Framework is 1.0 and the last version is 4.8.

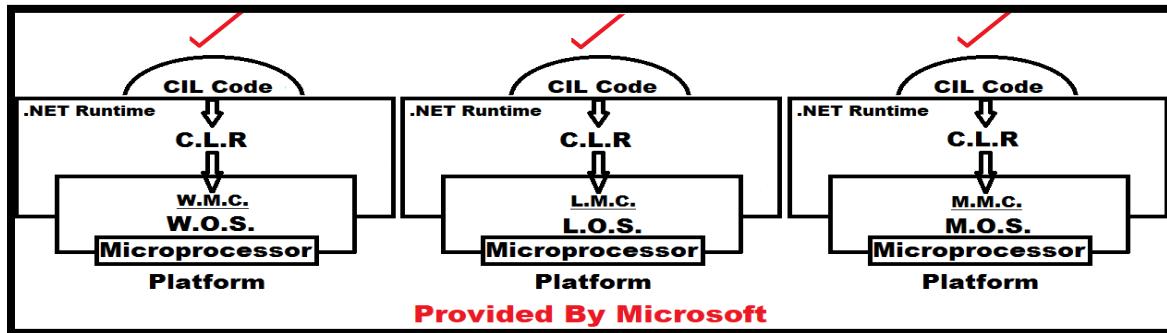


Note: with .NET Framework Runtime there is a criticism on .NET that it is not fully Platform Independent because Microsoft has given it only for Windows.

In the year 2016 Microsoft launched a new Runtime into the market with the name ".NET Core" and this runtime is provided for Windows, Linux, and Mac machines also. The first version of .NET Core is 1.0 and the last version is 3.1.



On November 10, 2020, Microsoft launched a new Runtime into the market by combining .NET Framework & .NET Core as **1 .NET** which starts from version 5.0 and the latest is 6.0 launched on November 2021.

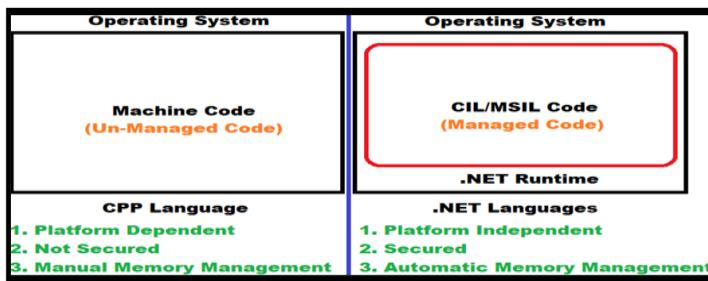


Note: the new .NET is nothing but .NET Core only but with-out again calling .NET Core and .NET Framework they made the name simple as just “.NET”.

What is a .NET Runtime?

Ans: It's software which must be installed on Client's Machine if at all we want to run .NET Application's on that Machine which sits on top of the O.S. and executes the CIL Code by masking the functionalities of an OS.

In case of platform dependent languages like Cobol, C, CPP, Visual Basic, etc. Compiled Code i.e., Machine Code runs directly under OS., whereas in case of .NET Languages, CIL Code will run under the .NET Runtime.



Note: Application's that directly run under the O.S. are known as Un-Managed App's whereas App's that run under .NET Runtime are known as Managed App's.

Applications that run under these runtime's are provided with the following features:

- Platform Independent or Portable
- Secured
- Automatic Memory Management

The development of .NET started with the development of this Runtime in late 90's originally under the name **“NGWS (Next Generation Windows Systems)”** and to develop this software first they prepared a specification known as **“CLI Specifications”**, where CLI stands from Common Language Infrastructure. This CLI Specification describes 4 aspects in it, those are:

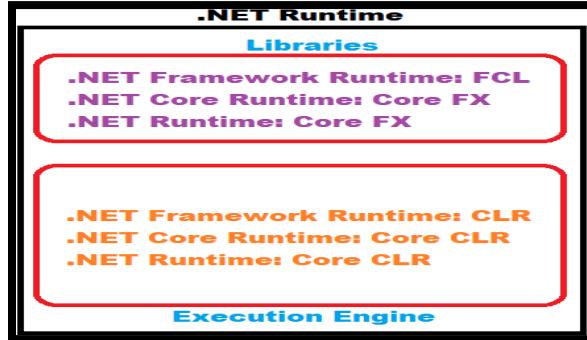
1. **CLS (Common Language Specification):** it's a set of base rules all Languages of .NET must follow to interoperate with each other, most importantly after compilation of source code all those languages need to generate the same type of output code known as CIL Code, so that when any 2 languages want to interoperate with each other, then compiled code mismatch will not come into picture.

2. **CTS (Common Type System):** According to this all languages of .NET should follow a standard regarding the Data Types i.e., “Uniform Data Type Structure” which means similar types must always be same in size irrespective of their names.

Note: Because of these CLS and CTS only, all language of .NET can interoperate or communicate with each other.

3. **Metadata:** Information about program structure is language-independent, so that it can be referenced between languages and tools, making it easy to work with code written in a language the developer are not aware.
4. **VES (Virtual Execution System):** this is nothing but CLR or Common Language Runtime.

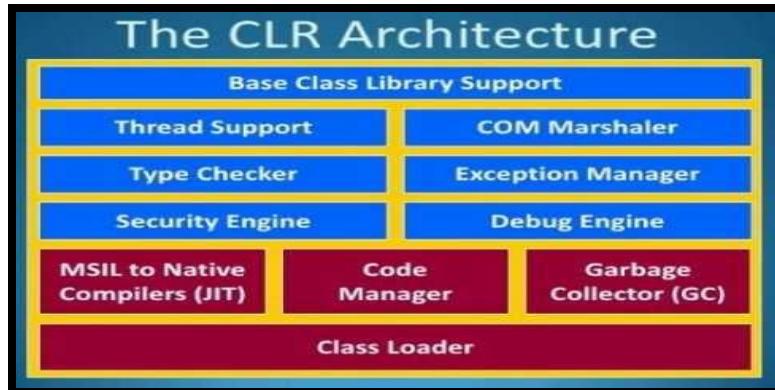
The runtime software internally contains 2 main components in it, those are the “Libraries” and an “Execution Engine” as following:



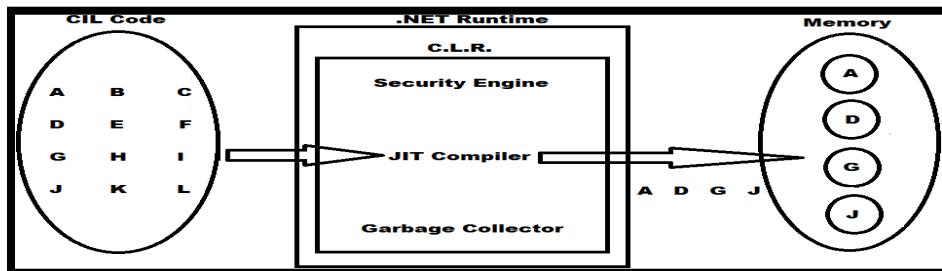
Libraries: A library is a set of re-usable functionalities, and every programming language has built-in libraries to it like **Header Files** in C & CPP Languages and **Packages** in Java Language same as that **.NET Languages** are also provided with built-in libraries and we call them “**FCL (Framework Class Libraries)**” in **.NET Framework** and “**CORE FX**” in **.NET Core** and **.NET**.

Execution Engine: as discussed earlier, .NET Applications will not run under the OS, but they will be running under the Runtime and in this Runtime, we have an Execution Engine responsible for the execution of Applications and we call this as “**CLR (Common Language Runtime)**” in **.NET Framework** and “**CORE CLR**” in **.NET Core** and **.NET**.

CLR and **Core CLR** are known as execution engine of **.NET Runtime**, where all **.NET Application** run under the supervision of this **CLR** and it internally it contains various components in it to manage various actions, like:



1. **Security Engine:** this is responsible for the security of our applications, i.e., it will take care that applications don't directly interact with the OS, as well as OS don't directly interact the application.
2. **JIT Compiler:** this is the compiler which is responsible for converting CIL Code into Machine Code based on the platform where we are executing the application adopting a process known as “Conversion gradually during the program’s execution”.



3. **Garbage Collector:** it is responsible for “Automatic Memory Management” where “Memory Management” is a process of allocation and de-allocation of memory that is required for a program to execute, and this is of 2 types:
 - Manual or Explicit
 - Automatic or Implicit

Manual or Explicit means, in this case programmers are responsible for allocation and de-allocation of the memory explicitly. Automatic or Implicit means, here programmers are not at all responsible for allocation and de-allocation of the memory and on behalf of the programmers Garbage Collector will take the responsibility for memory management.

What is Application Software?

Ans: Application software is commonly defined as any program or number of programs designed for end-users. In that sense, any end user program can be called an “application.” People often use the term “application software” to talk about bundles or groups of individual software applications, using a different term, “application program” to refer to individual applications. Examples of application software include items like Notepad, WordPad, Microsoft Word, Microsoft Excel, or any of the Web Browsers used to navigate the Internet, etc.

Another way to understand application software is, in a very basic sense, every program that you use on your computer is a piece of application software. The operating system, on the other hand, is system software. Historically, the application was generally born as computers evolved into systems where you could run a particular codebase on a given operating system. Even social media platforms have come to resemble applications, especially on our mobile phone devices, where individual applications are given the nickname “apps.” So, while the term

“application software” can be used broadly, it’s an important term in describing the rise of sophisticated computing environments.

How to develop Application software?

Ans: There are two basic camps of software development: Applications Development and Systems Development. Applications Development is focused on creating programs that meet the users' needs. These can range from mobile phone apps, video games, enterprise-level accounting software. Systems Development is focused on creating and maintaining operating systems and to do this we need to familiar with some Programming Language. Thousands of different programming languages have been created, and more are being created every year. Many programming languages are written in an imperative form (i.e., as a sequence of operations to perform) while other languages use the declarative form (i.e., the desired result is specified, not how to achieve it).

What is a Programming Language?

Ans: A programming language is a formal language comprising a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms. Most programming languages consist of instructions for computers. Since the early 1800s, programs have been used to direct the behavior of machines such as Jacquard looms, music boxes and player pianos.

“A computer programming language is a language used to write computer programs, which involves a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on.”

Anyone can come up with ideas, but a developer will be able to turn those ideas into something concrete. Even if you only want to work on the design aspects of software, you should have some familiarity with coding and be able to create basic prototypes. There are a huge variety of programming languages that we can learn. Very early computers, such as Colossus is thus regarded as the world's first programmable, electronic, digital computer, although it was programmed by switches and plugs and not by a stored program.

Slightly later, programs could be written in machine language, where the programmer writes each instruction in a numeric form the hardware can execute directly. For example, the instruction to add the value in two memory location might consist of 3 numbers: an “opcode” that selects the “add” operation, and two memory locations. The programs, in decimal or binary form, were read in from punched cards, paper tape, and magnetic tape or toggled in on switches on the front panel of the computer. Machine languages were later termed first-generation programming languages (1GL).

The next step was development of so-called second-generation programming languages (2GL) or assembly languages, which were still closely tied to the instruction set architecture of the specific computer. These served to make the program much more human-readable and relieved the programmer of tedious and error-prone address calculations.

The first high-level programming languages, or third-generation programming languages (3GL), were written in the 1950s. John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer. Unlike machine code, Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.

At the University of Manchester, Alick Glennie developed Autocode in the early 1950s. As a programming language, it used a compiler to automatically convert the language into machine code. The first code and compiler were developed in 1952 for the Mark 1 computer at the University of Manchester and is the first compiled high-level programming language.

In 1954, FORTRAN was invented at IBM by John Backus. It was the first widely used high-level general purpose programming language to have a functional implementation, as opposed to just a design on paper. It is still a popular language for high-performance computing and is used for programs that benchmark and rank the world's fastest supercomputers.

Another early programming language was devised by Grace Hopper in the US, called FLOW-MATIC. It was developed for the UNIVAC I at Remington Rand during the period from 1955 until 1959. Hopper found that business data processing customers were uncomfortable with mathematical notation, and in early 1955, she and her team wrote a specification for an English programming language and implemented a prototype. The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959. FLOW-MATIC was a major influence in the design of COBOL.

COBOL an acronym for "common business-oriented language" is a compiled English-like computer programming language designed for business use. It is imperative, procedural and, since 2002, object-oriented. COBOL is primarily used in business, finance, and administrative systems for companies and governments. COBOL is still widely used in applications deployed on mainframe computers, such as large-scale batch and transaction processing jobs. But due to its declining popularity and the retirement of experienced COBOL programmers, programs are being migrated to new platforms, rewritten in modern languages. Most programming in COBOL is now purely to maintain existing applications.

Pascal is an imperative and procedural programming language, designed by Niklaus Wirth as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. It is named in honor of the French mathematician, philosopher, and physicist Blaise Pascal. Pascal enabled defining complex data types and building dynamic and recursive data structures such as lists, trees, and graphs. Pascal has strong typing on all objects, which means that one type of data cannot be converted or interpreted as another without explicit conversions.

C is a general-purpose, imperative procedural computer programming language supporting structured programming, lexical variable scope, and recursion, with a static type system. By design, C provides constructs that map efficiently to typical machine instructions. It has found lasting use in applications previously coded in assembly language. Such applications include operating systems, various application software for computers that range from super computers to PLCs and embedded systems. A successor to the programming language B, C was originally developed at Bell Labs by Dennis Ritchie between 1972 and 1973 to construct utilities running on UNIX. It was applied to re-implementing the kernel of the UNIX operating system. During the 1980s, C gradually gained popularity. It has become one of the most widely used programming languages, with C compilers from various vendors available for most existing computer architectures and operating systems. C has been standardized by the ANSI since 1989 (ANSI C) and by the International Organization for Standardization (ISO).

C++ is a general-purpose programming language developed by Danish computer scientist Bjarne Stroustrup at Bell Labs since 1979 as an extension of the C programming language, or "C with Classes" as he wanted an efficient and flexible language like C that also provided high-level features for program organization. The

language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms. C++ has also been found useful in many contexts, with key strengths being software infrastructure and resource-constrained applications, including desktop applications, video games, servers (e.g., e-commerce, Web search, or SQL Servers), and performance-critical applications (e.g., telephone switches or space probes).

Objective-C is a general-purpose, object-oriented programming language that adds Smalltalk-style messaging to the C programming language. It was the main programming language supported by Apple for macOS, iOS, and their respective application programming interfaces (APIs). The language was originally developed in the early 1980s. It was later selected as the main language used by NeXT for its NeXTSTEP operating system, from which macOS and iOS are derived. Objective-C source code 'implementation' program files usually have .m filename extensions, while Objective-C 'header/interface' files have .h extensions, the same as C header files. Objective-C++ files are denoted with a .mm file extension.

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. Python is dynamically typed, and garbage collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented, and functional programming. Python was conceived in the late 1980s as a successor to the ABC language. Python 2.0 released in 2000 and Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible, i.e., Python 2 code does not run unmodified on Python 3. The Python 2 language was officially discontinued in 2020 (first planned for 2015) and now only Python 3.5.x and later are supported.

Java is a general-purpose programming language that is class-based and object-oriented, and designed to have as few implementation dependencies as possible. It is intended to let application developers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need of recompilation. Java applications are typically compiled to byte code that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. The syntax of Java is like C and C++. Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle) and released in 1995 as a core component of Sun Microsystems' Java platform.

C# (pronounced see sharp, like the musical note C#, but written with the number sign) is a general-purpose, multi-paradigm programming language encompassing strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed around 2000 by Microsoft as part of its .NET initiative and later approved as an international standard by ECMA in 2002 and ISO in 2003. C# was designed by Anders Hejlsberg, and its development team is currently led by "Mads Torgersen". The most recent version is 9.0, which was released on November 2020 alongside Visual Studio 2019.

What is .NET?

Ans: .NET is a free, cross-platform, open-source developer platform for building many different types of applications like Desktop, Web, Mobile, Games and IOT by using multiple languages, editors, and libraries.

What is a Platform?

Ans: It is the environment in which a piece of software is executed. A platform can also be called as the stage on which computer programs can run. Platform can refer to the type of processor (CPU) on which a given operating system runs, the type of operating system on a computer or the combination of the type of hardware and the type of operating system running on it. An example of a common platform is Microsoft Windows running on x86 architecture. Other well-known desktop computer platforms include Linux/Unix and macOS

What is Cross-platform?

Ans: In computing, cross-platform software (also multi-platform software or platform-independent software) is computer software that is implemented to run on multiple platforms. For example, a cross-platform application may run on Microsoft Windows, Linux, and macOS. Cross-platform programs may run on as many as all existing platforms, or on few platforms.

What is meant by developing applications using multiple languages?

Ans: .NET languages are programming languages that are used to produce libraries and programs that conform to the Common Language Infrastructure (CLI) specifications. Most of the CLI languages compile entirely to the Common Intermediate Language (CIL), an intermediate language that can be executed using the Common Language Runtime, implemented by .NET Framework, .NET Core, and Mono. As the program is being executed, the CIL code is just-in-time compiled to the machine code appropriate for the architecture on which the program is running. While there are currently over 30+ languages in .NET, but only a small number of them are widely used and supported by Microsoft. List of .NET languages include C#, F#, Visual Basic, C++, Iron Python, etc. and the most popular and widely used language as a developer choice is C#. Visit the following link to view the list of .NET Languages: https://microsoft.fandom.com/wiki/Microsoft_.NET_Languages

What is CLI (Common Language Infrastructure)?

Ans: The Common Language Infrastructure (CLI) is an open specification (technical standard) developed by Microsoft and standardized by ISO (International Organization for Standardization) and ECMA (European Computer Manufacturers Association) that describes about executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures. This implies it is platform independent. The .NET Framework, .NET Core and Mono are implementations of the CLI.

CLI specification describes the following four aspects:

1. The Common Language Specification (CLS):
2. The Common Type System (CTS):
3. The Metadata:
4. The Virtual Execution System (VES):

What is .NET Framework and .NET Core?

Ans: .NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications. There are various implementations of .NET, and each implementation allows .NET code to execute in different places - Linux, macOS, Windows, iOS, Android, and many more. Various implementations of the .NET include:

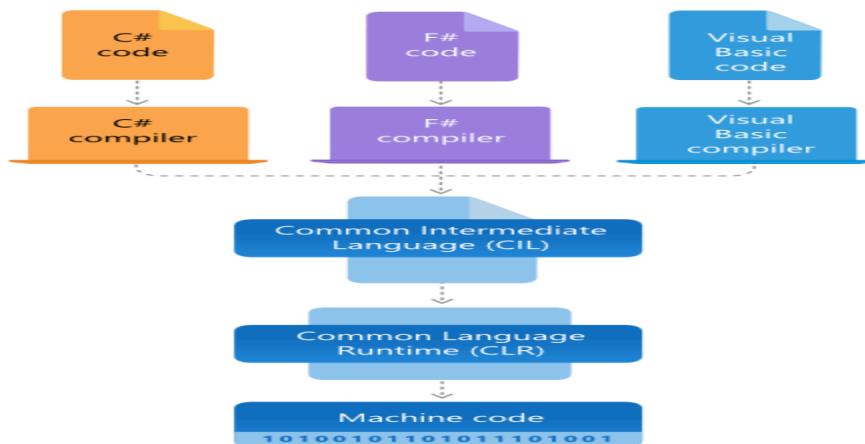
1. **.NET Framework:** it is the original implementation of .NET, and it supports running websites, services, desktop apps, and more on Windows.

2. **.NET Core:** it is a cross-platform implementation for running websites, services, and console apps on Windows, Linux, and macOS.
3. **Xamarin/Mono:** it is a .NET implementation for running apps on all the major mobile operating systems, including iOS and Android.

Architecture of .NET Framework: The two major components of .NET Framework are the .NET Framework Class Library and the Common Language Runtime.

1. The Class Library provides a set of APIs and types for common functionality. It provides types for strings, dates, numbers, etc. The Class Library includes APIs for reading and writing files, connecting to databases, drawing, and more.
2. The Common Language Runtime (CLR) is the heart of .NET Framework and the execution engine that handles running applications. It provides services like thread management, garbage collection, type-safety, exception handling, and more.

Architecture of .NET Framework CLR: .NET applications can be written in any .NET Language like C#, F#, or Visual Basic. Source Code we write by using some .NET Language is compiled into a language-agnostic Common Intermediate Language (CIL) and the compiled code is stored as assemblies (files with a ".dll" or ".exe" extension). When we run the applications, CLR takes the assemblies and uses a just-in-time compiler (JIT) to turn it into machine code that can execute on the specific architecture of the computer it is running on.



.NET Framework FAQ's

What is .NET Framework used for?

Ans: .NET Framework is used to create and run software applications. .NET apps can run on many operating systems, using different implementations of .NET. .NET Framework is used for running .NET apps on Windows.

Who uses .NET Framework?

Ans: Software developers and the users of their applications both use .NET Framework:

- Users need to install .NET Framework to run application built with the .NET Framework. In most cases, .NET Framework is already installed with Windows. If needed, you can download .NET Framework.
- Software developers use .NET Framework to build many different types of applications - websites, services, desktop apps, and more with Visual Studio. Visual Studio is an integrated development

environment (IDE) that provides development productivity tools and debugging capabilities. See the .NET customer showcase for examples of what people are building with .NET.

Why do I need .NET Framework?

Ans: You need .NET Framework installed to run applications on Windows that were created using .NET Framework. It is already included in many versions of Windows. You only need to download and install .NET Framework if prompted to do so.

How does .NET Framework work?

Ans: .NET Framework applications can be written in many languages like C#, F#, or Visual Basic and compiled to Common Intermediate Language (CIL). The Common Language Runtime (CLR) runs .NET applications on a given machine, converting the CIL to machine code. See Architecture of .NET Framework for more info.

What are the main components/features of .NET Framework?

Ans: The two major components of .NET Framework are the Common Language Runtime (CLR) and the .NET Framework Class Library. The CLR is the execution engine that handles running applications. The Class Library provides a set of APIs and types for common functionality.

How many versions do we have for .NET Framework?

Ans: There are multiple versions of .NET Framework, but each new version adds new features but retains features from previous versions. List of .NET Framework Versions:

.NET Framework 1.0	.NET Framework 1.1	.NET Framework 2.0	.NET Framework 3.0
.NET Framework 3.5	.NET Framework 4	.NET Framework 4.5	.NET Framework 4.5.1
.NET Framework 4.5.2	.NET Framework 4.6	.NET Framework 4.6.1	.NET Framework 4.6.2
.NET Framework 4.7	.NET Framework 4.7.1	.NET Framework 4.7.2	.NET Framework 4.8

Can you have multiple .NET Frameworks installed?

Ans: Some versions of .NET Framework are installed side-by-side, while others will upgrade an existing version (known as an in-place update). In-place updates occur when two .NET Framework versions share the same CLR version. For example, installing .NET Framework 4.8 on a machine with .NET Framework 4.7.2 and 3.5 installed will perform an in-place update of the 4.7.2 installation and leave 3.5 installed separately.

.NET Framework Version	CLR Version
.NET Framework 4.x	4.0
.NET Framework 2.x and 3.x	2.0
.NET Framework 1.1	1.1
.NET Framework 1.0	1.0

How much does .NET Framework cost?

Ans: .NET Framework is free, like the rest of the .NET platform. There are no fees or licensing costs, including for commercial use.

Which version of .NET Framework should I use?

Ans: In most cases, you should use the latest stable release and currently, that's .NET Framework 4.8. Applications that were created with any 4.x version of .NET Framework will run on .NET Framework 4.8. To run an application that was created for an earlier version (for example, .NET Framework 3.5), you should install that version.

What is the support policy for .NET Framework?

Ans: .NET Framework 4.8 is the latest version of .NET Framework and will continue to be distributed with future releases of Windows. If it is installed on a supported version of Windows, .NET Framework 4.8 will continue to also be supported.

Can customers continue using the .NET Framework and get support?

Ans: Yes. Many products both within and outside Microsoft rely on .NET Framework. The .NET Framework is a component of Windows and receives the same support as Windows version which it ships with or on which it is installed. .NET Framework 4.8 is the latest version of .NET Framework and will continue to be distributed with future releases of Windows. If it is installed on a supported version of Windows, .NET Framework 4.8 will continue to also be supported.

Architecture of .NET Core: The two main components of .NET Core are CoreCLR and CoreFX, respectively, which are comparable to the Common Language Runtime (CLR) and the Framework Class Library (FCL) of the .NET Framework's Common Language Infrastructure (CLI) implementation.

1. **CoreFX** is the foundational class libraries for .NET Core. It includes types for collections, file systems, console, JSON, XML, and many others.
2. **CoreCLR** is the .NET execution engine in .NET Core, performing functions such as garbage collection and compilation to machine code. As a CLI implementation of Virtual Execution System (VES), CoreCLR is a complete runtime and virtual machine for managed execution of .NET programs and includes a just-in-time compiler called RyuJIT.

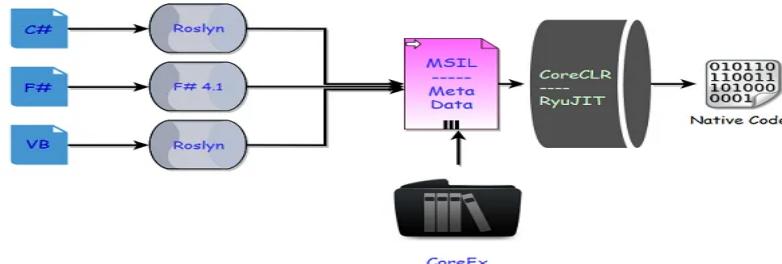
Note: .NET Core releases have a single product version, that is, there is no separate CLR version.

What is CoreFX?

Ans: CoreFX, also referred to as the Unified Base Class Library, consists of the basic and fundamental classes that form the core of the .Net Core platform. These set of libraries comprise the System.* (and to a limited extent Microsoft.*) namespaces. Majority of the .NET Core APIs are also available in the .NET Framework, so you can think of CoreFX as an extension of the .NET Framework Class Library.

What is CoreCLR?

Ans: CoreCLR is the .NET execution engine in .NET Core which is a complete runtime and virtual machine for managed execution of .NET programs and includes a just-in-time compiler called RyuJIT, performing functions such as garbage collection and compilation to machine code. CoreCLR is built from the same code base of the Framework CLR.



What is Roslyn?

Ans: Roslyn is the codename-that-stuck for the open-source compiler for C# and Visual Basic.NET. It is an open source, cross-platform, public language engine for C# and VB. The conversations about Roslyn were already ongoing when “Mads Torgersen” joined Microsoft in 2005 - just before .NET 2.0 would ship. That conversation was about rewriting C# in C# which is a normal practice for programming languages. But there was a more practical and important motivation: the creators of C# were not programming in C# themselves; they were coding in C++.

.NET CORE FAQ's

What is .NET Core?

Ans: The .NET Core platform is a new .NET stack that is optimized for open-source development. .NET Core has two major components. It includes a runtime that is built from the same codebase as the .NET Framework CLR. The .NET Core runtime includes the same GC and JIT (RyuJIT) but doesn't include features like Application Domains or Code Access Security. .NET Core also includes the base class libraries. These libraries are the same code as the .NET Framework class libraries but have been factored to enable to ship as smaller set of libraries. .NET Core refers to several technologies including ASP.NET Core, Entity Framework Core, and more.

What are the characteristics of .NET Core?

Ans: .NET Core has the following characteristics:

- **Cross Platform:** Runs on Windows, macOS, and Linux operating systems.
- **Open Source:** The .NET Core framework is open source, using MIT and Apache 2 licenses. .NET Core is a .NET Foundation project.
- **Modern:** It implements modern paradigms like asynchronous programming, no-copy patterns using struts', and resource governance for containers.
- **Performance:** Delivers high performance with features like hardware intrinsic, tiered compilation, and Span<T>.
- **Consistent Across Environments:** Runs your code with the same behavior on multiple operating systems and architectures, including x64, x86, and ARM.
- **Command-line Tools:** Includes easy-to-use command-line tools that can be used for local development and for continuous integration.
- **Flexible Deployment:** You can include .NET Core in your app or install it side-by-side (user-wide or system-wide installations). Can be used with Docker containers.

What is the composition of .NET Core?

Ans: NET Core is composed of the following parts:

- The .NET Core runtime, which provides a type system, assembly loading, a garbage collector, native interop, and other basic services. .NET Core framework libraries provide primitive data types, app composition types, and fundamental utilities.
- The ASP.NET Core runtime, which provides a framework for building modern, cloud-based, internet-connected apps, such as web apps, IOT apps, and mobile backend.
- The .NET Core SDK and language compilers (Roslyn and F#) that enable the .NET Core developer experience.
- The dotnet command, which is used to launch .NET Core apps and CLI commands. It selects and hosts the runtime, provides an assembly loading policy, and launches apps and tools.

What is .NET Core SDK?

Ans: The .NET Core SDK (Software Development Kit) includes everything you need to build and run .NET Core applications using command line tools or any editor like Visual Studio. It also contains a set of libraries and tools

that allow developers to create .NET Core applications and libraries. It contains the following components that are used to build and run applications:

1. The .NET Core CLI.
2. .NET Core libraries and runtime.
3. The dotnet driver.

What is .NET Core Runtime?

Ans: This includes everything you need to run a .NET Core Application. The runtime is also included in the SDK. When an app author publishes an app, they can include the runtime with their app. If they don't include the runtime, it's up to the user to install the runtime. There are three different runtimes you can install on Windows:

- ASP.NET Core runtime: Runs ASP.NET Core apps. Includes the .NET Core runtime.
- Desktop runtime: Runs .NET Core WPF and .NET Core Windows Forms desktop apps for Windows. Includes the .NET Core runtime.
- .NET Core runtime: This runtime is the simplest runtime and doesn't include any other runtime. It's highly recommended that you install both ASP.NET Core runtime and Desktop runtime for the best compatibility with .NET Core apps.

What's the difference between SDK and Runtime in .NET Core?

Ans: The SDK is all the stuff that is needed for developing a .NET Core application easier, such as the CLI and a compiler. The runtime is the "virtual machine" that hosts/runs the application and abstracts all the interaction with the base operating system.

What is the difference between .NET Core and .NET Framework?

Ans: .NET Core and .NET Framework share many of the same components and you can share code across the two. Some key differences include:

- .NET Core is cross-platform and runs on Linux, macOS, and Windows. .NET Framework only runs on Windows.
- .NET Core is open-source and accepts contributions from the community. The .NET Framework source code is available but does not take direct contributions.
- The majority of .NET innovation happens in .NET Core.
- .NET Framework is included in Windows and automatically updated machine-wide by Windows Update. .NET Core is shipped independently.

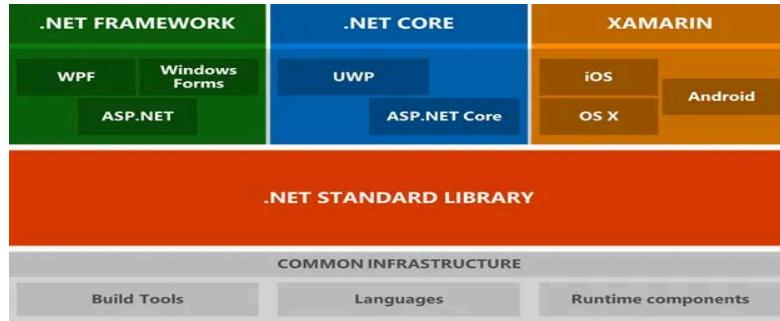
What is the difference between .NET Core and Mono?

Ans: To be simple, Mono is third party implementation of .Net Framework for Linux/Android/iOS and .Net Core is Microsoft's own implementation for same.

What's the difference between .NET Core, .NET Framework, and Xamarin?

Ans: difference between .NET Core, .NET Framework and Xamarin are:

- .NET Framework is the "traditional" flavor of .NET that's distributed with Windows. Use this when you are building a desktop Windows or UWP app or working with older ASP.NET 4.8.
- .NET Core is cross-platform .NET that runs on Windows, Mac, and Linux. Use this when you want to build console or web apps that can run on any platform, including inside Docker containers.
- Xamarin is used for building mobile apps that can run on iOS, Android, or Windows Phone devices.



What is the support policy to .NET Core?

Ans: .NET Core is supported by Microsoft on Windows, macOS, and Linux. It's updated for security and quality regularly (the second Tuesday of each month). .NET Core binary distributions from Microsoft are built and tested on Microsoft-maintained servers in Azure and follow Microsoft engineering and security practices.

Red Hat supports .NET Core on Red Hat Enterprise Linux (RHEL). Red Hat builds .NET Core from source and makes it available in the Red Hat Software Collections. Red Hat and Microsoft collaborate to ensure that .NET Core works well on RHEL (Red Hat Enterprise Linux).

Tizen (developed by Samsung) supports .NET Core on Tizen platforms.

How much does .NET Core cost?

Ans: .NET Core is an open-source and cross-platform version of .NET that is maintained by Microsoft and the .NET community on GitHub. All aspects of .NET Core are open source including class libraries, runtime, compilers, languages, ASP.NET Core web framework, Windows desktop frameworks, and Entity Framework Core data access library. There are no licensing costs, including for commercial use.

What is GitHub?

Ans: GitHub is a code hosting platform for collaboration and version control. It is a repository (usually abbreviated to "repo") is a location where all the files for a particular project are stored which lets you (and others) work together on projects. Each project has its own repo, and you can access it with a unique URL. Git is an open-source version control system that was started by "Linus Torvalds" - the same person who created Linux. Git is similar to other version control systems—Subversion, CVS, and Mercurial to name a few.

What is the release schedule for .NET Core?

Ans: .NET Core 2.1 and .NET Core 3.1 are the current LTS releases made available on August 2018 and December 2019, respectively. After .NET Core 3.1, the product will be renamed to .NET and LTS releases will be made available every other year in November. So, the next LTS release will be .NET 6, which will ship in November 2021. This will help customers plan upgrades more effectively.

How many versions do we have for .NET Core?

Ans: This table tracks release dates and end of support dates for .NET Core versions.

Version	Original Release Date	Support Level	End of Support
.NET Core 3.1	December 3, 2019	LTS	December 3, 2022
.NET Core 3.0	September 23, 2019	EOL	March 3, 2020
.NET Core 2.2	December 4, 2018	EOL	December 23, 2019

.NET Core 2.1	May 30, 2018	LTS	August 21, 2021
.NET Core 2.0	August 14, 2017	EOL	October 1, 2018
.NET Core 1.1	November 16, 2016	EOL	June 27, 2019
.NET Core 1.0	June 27, 2016	EOL	June 27, 2019

EOL (end of life) releases have reached end of life, meaning it is no longer supported and recommended moving to a supported version.

LTS (long-term support) releases have an extended support period. Use this if you need to stay supported on the same version of .NET Core for longer.

.NET 5 (.NET Core vNext)

.NET 5 is the next step forward with .NET Core. This new project and direction are a game-changer for .NET. With .NET 5, your code and project files will look and feel the same no matter which type of app you're building. You'll have access to the same runtime, API, and language capabilities with each app. The project aims to improve .NET in a few keyways:

- Produce a single .NET runtime and framework that can be used everywhere and that has uniform runtime behaviors and developer experiences.
- Expand the capabilities of .NET by taking the best of .NET Core, .NET Framework, Xamarin and Mono.
- Build that product out of a single code-base that developers (Microsoft and the community) can work on and expand together and that improves all scenarios.

Microsoft skipped the version 4 because it would confuse users that are familiar with the .NET Framework, which has been using the 4.x series for a long time. Additionally, they wanted to clearly communicate that .NET 5 is the future for the .NET platform. They are also taking the opportunity to simplify naming. They thought that if there is only one .NET going forward, they don't need a clarifying term like "Core". The shorter name is a simplification and communicates that .NET 5 has uniform capabilities and behaviors. Feel free to continue to use the ".NET Core" name if you prefer it.

Runtime experiences:

Mono is the original cross-platform implementation of .NET. It started out as an open-source alternative to .NET Framework and transitioned to targeting mobile devices as iOS and Android devices became popular. Mono is the runtime used as part of Xamarin.

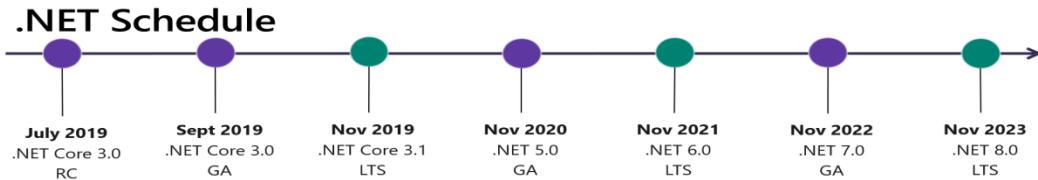
Core CLR is the runtime used as part of .NET Core. It has been primarily targeted at supporting cloud applications, including the largest services at Microsoft, and now is also being used for Windows desktop, IoT and machine learning applications.

Taken together, the .NET Core and Mono runtimes have a lot of similarities (they are both .NET runtimes after all) but also valuable unique capabilities. It makes sense to make it possible to pick the runtime experience you want. They are in the process of making Core CLR and Mono drop-in replacements for one another and will make it as simple as a build switch to choose between the different runtime options.

.NET – A unified platform



.NET Schedule: .NET 5 is shipped in November 2020, and then they intend to ship a major version of .NET once a year, every November:



- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

The .NET 5 Project is an important and exciting new direction for .NET. You will see .NET become simpler but also have broader and more expansive capability and utility. All new development and feature capabilities will be part of .NET 5, including new C# versions. We see a bright future ahead in which you can use the same .NET APIs and languages to target a broad range of application types, operating systems, and chip architectures. It will be easy to make changes to your build configuration to build your applications differently, in Visual Studio, Visual Studio for Mac, Visual Studio Code, and Azure DevOps or at the command line.

.NET 5.0 is the next major release of .NET Core following 3.1. They named this new release .NET 5.0 instead of .NET Core 4.0 for two reasons:

1. They skipped version numbers 4.x to avoid confusion with .NET Framework 4.x.
 2. They dropped “Core” from the name to emphasize that this is the main implementation of .NET going forward.
- .NET 5.0 supports more types of apps and more platforms than .NET Core or .NET Framework.

Note: ASP.NET Core 5.0 is based on .NET 5.0 but retains the name “Core” to avoid confusing with ASP.NET MVC 5. Likewise, Entity Framework Core 5.0 retains the name “Core” to avoid confusing it with Entity Framework 5 and 6.

The .NET 5 projects is an important and exciting new direction for .NET. You will see .NET become simpler but also have broader and more expansive capability and utility. All new development and feature capabilities will be part of .NET 5, including new C# versions.

We see a bright future ahead in which you can use the same .NET APIs and languages to target a broad range of application types, operating systems, and chip architectures. It will be easy to make changes to your build configuration to build your applications differently, in Visual Studio, Visual Studio for Mac, Visual Studio Code, and Azure DevOps or at the command line.

The current and latest version of .NET is 8.0 that was launched in November, 2023 with lots of exciting features for building Platform Independent applications targeting Windows, Linux, and Mac.

C# Programming Language

C# (pronounced see sharp, like the musical note \sharp , but written with the number sign) is a general-purpose, programming language encompassing strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed around 2000 by Microsoft as part of its .NET initiative and later approved as an international standard by ECMA (European Computer Manufacturers Association) in 2002 and ISO (International Organization for Standardization) in 2003.

The name “C Sharp” was inspired by the musical notation where sharp indicates that the written note should be made a semitone i.e., higher in pitch. This is like the language name of C++, where “++” indicates that a variable should be incremented by 1 after being evaluated. The sharp symbol also resembles a ligature of 4 “+” symbols (in a two-by-two grid), further implying that the language is an increment of C++. Due to technical limitations of display and the fact that the sharp symbol is not present on most keyboard layouts, the number sign “#” was chosen to approximate the sharp symbol in the written name of the programming language.

C# was designed by Anders Hejlsberg, and its development team is currently led by Mads Torgersen. C# has Procedural; Object Oriented syntax based on C++ and includes influences from several programming languages, most importantly Delphi and Java with a particular emphasis on simplification. The most recent stable version is 12, which was released in November 2023.

History: During the development of the .NET, the libraries were originally written using a managed code compiler system called “Simple Managed C” (SMC). In January 1999, Anders Hejlsberg formed a team to build a new language at the time called “COOL”, which stood for “C-like Object Oriented Language”.

Microsoft had considered keeping the name “COOL” as the final name of the language but chose not to do so for trademark reasons. By the time .NET project was publicly announced at the July 2000 in Professional Developers Conference, the language had been renamed “C#”, and the libraries and ASP.NET runtime had been ported to “C#”.

Anders Hejlsberg is C#’s principal designer and lead architect at Microsoft and was previously involved with the design of Turbo Pascal, Borland Delphi, and Visual J++. In interviews and technical papers, he has stated that flaws in most major programming languages (e.g., C++, Java, Delphi, and Smalltalk) drove the design of the C# language.

Design Goals: The ECMA standard lists these design goals for C#.

- The language is intended to be a simple, modern, general-purpose, and object-oriented language.
- The language, and implementations thereof, should provide support for software engineering principles such as strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection.
- Support for internationalization is very important.
- The language is intended for use in developing software components suitable for deployment in distributed environments.
- Portability is very important for programmers, especially those already familiar with C and C++.
- C# is intended to be suitable for writing applications for both hosted and embedded systems, ranging from the very large that use sophisticated operating systems, down to the very small having dedicated functions.

Versions of the language: 1.0, 1.2, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0

New features in C# 2.0:

- Generics
- Partial types
- Anonymous methods
- Iterators
- Nullable value types
- Getter/setter separate accessibility
- Static classes
- Delegate inference
- Null coalescing operator

New features in C# 3.0:

- Implicitly typed local variables
- Object initializers
- Collection initializers
- Auto-Implemented properties
- Anonymous types
- Extension methods
- Query expressions
- Lambda expressions
- Expression trees
- Partial methods

New features in C# 4.0:

- Dynamic binding
- Named and optional arguments
- Generic covariant and contravariant
- Embedded interop types

New features in C# 5.0:

- Asynchronous methods
- Caller info attributes
- Compiler API

New features in C# 6.0:

- Static imports
- Exception filters
- Auto-property initializers
- Default values for getter-only properties
- Expression bodied members
- Null propagator
- String interpolation

- nameof operator
- Index initializers
- Await in catch/finally blocks

New features in C# 7.0:

- Out variables
- Tuples and deconstruction
- Pattern matching
- Local functions
- Expanded expression bodied members
- Ref locals and returns
- Discards
- Binary Literals and Digit Separators
- Throw expressions

New features in C# 7.1:

- Async main method
- Default literal expressions
- Inferred tuple element names
- Pattern matching on generic type parameters

New features in C# 7.2:

- Techniques for writing safe efficient code
- Non-trailing named arguments
- Leading underscores in numeric literals
- private protected access modifier
- Conditional ref expressions

New features in C# 7.3:

- Accessing fixed fields without pinning

- Reassigning ref local variables
- Using initializers on stackalloc arrays
- Using fixed statements with any type that supports a pattern
- Using additional generic constraints

New features in C# 8.0:

- Readonly members
- Default interface methods
- Pattern matching enhancements:
 - Switch expressions
 - Property patterns
 - Tuple patterns
 - Positional patterns
- Using declarations
- Static local functions
- Disposable ref structs
- Nullable reference types
- Asynchronous streams and asynchronous disposable
- Indices and ranges
- Null-coalescing assignment
- Unmanaged constructed types
- Enhancement of interpolated verbatim strings

New features in C# 9.0 (Supported on .NET 5 only):

- Records
- Init only setters
- Top-level statements
- Pattern matching enhancements
- Native sized integers

- Function pointers
- Suppress emitting localsinit flag
- Target-typed new expressions
- static anonymous functions
- Target-typed conditional expressions
- Covariant return types
- Extension GetEnumerator support for foreach loops
- Lambda discard parameters
- Attributes on local functions
- Module initializers
- New features for partial methods

New features in C# 10 (Supported on .NET 6 only):

- ☒ Record structs
- ☒ Improvements of structure types
- ☒ Interpolated string handlers
- ☒ global using directives
- ☒ File-scoped namespace declaration
- ☒ Extended property patterns
- ☒ Improvements on lambda expressions
- ☒ Allow const interpolated strings
- ☒ Record types can seal ToString()
- ☒ Improved definite assignment
- ☒ Allow both assignment and declaration in the same deconstruction
- ☒ Allow AsyncMethodBuilder attribute on methods
- ☒ CallerArgumentExpression attribute
- ☒ Enhanced #line pragma

New features in C# 11 (Supported on .NET 7 only):

- Raw string literals
- Generic math support
- Generic attributes
- UTF-8 string literals
- Newlines in string interpolation expressions

- List patterns
- File-local types
- Required members
- Auto-default structs
- Pattern match Span<char> on a constant string
- Extended nameof scope
- Numeric IntPtr
- ref fields and scoped ref
- Improved method group conversion to delegate
- Warning wave 7

New features in C# 12 (Supported on .NET 8 only):

- Primary constructors
- Collection expressions
- ref readonly parameters
- Default lambda parameters
- Alias any type
- Inline arrays
- Experimental attribute
- Interceptors

.NET Framework, .NET Core and .NET 5 support for C# language versions:

Target Runtime	version	C# language version
.NET	8.x	C# 12
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Framework	all	C# 7.3

Writing a program by using different Programming Approaches

To write a program we generally follow 2 different approaches in the industry:

1. Procedural Programming Approach
2. Object Oriented Programming Approach

Procedural Programming Approach: This is a very traditional approach followed by the industry to develop applications till 70's. E.g.: COBOL, Pascal, FORTRAN, C, etc.

In this approach a program is a collection of **members** like **variables** and **functions**, and the **members** that are defined inside the program should be explicitly called for execution and we do that calling from "**main**" function because it is the **entry point** of any **program** that is developed by using any **programming language**.

C Program

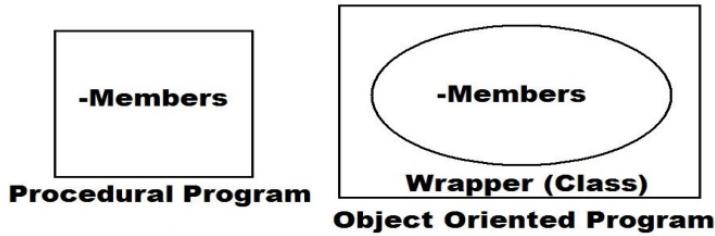
```
-Collection of Members (Variables & Functions)
void main() <= Entry Point
{
    -Call the members from here for execution
}
```

Note: the drawbacks of procedural programming languages are they don't provide **security** and **re-usability**.

Object Oriented Programming Approach: This came into existence in late 70's to overcome the drawbacks of Procedural Programming Language's by providing **Security** and **Re-usability**.

E.g.: CPP, Python, Java, C#, etc.

In an **Object-Oriented Programming** approach also, a program is a Collection of **Members** like **Variables** and **Functions** only, but the main difference between **Object Oriented Languages** and **Procedural Languages** is, here to protect the **members** of a program we put them under a **container** or **wrapper** known as a "**class**".



What is a class?

Ans: it is a **user-defined type** very much like **structures** we have learnt in **C** language, i.e., by using these we can define new types, whereas the difference between the two are, structure in "C" language can contain only **variables** in it but **class** of **Object-Oriented** languages can contain both **variables** and **functions** also.

Syntax to define Class and Structure:

```
struct <Name>           class <Name>
{
    -Variables
};

    -Functions
};
```

Example:

<pre>struct Student { int Id; char Name[25]; float Marks, Fees; };</pre>	<pre>class Employee { int Id; string Name, Job; float Salary; -Can be defined with functions also };</pre>
--	--

In the above case int, float and char are pre-defined structures whereas string is a pre-defined class which we are calling them as types, same as that Student and Employee are also types (user-defined). The other difference between int, float, char, and string types, as well as Student and Employee types is the 1st 4 are scalar types which can hold 1 and only 1 value under them whereas the next 2 are complex types which can hold more than 1 value under them.

How to consume a type?

Ans: types can't be consumed directly because they do not have any memory allocation.

```
int = 100; //Invalid
```

So, to consume a type first we need to create a copy of that type:

```
int i = 100; //Valid
```

Note: In the above case "i" is a copy of pre-defined type int for which memory gets allocated and the above rule of types can't be consumed directly, applies both to pre-defined and user-defined types also.

<pre>int i;</pre>	<i>//i is a copy of pre-defined type int</i>
<pre>string s;</pre>	<i>//s is a copy of pre-defined type string</i>
<pre>Student ss;</pre>	<i>//ss is a copy of user-defined type Student</i>
<pre>Employee emp;</pre>	<i>//emp is a copy of user-defined type Employee</i>

Note: Generally, copies of scalar types like int, float, char, bool, string, etc. are known as **variables**, whereas copies of complex types which we have defined like Student and Employee are known as **Objects** or **Instances**.

Conclusion: After defining a **class** or **structure** if we want to consume them, first we need to create a **copy** of them and then only the **memory** which is required for execution gets allocated and by using that copy (**Object** or **Instance**) only we can call members that are defined under them.

CPP Program

```
class Example
{
    -Collection of Members (Variables & Functions)
};

void main() <= Entry Point
{
    -Create the object of class
    -Call members of class by using the object created
}
```

Note: CPP is the first Object Oriented Programming Language which came into existence, but still, it suffers from a criticism that it is not fully Object-Oriented Language; because in CPP Language we can't write main function inside of the class and according to the standards of Object-Oriented Programming each and every Member of the Program should be inside of the class.

The reason why we write main function outside of class is, if it is defined inside of the class then it becomes a member of that class and members of a class can be called only by using object of that class, but unfortunately we create object of class inside main function only, so until and unless object of class is created main function can't be called and at the same time until and unless main function starts its execution, object creation will not take place and this is called as "**Circular Dependency**" and to avoid this problem, in CPP Language we write main function outside of the class.

Object Oriented Programming in Java: Java language came into existence in the year **1995** and here also a class is a collection of members like **variables** and **methods**. While designing the language, designers have taken it as a challenge that their language should not suffer from the criticism that it is not fully Object Oriented, so they want "**main**" method of the class to be present inside of the class only and still execute without the need of class object and to do that they have divided members of a class into 2 categories, like:

- Non-static Members
- Static Members

Every member of a class is by default a non-static member only and what we have learnt till now in **C** or **C++** Languages is also about non-static members only, whereas if we prefix any of those members with **static** keyword, we call them as **Static Members**.

```
class Test
{
    int x = 100;           //Non-Static Member
```

```
    static int y = 200;      //Static Member
}
```

Note: Static members of the class doesn't require object of that class for both **initialization** and **execution** also, whereas non-static members require it, so in Java Language "**main method**" is defined inside of the class only but declared as **static**, so even if it is inside of the class also it can start the execution without the need of class object.

Java Program

```
class Example
{
    -Collection of Members (Static & Non-Static)
    public static void main(string[] args)
    {
        -Create the object of class
        -Call non-static members of class by using the object created
        -Call static members of class by prefixing the class name
    }
}
```

Object Oriented Programming in C#: C# Language came into existence after **Java** and was influenced by **Java**, so in **C#** Language also the programming style will be same as **Java** i.e., defining **Main** method inside the class by declaring it as "**static**".

C# Program

```
class Example
{
    -Collection of Members (Static & Non-Static)
    static void Main()
    {
        -Create the instance of class
        -Call non-static members of class by using the instance created
        -Call static members of class by prefixing the class name
    }
}
```

Note: In **Java** or **C# Languages** if at all the class contains only **Main** method in it, we **don't** require creating **object** or **instance** of that class to run the **class**.

Visual Studio Installation:

Step 1: Visit the site: <https://visualstudio.microsoft.com/downloads>.

Step 2: Download and install the latest version of **Visual Studio Community Edition** i.e., VS 2022 (Version 17) latest. When you click on the download button it will download the installer for downloading and we find it in the bottom of LHS in the Browser or you can also find it in downloads folder, click on it to launch the installer.

Step 3: Once the installer is loaded and opened choose the below options in it:

1. Workloads:

- ASP.NET and web development

- Azure development
 - .NET desktop development
 - Data storage and processing
 - Visual Studio extension development
2. **Individual Components:** Select all the Checkbox's under ".NET" option except "Out of support" Checkbox's and "ML.NET Model Builder" CheckBox and Android, Mac or iOS and Linux options. Now scroll down and go to "Code Tools" section and under that select the CheckBox "LINQ to SQL Tools".
 3. **Language Packs:** Don't change anything here (default is English).
 4. **Installation Folders:** Don't change anything here also.

Step 4: Click on Download and Install button to complete the installation.

Writing programs by using C# Language: C# language has lot of standards to be followed while writing code, as following:

1. It's a case sensitive language so we need to follow the below rules and conventions:
 - I. All keywords in the language must be in lower case (rule).
 - II. While consuming the libraries, names will be in Pascal Case (rule). E.g.: WriteLine, ReadLine
 - III. While defining our own classes and members to name them we can follow any casing pattern, but Pascal case is suggested (convention).
2. A C# program should be saved with ".cs" extension.
3. We can use any name as a file name under which we write the program, but class name is suggested to be used as file name also.
4. To write programs in C# we use an IDE (Integrated Development Environment) known as Visual Studio but we can also write them by using any text editor like Notepad also.

Syntax to define a class:

```
[<modifiers>] class <Name>
{
  -Define Members here
}
```

[] => Optional
<> => Any

- **Modifiers** are some special keywords that can be used on a class like **public, internal, static, abstract, partial, sealed**, etc.
- Class is a keyword to tell that we are defining a class just like we used, struct keyword to define a structure in C Language.
- <Name> refers to name of the class for identification.
- Members refer to contents of the class like fields, methods, etc.

Syntax to define Main Method in the class:

```
static void Main( [string[] args] )
{
  -Stmt's
}
```

- **Static** is a keyword we use to declare a member as static member and if a member is declared as static, instance of the class is not required to call or execute it. In C# Main method should be declared static to start the execution from there.

- `Void` is a keyword to specify that the method is non-value returning.
- `Main` is name of the method, which can't be changed and more over it should be in Pascal Case only.
- If required (optional) we can pass parameters to `Main` method but it should be of type `string array` only.
- Statements refer to the `logic` we want to implement.

Writing the first program in C# using Notepad:

Step 1: Open Notepad and write the below code in it:

```
class First
{
    static void Main()
    {
        System.Console.Clear();
        System.Console.WriteLine("My first C# program using Notepad.");
    }
}
```

Step 2: Saving the program.

Create a `folder` on any drive of your computer with the name "**CSharp**" and save the above file into that folder naming it as "**First.cs**".

Step 3: Compilation of the program.

We need to compile our C# program by using **C# Compiler** at "**Developer Command Prompt**", provided along with the **Visual Studio** software, and to do that go to **Windows Search** and search for "**Developer Command Prompt for VS**", click on it to open. Once it is open it will be pointing to the location where **Visual Studio** software is installed, so change to the location where you have created the folder and compile the program as below:

Syntax: csc <File Name>

E.g.: <drive>:\CSharp> csc First.cs ↵

Once the program is compiled successfully it generates an output file with the name `First.exe` that contains "**CIL (Common Intermediate Language)** or **MSIL (Microsoft Intermediate Language) Code**" in it which we need to execute.

Step 4: Execution of the program.

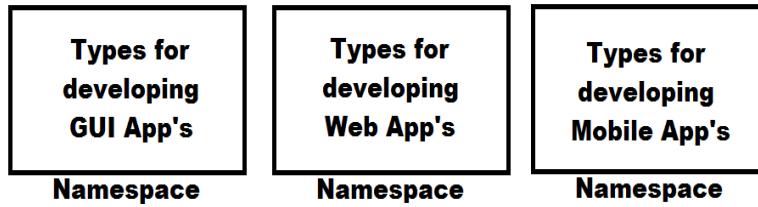
Now at the same Command Prompt we can run our First.exe file as below:

E.g.: <drive>:\CSharp> First ↵

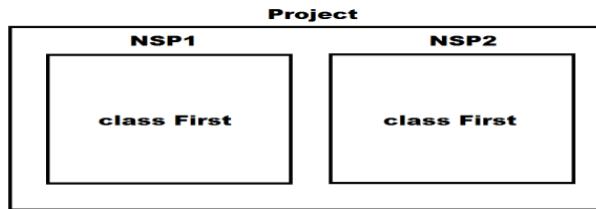
System.Console.WriteLine & System.Console.Clear: `Console` is a pre-defined class under the `libraries` of our language which provides with a set of `static` members using which we can perform `IO` operations on the standard `IO` devices. `WriteLine` is a method in the class `Console` to display output on the monitor, and apart from `WriteLine` method there are many other methods present in the class `Console` like: `Write`, `Read`, `ReadLine`, `.ReadKey`, `Clear`, etc. and all these methods are also `static`, so we can call them directly by prefixing the class name.

`System` is a `namespace`, and a `namespace` is a logical container for types like: `Class`, `Structure`, `Interface`, `Enum` and `Delegate`, and we use these namespaces in a language for 2 reasons:

1. **Grouping** related types i.e., types that are designed for developing similar kind of App's are grouped together under a namespace for easy access and identification as following:



2. To overcome the naming **collision** i.e., if a project contains multiple types with the same name, we can overcome conflict between names by putting them under separate namespaces, as following:



Note: Every pre-defined type in our **Libraries** is defined under some **namespace** and we can also define types under **namespaces**, and we will learn this process while working with **Visual Studio**.

If a type is defined under any namespace, then, whenever and wherever we want to consume the type, we need to prefix namespace name to type name, and this is the reason why in our previous program we have referred to "**Console**" class as "**System.Console**". To overcome the problem of prefixing namespace, name every time before the type, we are provided with an option of "**importing a namespace**" which is done by "**using directive**" as following:

Syntax: using <namespace>;
 using System;
 using Microsoft.VisualBasic;

Note: We can import any no. of namespaces as above but each import should be a separate statement.

What is a directive?

Ans: directive in our language is an instruction that is given to the compiler which it must follow, by importing the namespace we are telling the C# compiler that types consumed in the program are from the imported namespace.

To test the process of importing a namespace write the below code in Notepad and execute:

```
using System;
class Second
{
    static void Main()
    {
        Console.Clear();
        Console.WriteLine("Importing a namespace.");
    }
}
```

Note: If there are multiple namespaces containing a type with same name then it's not possible to consume those types by importing the namespace, and in such cases it's mandatory to refer to each type by prefixing the namespace name to them as following:

E.g.: NSP1.First NSP2.First

using static directive: This is a new feature introduced in “C# 6.0” which allows us to import a type and then consume all the **static members** of that type without a type name prefix.

Syntax: **using static <namespace.type>;**
using static System.Console;

To test the process of importing a class write below code in Notepad and execute:

```
using static System.Console;
class Third
{
    static void Main()
    {
        Clear();
        WriteLine("Importing a type.");
    }
}
```

Data Types in C#

C# Types	CIL Types	Size/Capacity	Default Value
Integer Types			
byte	System.Byte	1 byte (0 - 255)	0
short	System.Int16	2 bytes (-2 ^ 15 to 2 ^ 15 - 1)	0
int	System.Int32	4 bytes (-2 ^ 31 to 2 ^ 31 - 1)	0
long	System.Int64	8 bytes (-2 ^ 63 to 2 ^ 63 - 1)	0
sbyte	System.SByte	1 byte (-128 to 127)	0
ushort	System.UInt16	2 bytes (0 to 2 ^ 16 - 1)	0
uint	System.UInt32	4 bytes (0 to 2 ^ 32 - 1)	0
ulong	System.UInt64	8 bytes (0 to 2 ^ 64 - 1)	0
Decimal Types			
float	System.Single	4 bytes	0
double	System.Double	8 bytes	0
decimal	System.Decimal	16 bytes	0
Boolean Type			
bool	System.Boolean	1 byte	False
DateTime Type			
DateTime	System.DateTime	8 bytes	01/01/0001 00:00:00
Unique Identifier Type			
Guid	System.Guid	32 bytes	00000000-0000-0000-0000-000000000000
Character Types			
char	System.Char	2 bytes	\0

string	System.String		Null
Base Type			
object	System.Object		Null

- All the above types are known as primitive/pre-defined types i.e., they are defined under the libraries of our language which can be consumed from anywhere.
- All C# Types after compilation of source code gets converted into CIL Types and in CIL Format these types are either classes or structures defined under the "System" namespace. String and Object types are classes, whereas rest of the other 15 types, are structures.
- short, int, long and sbyte types can store signed integer (Positive or Negative) values whereas ushort, uint, ulong and byte types can store un-signed integer (Pure Positive) values only.
- Guid is a type used for storing Unique Identifier values that are loaded from SQL Server Database, which is a 32-byte alpha-numeric string holding a Global Unique Identifier value and it will be in the following format: 00000000-0000-0000-0000-000000000000.
- The size of char type has been increased to 2 bytes for giving support to Unicode characters i.e., characters of languages other than English.
- We are aware that every English language character has a numeric value representation known as ASCII; characters of languages other than English also have that numeric value representation and we call it as Unicode.

char ch = 'A'; => ASCII => Binary
char ch = '3'; => Unicode => Binary

- Just like ASCII values converts into binary for storing by a computer; Unicode values also converts into binary, but the difference is ASCII requires 1 byte of memory for storing its value whereas Unicode requires 2 bytes of memory for storing its value.
- String is a variable length type i.e.; it doesn't have any fixed size and its size varies based on the value that is assigned to it.
- Object is a parent of all the types, so capable of storing any type of value in it and more over it is also a variable length type.

Syntax to declare fields and variables in a class:

[<modifiers>] [const] [readonly] <type> <name> [=default value] [,...n]

```
class Test
{
    int x; //Field (Global Scope)
    static void Main()
    {
        int y = 100; //Variable (Local Scope)
    }
}
```

- “<type>” refers to the data type of field or variable we want to declare, and it can be any of the 17 types we discussed above.
- “<name>” refers to the name of the field or variable and it should be unique within the location.
E.g.: int i; float f; bool b; char c; string s; object o; DateTime dt; Guid id;

- Fields and variables can be initialized with any value at the time of their declaration and if they are not initialized then every **field** has a default value which is “0” for all numeric types, “false” for bool type, “\0” for char type, “00000000-0000-0000-0000-000000000000” for Guid type, “01/01/0001 00:00:00” for DateTime type and “null” for string and object types.

Note: Variables doesn't have any default value so it is must to initialize them while declaration or before consumption.
 E.g.: int x = 100;

- Modifiers are generally used to define the scope of a field i.e., from where it can be accessed, and the default scope for every member of a class in our language is **private** which can either be changed to **public** or **internal** or **protected**.
- “**const**” is a keyword to declare a constant and those constants values can't be modified once after their declaration:

```
const float pi = 3.14f; //Declaration and Initialization
```

- “**readonly**” is a keyword to declare a field as readonly and these readonly field values also can't be modified, but after initialization:

```
readonly float pi;           //Declaration
pi = 3.14f;                 //Initialization
```

Note: decimal values are by default treated as double by the compiler, so if we want to use them as float the value should be suffixed with character “**f**” and “**m**” to use the value as decimal.

```
float pi = 3.14f;  double pi = 3.14;  decimal pi = 3.14m;
```

```
using System;
class TypesDemo
{
    static int x;           //Field
    static void Main()
    {
        Console.Clear();
        Console.WriteLine("Field x value is: " + x + " and it's type is: " + x.GetType());

        int y = 10;           //Variable
        Console.WriteLine("Variable y value is: " + y + " and it's type is: " + y.GetType());
        float f = 3.14f;      //Variable
        Console.WriteLine("Variable f value is: " + f + " and it's type is: " + f.GetType());
        double d = 3.14;       //Variable
        Console.WriteLine("Variable d value is: " + d + " and it's type is: " + d.GetType());
        decimal de = 3.14m;    //Variable
        Console.WriteLine("Variable de value is: " + de + " and it's type is: " + de.GetType());
        bool b = true;         //Variable
        Console.WriteLine("Variable b value is: " + b + " and it's type is: " + b.GetType());
        Char ch = 'A';         //Variable
        Console.WriteLine("Variable ch value is: " + ch + " and it's type is: " + ch.GetType());
    }
}
```

Note: `GetType` is a pre-defined method which returns the **type (CIL Format)** of a **variable or field or instance** on which it is called.

Data Types are divided into 2 categories:

1. Value Types
2. Reference Types

Value Types:

- All fixed length types come under the category of value types. E.g.: **integer types, decimal types, bool type, char type, DateTime type and Guid type**.
- Value types will store their values on “**Stack**” and **Stack** is a **Data Structure** that works on a principal “**First in Last out (FILO)**” or “**Last in First out (LIFO)**”.
- Each program when it starts the execution, a **Stack** will be created and given to that program for storing its values and in the end of program’s execution **Stack** is destroyed.
- Every program will be having its own **Stack** for storing values that are associated with the program and no 2 programs can share the same **Stack**.
- **Stack** is under the control of **Operating System** and memory allocation is performed only in **fixed length** i.e., once allocated that is **final** which can’t either be **increased** or **decreased** also.

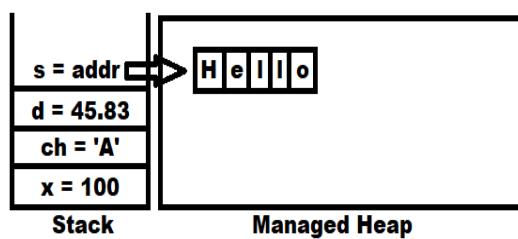
Reference Types:

- All variable length types come under the category of **reference types** and these types will store their values on “**Heap**” memory and their **address or reference** is stored on “**Stack**”. E.g.: **String and Object types**.
- **Heap** memory doesn’t have any limitations like **Stack**, and it provides a beautiful feature called **Dynamic Memory Management** and because of that, all programs in execution can share the same **Heap**.
- In older programming languages like **C and C++**, **Heap** memory is under developer’s control, whereas in modern programming languages like **Java and .NET**, **Heap** memory is under control of a special component known as “**Garbage Collector**”, so we call **Heap** memory in these languages as “**Managed Heap**”.

Suppose if we declare fields in a program as following:

```
int x = 100;    char ch = 'A';    double d = 45.83;    string s = "Hello";
```

Then memory is allocated for them as following:



Nullable Value Types: These are introduced in **C# 2.0** for storing **null** values under **value types** because; by default **value types** can’t store **null** values under them whereas **reference types** can store **null** values under them.

```
string str = null;          //Valid
object obj = null;          //Valid
int i = null;                //Invalid
decimal d = null;            //Invalid
```

To overcome the above problem **nullable value types** came into picture and if we want a **value type** as **nullable** we need to **suffix** the type with “?” and declare it as following:

int? i = null;	//Valid
decimal? d = null;	//Valid

Implicitly typed variables: This is a new feature introduced in **C# 3.0**, which allows declaring variables by using “**var**” keyword, so that the type of that variable is identified based on the value that is assigned to it, for example:

var i = 100;	//i is of type int
var f = 3.14f;	//f is of type float
var b = true;	//b is of type bool
var s = "Hello";	//s is of type string

Note: While using implicitly typed variables we have 2 restrictions:

1. We can't declare these variables with-out initialization. E.g.: var x; //Invalid
2. We can use “**var**” only on variables but not on fields.
3. Changing the type after declaration is not possible. E.g.: var i = 100; //Valid
 i = 34.56; //Invalid

Dynamic Type: This is a new type introduced in **C# 4.0**, which is very similar to implicitly typed variables we discussed above, but here in place of “**var**” keyword we use “**dynamic**”.

Differences between “var” and “dynamic”

Var	Dynamic
Type identification is performed at compilation time.	Type identification is performed at runtime.
Once the type is identified can't be changed to a new type again. var v = 100; //v is of type int v = 34.56; //Invalid	We can change the type of dynamic with a new value in every statement. dynamic d = 100; //d is of type int d = 34.56; //d is of type double (Valid)
Can't be declared with-out initialization. var v; //Invalid	Declaration time initialization is only optional. dynamic d; //Valid d = 100; //d is of type int d = false; //d is of type bool d = "Hello"; //d is of type string d = 34.56; //d is of type double
Can be used for declaring variables only.	Can be used for declaring variables and fields also.

```
using System;
class VarDynamic
{
    static void Main()
    {
        var i = 100;
        Console.WriteLine(i.GetType());
        var c = 'A';
```

```

Console.WriteLine(c.GetType());
var f = 45.67f;
Console.WriteLine(f.GetType());
var b = true;
Console.WriteLine(b.GetType());
var s = "Hello";
Console.WriteLine(s.GetType());
Console.WriteLine("-----");
dynamic d;
d = 100;
Console.WriteLine(d.GetType());
d = 'Z';
Console.WriteLine(d.GetType());
d = 34.56;
Console.WriteLine(d.GetType());
d = false;
Console.WriteLine(d.GetType());
d = "Hello";
Console.WriteLine(d.GetType());
}
}

```

Boxing and Un-Boxing:

Boxing is a process of converting **values types** into **reference types**:

```

int i = 100;
object obj = i;                                //Boxing

```

Unboxing is a process of converting a **reference type** which is created from a **value type** back into **value type**, but un-boxing requires an **explicit conversion**:

```

int j = Convert.ToInt32(obj);                     //Un-Boxing

```

Value Type	=> Reference Type	//Boxing	
Value Type	=> Reference Type	=> Value Type	//UnBoxing
Reference Type	=> Value Type		//Invalid

Note: “Convert” is a predefined class in “System” namespace and “ToInt32” is a static method under that class, and this class also provides other methods for conversion like “ToDouble”, “ToSingle”, “.ToDecimal”, “.ToBoolean”, etc, to convert into different types.

Taking input from end user's, into a program:

```

using System;
class AddNums
{
    static void Main()
    {

```

```

Console.Clear();

Console.Write("Enter 1st number: ");
string s1 = Console.ReadLine();
double d1 = Convert.ToDouble(s1);

Console.Write("Enter 2nd number: ");
string s2 = Console.ReadLine();
double d2 = double.Parse(s2);

double d3 = d1 + d2;

Console.WriteLine("Sum of " + d1 + " & " + d2 + " is: " + d3);
Console.WriteLine("Sum of {0} & {1} is: {2}", d1, d2, d3);
Console.WriteLine($"Sum of {d1} & {d2} is: {d3}");
}
}

```

ReadLine method of the `Console` class is used for reading the input from end users into our programs and this method will perform 3 actions when used in the program, those are:

1. Waits at the command prompt for the user to enter a value.
2. Once the user finishes entering his value, immediately the value will be read into the program.
3. Returns the value as string by performing boxing because return type of the method is string.

public static string ReadLine()

Note: after reading the value as string in our program we need to convert it back into its original type by performing an un-boxing which can be done in either of the ways:

<pre> string s1 = Console.ReadLine(); double d1 = Convert.ToDouble(s1); </pre> <p style="text-align: center;">or</p> <pre> double d1 = Convert.ToDouble(Console.ReadLine()); </pre>	<pre> string s2 = Console.ReadLine(); double d2 = double.Parse(s2); </pre> <p style="text-align: center;">or</p> <pre> double d2 = double.Parse(Console.ReadLine()); </pre>
---	---

Parse(String): this method is used to convert the `string` representation of a `value` to its equivalent type on which the method is called.

```

string s1 = "100" ;      int i = int.Parse(s1);
string s2 = "34.56";    double d = double.Parse(s2);
string s3 = "true";     bool b = bool.Parse(s3);

```

String Interpolation: String interpolation provides a more readable and convenient syntax to create formatted strings than a string composite formatting feature. An interpolated string is a string literal that might contain interpolation expressions. When an interpolated string is resolved to a result string, items with interpolation expressions are replaced by the string representations of the expression results. This feature is available starting with `C# 6.0`.

Operators in C#: An operator is a special symbol that tells the compiler to perform a specific mathematical or logical operation when used between a set of operands. C# has a rich set of built-in operators as below:

Arithmetic Operators	=>	<code>+,-,*,/,%</code>
-----------------------------	----	------------------------

Assignment Operators	=> =, +=, -=, *=, /=, %=
Relational Operators	=> ==, !=, <, <=, >, >=
Logical Operators	=> &&, , !
Unary Operators	=> ++, --
Miscellaneous Operators	=> sizeof(), typeof(), is, as, ?: (Ternary), ?? (Coalesce)

```

using System;
class OperatorsDemo
{
    static void Main()
    {
        Console.WriteLine(sizeof(double));
        Console.WriteLine(typeof(float));

        double d = 34.56;
        object obj1 = d;
        if(obj1 is double)
            Console.WriteLine("d is of type System.Double");
        string str1 = "Hello World";
        object obj2 = str1;
        string str2 = (string)obj2;
        string str3 = obj2 as string;
        string str4 = obj2.ToString();
        int i = 100;
        Console.WriteLine(i == 100 ? "Hello India" : "Hello World");

        string Country1 = null;
        string Country2 = null;
        Console.WriteLine(Country1 ?? Country2);
        Country2 = "India";
        Console.WriteLine(Country1 ?? Country2);
        Country1 = "America";
        Console.WriteLine(Country1 ?? Country2);
    }
}

```

Conditional Statements in C#: it's a block of code that executes based on a **condition** and they are divided into 2 categories.

1. **Conditional Branching**
2. **Conditional Looping**

Conditional Branching: these statements allow us to branch the code depending on whether certain conditions are met or not. C# has 2 constructs for branching code, the “**if**” statement which allow us to test whether a specific condition is met or not, and the “**switch**” statement which allows us to compare an expression with a number of different values.

Syntax of “if” Condition:

```

if (<condition>
    [{} <statement(s)>; {}])
else if (<condition>
    [{} <statement(s)>; {}])
[<multiple else if's>]
else
    [{} <statement(s)>; {}]

```

Note: Curly braces are **optional** if the conditional block contains **single statement** in it or else it's **mandatory**.

```

using System;
class IfDemo
{
    static void Main()
    {
        Console.WriteLine("Enter 1st number: ");
        double d1 = double.Parse(Console.ReadLine());
        Console.WriteLine("Enter 2nd number: ");
        double d2 = double.Parse(Console.ReadLine());

        if(d1 > d2)
            Console.WriteLine("1st number is greater than 2nd number.");
        else if(d1 < d2)
            Console.WriteLine("2nd number is greater than 1st number.");
        else
            Console.WriteLine("Both the given numbers are equal.");
    }
}

```

Syntax of “switch case” Condition:

```

switch (<expression>)
{
    case <value>:
        <stmts>;
        break;
[<multiple case blocks>]
    default:
        <stmts>;
        break;
}

```

Note: In **C** and **CPP** languages using a **break** statement after each **“case block”** is only optional whereas it is mandatory in case of **C#** language, which should be used after **“default block”** also.

```

using System;
class SwitchDemo
{

```

```

static void Main()
{
    Console.Write("Enter Student Id. (1-3): ");
    int Id = int.Parse(Console.ReadLine());
    switch(Id)
    {
        case 1:
            Console.WriteLine("Student 1");
            break;
        case 2:
            Console.WriteLine("Student 2");
            break;
        case 3:
            Console.WriteLine("Student 3");
            break;
        default:
            Console.WriteLine("No student exists with the given Id.");
            break;
    }
}
}

```

Conditional Looping: C# provides 4 different loops that allow us to execute a block of code **repeatedly** until a certain **condition** is met and those are:

1. **for loop**
2. **while loop**
3. **do..while loop**
4. **foreach loop**

Every loop requires 3 things in common:

1. **Initialization:** This set's the **starting** point for a loop.
2. **Condition:** This decides when the loop must **end**.
3. **Iteration:** This takes the loop to the **next level** either in **forward** or **backward** direction.

Syntax of “for loop”:

```

for (initializer;condition;iteration)
{
    -<statement's>;
}

```

Example:

```

for(int i = 1;i <= 100;i++)
{
    Console.WriteLine(i);
}

```

Syntax of “while loop”:

```

while (<condition>)
{
    -<statement's>;
}

```

Example:

```
int i = 1;
while(i <= 100)
{
    Console.WriteLine(i);
    i++;
}
```

Syntax of “do..while loop”:

```
do
{
    -<statement's>;
}
while (<condition>);
```

Example:

```
int i = 1;
do
{
    Console.WriteLine(i);
    i++;
}
while (i <= 100);
```

Note: the minimum no. of **execution's** in case of a “**for loop**” and “**while loop**” are “**0**” because in both these cases the loop starts its execution only when the given **condition** is **satisfied** whereas the minimum no. of **execution's** in case of a “**do...while loop**” is “**1**” because in this case after **executing** the loop for **first** time, then it will check for the condition to continue the loop's execution.

Syntax of “foreach loop”:

```
foreach(type var_name in array_name|collection_name)
{
    -<statements>;
}
```

Note: **foreach loop** is specially designed for accessing values from an **array** or **collection**.

Jump Statements: these are statements which will transfer the control from 1 line of execution to another line. C# has no. of statements that allows jumping to another line in a program, those are:

1. **goto**
2. **break**
3. **continue**
4. **return**

goto: it allows us to jump directly to another specified line in the program, indicated by a **label** which is an identifier followed by a **colon**.



```
Console.WriteLine("Hello World.");
xxx:
Console.WriteLine("Goto Called.");
```

break: it is used to **exit** from a **case** in a **switch** statement and used to **exit** from any **conditional loop** statement which will switch the control to the statement immediately after end of the loop.

```
for (int i = 1;i <= 100;i++)
{
    Console.WriteLine(i);
    if (i == 50)
        break;
}
Console.WriteLine("End of the loop.");
```



continue: it is used only in a loop which will jump the control to iteration part of the loop without executing any other statement that is present next to it.

```
for (int i = 1;i <= 100;i++)
{
    if (i == 7 || i == 77)
        continue;
    Console.WriteLine(i);
}
```



return: this is used to terminate the execution of a method in which it is used and jumps out of that method, while jumping out it can also carry a value out of that method which was only optional.

```
using System;
class Table
{
    static void Main()
    {
        Console.Clear();
        Console.Write("Enter an un-signed integer value: ");
        bool Status = uint.TryParse(Console.ReadLine(), out uint x);

        if (Status == false)
        {
            Console.WriteLine("Please enter un-signed integer's only.");
            return;
        }
        if (x == 0 || x == 1)
        {
            Console.WriteLine("Please enter a number greater than 1.");
            return;
        }
}
```



```

        Console.WriteLine();
        for (int i=1;i<=10;i++)
        {
            Console.WriteLine($"x * {i} = {x*i}");
        }
    } //End of the method
}

```

Arrays

It is a set of similar type values that are stored in a sequential order either in the form of a **row** or **rows & columns**. In C# language also we access the values of an array by using the **index** only which will start from “**0**” and ends at the “**no. of items - 1**”. In C# arrays can be declared either as **fixed length** or **dynamic**, where a fixed length array can store a pre-defined no. of items whereas the size of a dynamic array increases as we add new items to it.

1-Dimensional Array's: these arrays will store data in the form of a row and are declared as following:

Syntax: **<type>[] <array_name> = new <type>[length|size]**

Example:

```

int[] arr = new int[5];           //Declaration and Initialization with default values
Or
int[] arr;                      //Declaration
arr = new int[5];                //Initialization with default values
Or
int[] arr = { <list of values> }; //Declaration and Initialization with given set of values

using System;
class SDArray1
{
    static void Main()
    {
        Console.Clear();
        int x = 0;
        int[] arr = new int[6];

        //Accessing values of a SD Array by using for loop
        for(int i=0;i<6;i++)
        {
            Console.Write(arr[i] + " ");
        }
        Console.WriteLine();

        //Assigning values to a SD Array by using for loop
        for(int i=0;i<6;i++)
        {
            x += 10;
            arr[i] = x;
        }
    }
}

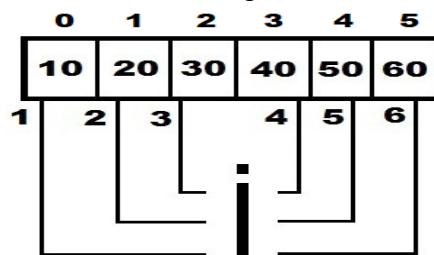
```

```

//Accessing values of a SD Array by using foreach loop
foreach(int i in arr)
    Console.WriteLine(i + " ");
Console.WriteLine();
}
}

```

foreach loop: this loop is specially designed for accessing values from an array or a collection. When we use foreach loop for accessing values, the loop starts providing access to values of the array or collection by assigning the values to loop variable in a sequential order as following:



Differences between for loop and foreach loop in accessing values of an array or collection:

1. In case of a “for loop”, the loop variable refers to **index** of the array whereas in case of a “foreach loop”, the loop variable refers to **values** of the array.
2. By using a “for loop” we can either access or assign values to an array whereas by using a “foreach loop” we can only access the values from an array.
3. In case of a “for loop”, the data type of loop variable is always **int** only irrespective of the type of values in the array, whereas in case of a “foreach loop”, the data type of loop variable will be same as the **type** of values in the array.

```

int[] iarr = { 10, 20, 30, 40, 50 };
double[] darr = { 12.34, 34.56, 56.78, 78.90, 90.12 };
string[] sarr = { "Red", "Blue", "Green", "Yellow", "Magenta" };

```

<pre> for (int i=0;i<iarr.Length;i++) for (int i=0;i<darr.Length;i++) for (int i=0;i<sarr.Length;i++) </pre>	<pre> foreach(int i in iarr) foreach(double d in darr) foreach(string s in sarr) </pre>
---	---

Array Class: this is a pre-defined class under the “**System**” namespace which provides with a set of members in it to perform actions on an array, those are:

Sort(Array arr)	=> void	//Method
Reverse(Array arr)	=> void	//Method
Copy(Array source, Array target, int n)	=> void	//Method
GetLength(int dimension)	=> int	//Method
Length	=> int	//Property (Field)

```

using System;
class SDArray2
{

```

```

static void Main()
{
    Console.Clear();
    int[] arr = { 54, 79, 59, 8, 42, 22, 93, 3, 73, 38, 67, 48, 18, 61, 32, 86, 15, 27, 81, 96 };

    for(int i=0;i<arr.Length;i++)
        Console.Write(arr[i] + " ");
    Console.WriteLine();
    Array.Sort(arr);
    foreach(int i in arr)
        Console.Write(i + " ");
    Console.WriteLine();

    Array.Reverse(arr);
    foreach(int i in arr)
        Console.Write(i + " ");
    Console.WriteLine();
    int[] brr = new int[10];
    Array.Copy(arr, brr, 7);
    foreach(int i in brr)
        Console.Write(i + " ");
    Console.WriteLine();
}
}

```

2-Dimensional Array's: these arrays will store data in the form of **rows & columns**, and are declared as following:

Syntax: `<type>[,] <array_name> = new <type>[rows, columns]`

Example:

```

int[,] arr = new int[4,5];           //Declaration and Initialization with default values
or
int[,] arr;                         //Declaration
arr = new int[4,5];                  //Initialization with default values
or
int[,] arr = { <list of values> };   //Declaration and Initialization with given set of values

```

```

using System;
class TDArray
{
    static void Main()
    {
        int x = 0; int[,] arr = new int[4, 5];

        //Accessing values of TD Array by using foreach loop
        foreach(int i in arr)
            Console.Write(i + " ");
        Console.WriteLine();
    }
}

```

```

//Assigning values to TD Array by using nested for loop
for(int i=0;i<arr.GetLength(0);i++) {
    for(int j=0;j<arr.GetLength(1);j++) {
        x += 5; arr[i,j] = x;
    }
}

//Accessing values of TD Array by using nested for loop
for(int i=0;i<arr.GetLength(0);i++) {
    for(int j=0;j<arr.GetLength(1);j++)
        Console.Write(arr[i,j] + " ");
    Console.WriteLine();
}
}
}

```

Assigning values to 2-D Array at the time of it's declaration:

```

int[,] arr = {
    { 11, 12, 13, 14, 15 },
    { 21, 22, 23, 24, 25 },
    { 31, 32, 33, 34, 35 },
    { 41, 42, 43, 44, 45 }
};

```

Jagged Arrays: these are also 2-Dimensional arrays only which will store the data in the form of rows and columns but the difference is in-case of a 2-Dimensional array all the rows will be having equal no. of columns whereas in case of a jagged array the column size varies from row to row. Jagged arrays are also known as “array of arrays” because here each row is considered as a single dimensional array and multiple single dimensional arrays with different sizes are combined together to form a new array.

Syntax: <type>[][] <array_name> = new <type>[rows][]

Example:

```

int[][] arr = new int[4][];
//Declaration
or
int[][] arr = { <list of values> };
//Declaration & Initialization with given set of values

```

Note: in case of a jagged array, we can't initialize the array with default values at the time of its declaration i.e. first we need to specify the no. of rows and then pointing to each row we need to specify the no. of columns to that row, as following:

```

int[][] arr = new int[4][];
//Declaration
arr[0] = new int[5];
//Initialization of 1st row
arr[1] = new int[6];
//Initialization of 2nd row
arr[2] = new int[8];
//Initialization of 3rd row
arr[3] = new int[4];
//Initialization of 4th row

```

Internally the memory is allocated for the array as following:

	0	1	2	3	4	5	6	7	
0	1	2	3	4	5				→arr[0]
1	1	2	3	4	5	6			→arr[1]
2	1	2	3	4	5	6	7	8	→arr[2]
3	1	2	3	4					→arr[3]

```

using System;
class JArrayDemo
{
    static void Main() {
        Console.Clear();
        int[][] arr = new int[4][];
        arr[0] = new int[5];
        arr[1] = new int[6];
        arr[2] = new int[8];
        arr[3] = new int[4];

        //Accessing values of Jagged Array by using nested foreach loop
        foreach(int[] iarr in arr)
        {
            foreach(int x in iarr)
                Console.Write(x + " ");
            Console.WriteLine();
        }
        Console.WriteLine("-----");
        //Accessing values of Jagged Array by using for loop in foreach loop
        foreach(int[] iarr in arr)
        {
            for(int i=0;i<iarr.Length;i++)
                Console.Write(iarr[i] + " ");
            Console.WriteLine();
        }
        Console.WriteLine("-----");

        //Assigning values to Jagged Array by using for loop in foreach loop
        foreach(int[] iarr in arr)
        {
            for(int i=0;i<iarr.Length;i++)
            {
                iarr[i] = i + 1;
            }
        }
    }
}

```

```

//Accessing values of Jagged Array by using nested for loop
for(int i=0;i<arr.GetLength(0);i++)
{
    for(int j=0;j<arr[i].Length;j++)
        Console.Write(arr[i][j] + " ");
    Console.WriteLine();
}
Console.WriteLine("-----");

//Assigning values to Jagged Array by using nested for loop
for(int i=0;i<arr.GetLength(0);i++)
{
    for(int j=0;j<arr[i].Length;j++)
    {
        arr[i][j] = i + 1;
    }
}

//Accessing values of Jagged Array by using foreach loop in for loop
for(int i=0;i<arr.GetLength(0);i++)
{
    foreach(int x in arr[i])
        Console.Write(x + " ");
    Console.WriteLine();
}
}
}

```

Assigning values to Jagged Array at the time of its declaration:

```

int[][] arr =
{
    new int[5] { 11, 12, 13, 14, 15 },
    new int[6] { 21, 22, 23, 24, 25, 26 },
    new int[8] { 31, 32, 33, 34, 35, 36, 37, 38 },
    new int[4] { 41, 42, 43, 44 }
};

```

Implicitly typed arrays: Just like we can declare **variables** by using “**var**” keyword we can also declare **arrays** by using the same “**var**” keyword and here also the **type identification** is performed based on the **values** that are assigned to the **array**.

```

var iarr = new[] { 10, 20, 30, 40, 50 };           //Implicitly type integer array
var sarr = new[] { "Red", "Blue", "Green", "Yellow", "Magenta" }; //Implicitly typed string array
var darr = new[] { 12.34, 34.56, 56.78, 78.96, 90.12 }; //Implicitly typed double array

var jarr = new[]

```

```

{
    new[] { 11, 12, 13, 14, 15 },
    new[] { 21, 22, 23, 24, 25, 26 },
    new[] { 31, 32, 33, 34, 35, 36, 37, 38 },
    new[] { 41, 42, 43, 44 },
    new[] { 51, 52, 53, 54, 55, 56, 57 }
};

//Implicitly typed jagged integer array

```

Command Line Arguments: Arguments which are passed by the user or programmer to the **Main** method are known as **Command-Line Arguments**. Main method is the entry point for the execution of a program and this **Main** method can accept an **array of strings**.

```

using System;
class Params
{
    static void Main(string[] args)
    {
        foreach(string str in args) {
            Console.WriteLine(str);
        }
    }
}

```

After compilation of the program execute the program at Command Prompt as following:

```
<drive>:\CSharp> Params 100 Hello 34.56 A true ↵
```

Note: We can pass **any no. of values** as well as **any type of values** as **command line arguments** to the program, but each value should be separated with a **space** and all those values we passed will be captured in the **string array** (**args**) of **Main** method. In the above case (**100, Hello, 34.56, A, true**) are 5 values we have supplied to the **Main** method of **Params** class as command line arguments.

Adding a given set of numbers that are passed as Command Line Arguments:

```

using System;
class AddParams
{
    static void Main(string[] args)
    {
        double Sum = 0;
        foreach(string str in args) {
            Sum = Sum + double.Parse(str);
        }
        Console.WriteLine("Sum of given {0} no's is: {1}", args.Length, Sum);
    }
}

```

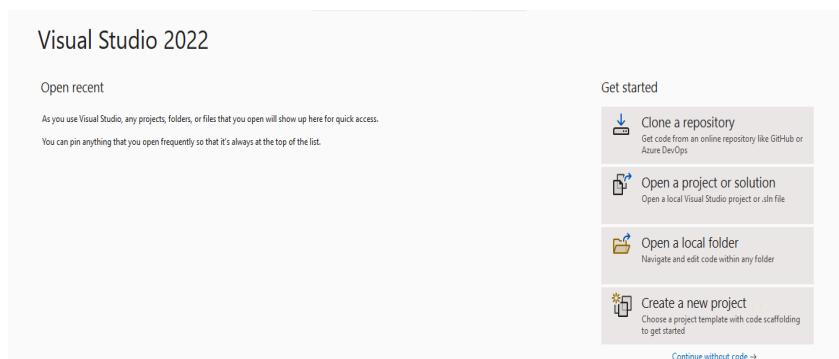
After compilation of the program execute the program at Command Prompt as following:

```
<drive>:\CSharp> AddParams 10 20 30 ↵
<drive>:\CSharp> AddParams 34.56 28.93 98.45 63.28 ↵
<drive>:\CSharp> AddParams 938.387 534 348.378 836 174.392 ↵
<drive>:\CSharp> AddParams 18 48.37 75 56.43 97 85.19 ↵
```

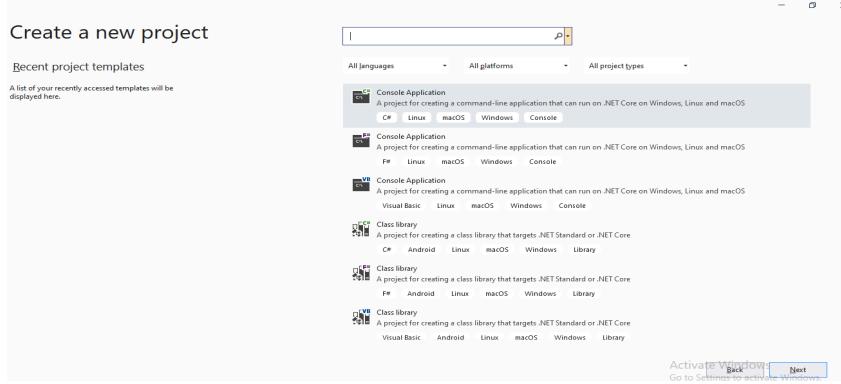
Working with Visual Studio

Visual Studio is an IDE (Integrated Development Environment) used for developing .NET Applications by using any .NET Language like C#, Visual Basic, F# etc., as well as we can develop any kind of applications like Console, Windows, and Web etc.

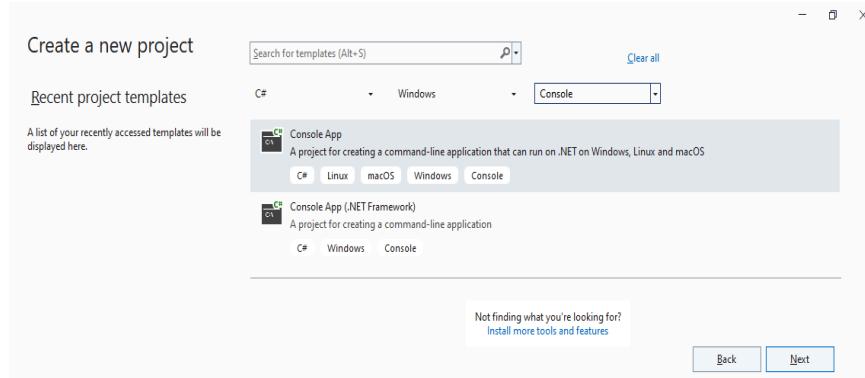
To open Visual Studio, go to Windows Search and search for Visual Studio 2022 and click on it to open, which will launch as following:



Applications that are developed under Visual Studio are known as Projects, where each Project is a collection of items like Class, Interface, Structure, Enum, Delegate, Html Files, XML Files, and Text Files etc. To create a Project, click on “Create a new project” option in the above Page which opens a new window as following:



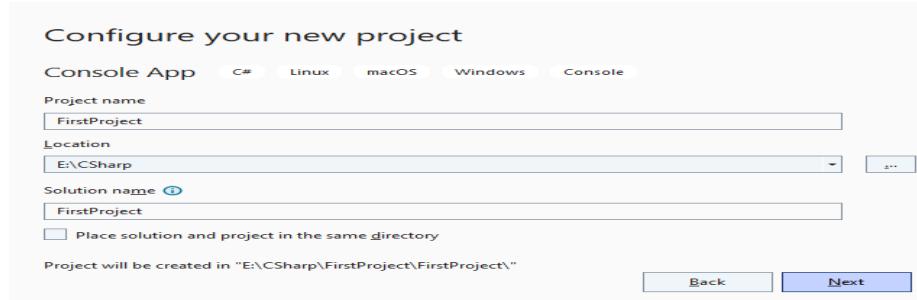
In the above window under “All languages” DropDownList select “C#”, under “All platforms” DropDownList select “Windows” and under “All project_types” DropDownList select “Console” which will display the options as following:



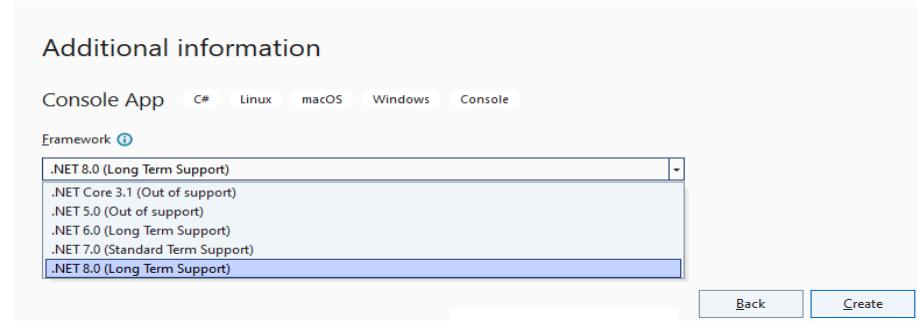
Now select “Console Application” in the above window and click “Next” button which opens a new window as following:



In that above window under “Project Name” TextBox enter the name of project as “FirstProject”, under Location TextBox enter or select our personal folder location i.e., “<drive>:\CSharp” and click on “Next” button:



This will open a new window asking to select the **Target Framework** choose **.NET 8.0 (Long-term support)** which is the latest version of **.NET** and then click on “**Create**” button:



This action will create a project and a file named “Program.cs” which contains the below code in it: Starting with **.NET 6 (C# 10.0)** and above, the project template for new **C# console App’s** generates the following code in the **Program.cs** file:

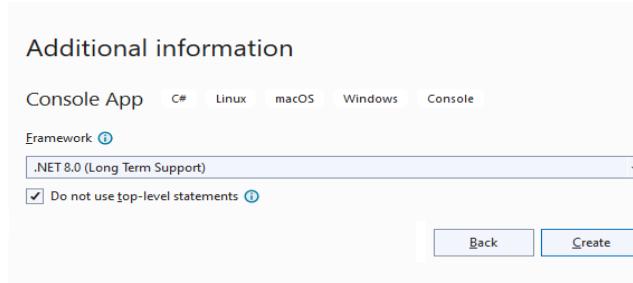
```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

This is a new feature in “**C# 10.0**” i.e., we don’t require to define a **class** and **Main** method explicitly which means the code we write in the file will directly execute as if the code we write in a **Main** method. In the above code first line is a comment and second line is a **WriteLine** statement to print **output** on the **monitor**. Whereas if we create the project by choosing the **Framework** as **.NET 5.0 (Out of support)** then it is **C# 9.0** and up to this version defining **class & Main** method is mandatory to run the code, so we find code in “**Program.cs**” file as below:

```
using System;
namespace FirstProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The above 2 forms of code represent the same class, program. Both are valid from **C# 10.0**. When you use the newer version, you only need to write the body of the **Main** method. The compiler synthesizes a **Program** class with a **Main** method and places all your **top-level statements** in that **Main** method. You don't need to include the other program elements; the compiler **generates** them for you.

Note: If you don't want to use the **top-level statements** and generate a **class** explicitly, we need to check the Checkbox "**Do not use top-level statements**" while creating the project, in the window where we selected the **Framework**:



From "**C# 10.0**" there is a concept of "**Global Imports**" i.e., we don't require writing **import statements** in all the files as we have done in case of **Notepad** by the help of "**using directive**". We can now write all the import statements that are required in our project with-in a single file prefixing the keyword "**global**", so that they are applied to all the classes in our project.

Syntax: `global using <Namespace_Name>;`

Example: `global using System;`

Note: By default, some namespaces are already **imported** for us to consume in our project created under **VS 2022** choosing "**.NET 6.0 or above**" within the file "**GlobalUsings.g.cs**" and the content of the file is as below:

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

To run the class either hit **Ctrl + F5** or go to "**Debug**" menu and select the option "**Start Without Debugging**" which will **save**, **compile**, and **executes** the program by displaying the output "**Hello World!**" on the console window because we have opened a "**Console App.**" Project. To close the console window that is opened, it will display a message "**Press any key to continue . . .**", so once we hit any key it will close the window and takes us back to the **Visual Studio**.

Note: if you are confused of writing the code without a class and Main method, simply delete the whole code in the file and write the below code:

```
namespace FirstProject
{
    class Program
```

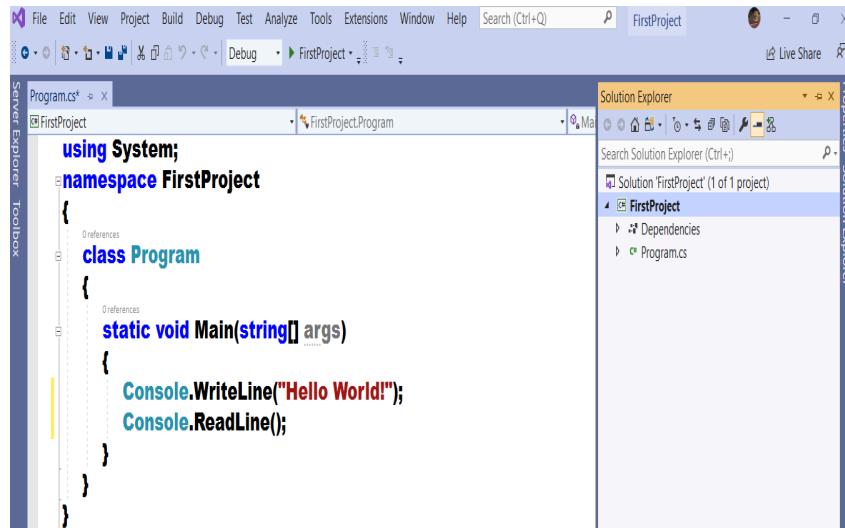
```

{
    static void Main()
    {
        Console.WriteLine("Hello World!");
    }
}

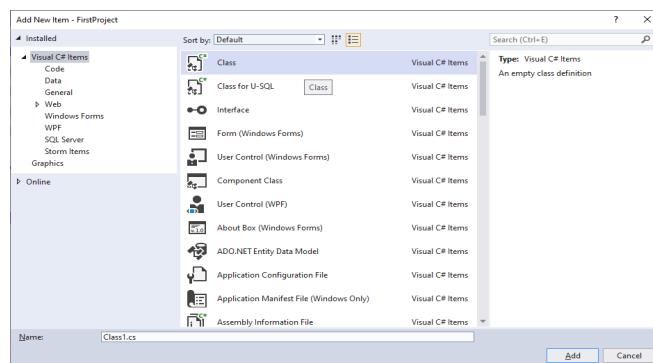
```

We can also run the class by hitting **F5** or clicking on the **FirstProject** button in the “Tool Bar” or go to “Debug” menu and select the option “Start Debugging”, but in this case it **save**, **compile** and **executes** the program but we can’t view the output because the **Console** window gets closed immediately and in this case to view the output we need to hold console window and to do that use “**Console.ReadLine();**” method after “**Console.WriteLine("Hello World!");**” in **Main** method of the program.

Adding new items in the project: under **Visual Studio** we find a window in the **RHS** known as **Solution Explorer** used for organizing the complete application, which allows us to **view**, **add** and **delete** items under the **projects**, if it is not visible in the **RHS** then go to “**View**” menu and select “**Solution Explorer**” which will launch it on **RHS** which looks as below:



To add new classes under **project**, open **Solution Explorer**, right click on the **project**, select **Add => choose “New Item”** option, which opens the “**Add New Item**” window as following:



In this window select “**Class**” template, specify a name to it in the bottom or leave the existing name and click on **Add** button, which adds the class under our project with the name “**Class1.cs**”. The new class we added, also comes under the same **Namespace**, i.e., “**FirstProject**” and we find the below code in it:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace FirstProjet
{
    internal class Class1
    {
    }
}
```

By default, the class will have 5 import statements, importing a set of **namespaces** and these are not important for us now because from “**C# 10.0**” all the above namespaces except “**System.Text**” are global imports, so you can either **delete** them or leave them as same.

Now under the class define a Main method which should now look as below:

```
namespace FirstProjet
{
    internal class Class1
    {
        static void Main()
        {
            Console.WriteLine("Second class under the project.");
            Console.ReadLine();
        }
    }
}
```

Now when we **run** the project, we get an **error** stating that there are **multiple entry points** in the project because the 2 classes that are defined in the project contains a **Main** method and every **Main** method is an entry point, so to resolve the problem we need to set a property known as “**Startup Object**”.

To set the “**Startup Object**” property, open Solution Explorer => right click on the project => select the option “**Edit Project File**”, which is an “**XML File**” with the name “**FirstProject.csproj**” added to the **document window** in **Visual Studio**. We can also open this “**.csproj**” file by double clicking on the project in **Visual Studio**.

Under the project file, by default we find the below code:

```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
```

```
</PropertyGroup>
</Project>
```

- ② `OutputType` element is used to specify the generated output file after compilation of the project will be having an “.exe” extension.
- ② `TargetFramework` element is used to specify the .NET Runtime Version we are using to build the project and currently we are using the latest version of “.NET” i.e., “.NET 8.0” and in this version of Runtime, C# version is “12.0”, same as this if we choose “.NET 7.0” then C# version is 11.0, for “.NET 6.0” C# version is 10.0, for “.NET 5.0” C# version is “9.0” and if we choose “.NET Core 3.1” then the version of C# is “8.0”.
- ② `ImplicitUsings` element is used to enable the feature global imports which is new from “C# 10.0” and if set as disable the feature will not work and we need to explicitly import the namespaces on top of each class with the help of “using” directive.
- ② `Nullable` element is used to enable a new feature “Non-Nullable Reference Types” which was introduced in “C# 8.0”. By default, Reference Types are Nullable, but we can make them non-Nullable by enabling the Nullable feature in the project file, so when we assign a Null value to them, we get a warning, but if we really want to assign a Null value to them, we need to declare them by suffixing with a “?”, for example: `string?` and `object?.` If we want to disable the feature of reference types not accepting null values by default change the value “enable” as “disable” under the `Nullable` element of the project file.

To set the “Startup Object” property, add “`<StartupObject></StartupObject>`” element within the `<ProjectGroup>` element and the code in “FirstProject.csproj” file should be as below now:

```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
<OutputType>Exe</OutputType>
<TargetFramework>net8.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
<StartupObject>FirstProject.Class1</StartupObject>
</PropertyGroup>
</Project>
```

Note: follow the same process to run the new classes we add in the project.

Object Oriented Programming

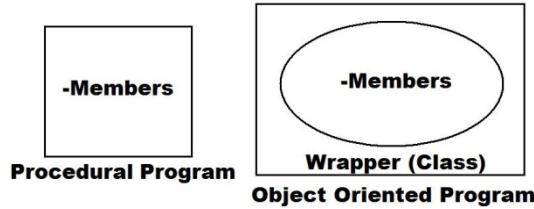
This is an approach that we use in the industry for developing application, introduced in late 70's or early 80's replacing traditional **Procedural Programming Approach** because **Procedural Programming Approach** doesn't provide **Security** and **Re-usability**, whereas these 2 are the main strength of **Object-Oriented Programming Approach**.

Any language to be called as **Object Oriented** needs to satisfy 4 important principals that are prescribed under the standards of **Object-Oriented Programming**, and they are:

1. **Encapsulation** => hiding the data
2. **Abstraction** => hiding the complexity
3. **Inheritance** => re-usability

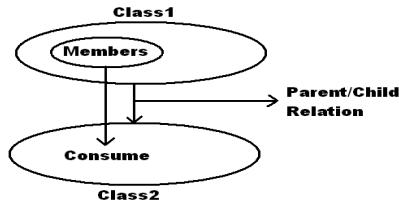
4. Polymorphism => behaving in different ways based on input received

Encapsulation: this is all about hiding of data or members of a program by wrapping them under a container known as a Class, which provides security for all its contents.



Abstraction: this is all about hiding the complexity of code and then providing with a set of interfaces to consume those functionalities, for example functions/methods in a program are good example for this, because here we are only aware of how to call them, but we are never aware of the underlying logic behind that implementation.

Inheritance: this provides re-usability i.e., members that are defined in 1 class can be consumed from other classes by establishing **parent/child** relation between the classes.



Polymorphism: behaving in different ways based on the input received is known as polymorphism i.e., whenever the input changes then the output or behavior also changes accordingly.

Class: It's a **user-defined type** which is in-turn a collection of **members** like:

- **Fields**
- **Methods**
- **Constructors**
- **Finalizers**
- **Properties**
- **Indexers**
- **Events**
- **De-constructors (Introduced in C# 7.0)**

Method: It is a named block of code which performs an **action** whenever it is called and after completion of that action it may or may not return any result of that action, and they are divided into 2 categories:

1. **Value returning method (Function)**
2. **Non-value returning method (Sub-Routine)**

Syntax to define a method:

```
[<modifiers>] void|type <Name>(<Parameter List>)
{
    -Stmt's or Logic
}
```

[] => Optional
=> Any

Modifiers are some special keywords which can be used on a method if required (optional) like **public**, **internal**, **protected**, **static**, **virtual**, **abstract**, **override**, **sealed**, **partial**, etc.

void|type is to tell whether our method is value returning or non-value returning i.e., “**void**” implies that the method is non-value returning, whereas if we want our method to return any value then we need to specify the **type** of value it has to return by using the “**type**”.

Example for non-value returning methods:

```
public static void Clear()  
public static void WriteLine(<type> var)
```

Example for value returning method:

```
public static string ReadLine()
```

Note: the return type of a method need not be any **pre-defined type** like **int**, **float**, **char**, **bool**, **DateTime**, **Guid**, **string**, **object**, etc., but it can also be any **user-defined type** also.

<Name> refers to the “**ID**” of method for identification.

[<Parameter List>]: if required we can pass parameters to our methods for execution, and parameters of a method will make an action **dynamic**, for example:

GetLength(0)	=> Returns rows
GetLength(1)	=> Returns columns

Syntax to pass parameters to a method:

```
[ref|out] [params] <type> <var> [=default value] [...n]
```

Where should we define methods?

Ans: As per the rule of **Encapsulation** methods should be defined inside of a **class**.

How to execute a method that is defined under a class?

Ans: The methods that are defined in a class must be **explicitly** called for execution, except **Main** method because **Main** is implicitly called.

How to call a method that is defined in a class?

Ans: Methods are of 2 types:

1. Non-Static **2. Static**

Note: By default, every method of a class is **non-static** only, and if we want to make it as **static**, we need to prefix the “**static**” modifier before the method as we are doing in case of Main method.

To call a method that is defined under any class we require to create **instance** of that class provided the methods are **non-static**, whereas if the methods are **static**, we can call them directly by using class name, for example **WriteLine** and **ReadLine** are **static** methods in class **Console** which we are calling in our code as **Console.WriteLine** and **Console.ReadLine**.

How to create instance of a class?

Ans: We create the instance of class as following:

Syntax: <class_name> <instance_name> = new <class_name> ([<List of values>])

Example:

```
Program p = new Program();      //Declaration and Initialization  
or  
Program p;                  //Declaration  
p = new Program();           //Initialization
```

Note: without using “new” keyword we can’t create the instance of a class in Java and .NET Languages.

Where should we create the instance of a class?

Ans: instance of a class can be created either with-in the same class or in other classes also.

If instance is created in the same class, it should be created under any static block; generally, we create instances in Main method because of 2 reasons:

1. **Entry Point of the program.**
2. **It is a static block.**

If instance is created in another class, then it can be created in any block of that new class i.e., either static or non-static also.

To try all the above, create a new **Console App.** project in **Visual Studio** naming it as “**OOPSProject**”, delete all the code that is present in the default file “**Program.cs**” and write the below code over there:

```
namespace OOPSProject  
{  
    internal class Program  
    {  
        //Non-value returning method without parameters  
        public void Test1() //Static in behavior  
        {  
            int x = 5;  
            for (int i = 1; i <= 10; i++)  
            {  
                Console.WriteLine($"{x} * {i} = {x * i}");  
            }  
        }  
        //Non-value returning method with parameters  
        public void Test2(int x, int ub) //Dynamic in behavior  
        {  
            for (int i = 1; i <= ub; i++)  
            {  
                Console.WriteLine($"{x} * {i} = {x * i}");  
            }  
        }  
        //Value returning method without parameters  
        public string Test3() //Static in behavior
```

```

{
    string str = "hello world";
    str = str.ToUpper();
    return str;
}
//Value returning method with parameters
public string Test4(string str) //Dynamic in behavior
{
    str = str.ToUpper();
    return str;
}
static void Main()
{
    Program p = new Program();
    //Calling non-value returning methods.
    p.Test1();
    Console.WriteLine();

    p.Test2(8, 15);
    Console.WriteLine();

    //Calling value returning methods
    string s1 = p.Test3();
    Console.WriteLine(s1);

    string s2 = p.Test4("hello india");
    Console.WriteLine(s2);
    Console.ReadLine();
}
}
}

```

Consuming a class from other classes: It is possible to **consume** a **class** and its **members** from other classes in 2 different ways:

1. Inheritance
2. Creating an instance

To test the second, add a new class in the project naming it as “TestProgram.cs” and write the below code in it:

```

internal class TestProgram
{
    public void CallMethods()
    {
        Program p = new Program();
        p.Test1();
        Console.WriteLine();
        p.Test2(9, 12);
        Console.WriteLine();
    }
}

```

```

        Console.WriteLine(p.Test3());
        Console.WriteLine(p.Test4("hello america"));
    }
    static void Main()
    {
        new TestProgram().CallMethods();           //Un-named instance
        Console.ReadLine();
    }
}

```

Note: Un-named **instances** are created and used when we want to call any single **member** of a **class** or when we want to use that **instance** only for 1 time.

Code files in a project: When we want to add a new class under any project, we first open the “**Add New Item**” window and in that we choose “**Class Item Template**”, which when added will add a file with a class template in it, same as that we also find “**Code File Item Template**” which when added will add a **blank file** and we need to write everything manually in it, just like we write code using **Notepad**.

Defining multiple classes in a file: It’s possible to define “n” no. of classes under a single **.cs** file, but “Main” method can be defined under 1 **class** only. Even if it is not **mandatory** it is **advised** to use the “**Class Name**” under which we defined “**Main**” method as the “**File Name**”.

User-defined return types to a Methods: The return type of a method need not be any **pre-defined type** but can also be any **user-defined type** also i.e., a type which is defined representing some **complex** data.

To test all the above, add a “**Code File**” under the project naming it as “**UserDefinedTypes.cs**” and write the below code in it:

```

namespace OOPSProject
{
    class Emp
    {
        public int? Id;
        public string? Name, Job;
        public double? Salary;
        public bool? Status;
    }
    class UserDefinedTypes
    {
        public Emp GetEmpDetails(int Id)
        {
            Emp emp = new Emp();
            emp.Id = Id;
            emp.Name = "Raju";
            emp.Job = "Manager";
            emp.Salary = 50000.00;
            emp.Status = true;
        }
    }
}

```

```

        return emp;
    }
    static void Main()
    {
        UserDefinedTypes udt = new UserDefinedTypes();
        Emp obj = udt.GetEmpDetails(1001);
        Console.WriteLine(obj.Id + " " + obj.Name + " " + obj.Job + " " + obj.Salary + " " + obj.Status);
        Console.ReadLine();
    }
}
}

```

In the above case “**Emp**” is a new type (**User-Defined and Complex**) and that **type** is used as a **return type** for our method “**GetEmpDetails**”.

Parameters of a Method: we define **parameters** to methods for making actions **dynamic** i.e., as discussed earlier every method is an **action** and to make those actions **dynamic**, we define **parameters** to methods.

Syntax for defining parameters to a method:

[ref | out] [params] <type> <parameter_name> [= default value] [, ..n]

Parameters of a method are classified as:

1. Input Parameters
2. Output Parameters
3. InOut Parameters

- Input parameters will bring values into the method for execution.
- Output parameters will carry results out of the method after execution.
- InOut Parameters are a combination of above 2 i.e., these parameters will 1st bring a value into the method for execution and after execution, the same parameter will carry results out of the method.

By default, every parameter is an **input** parameter whereas if we want to declare any parameter as **output** we need to prefix “**out**” keyword and to declare a parameter as **InOut** we need to prefix “**ref**” keyword before the parameter, as following:

public void Test(int a, out int b, ref int c)

To test **Output Parameters**, add a new class in the **Project** naming it as “**OutputParameters.cs**” and write the below code in it:

```

internal class OutPutParameters
{
    public void Math1(int a, int b, out int c, out int d)
    {
        c = a + b;
        d = a * b;
    }
    //Introduced in C# 7.0 i.e., Tuples
    public (int, int) Math2(int a, int b)
    {
        int c = a + b;
    }
}

```

```

        int d = a + b;
        return (c, d);
    }
    static void Main()
    {
        OutPutParameters p = new OutPutParameters();

        int Sum1, Product1;
        p.Math1(100, 25, out Sum1, out Product1);
        Console.WriteLine("Sum of the given number's is: " + Sum1);
        Console.WriteLine("Product of the given number's is: " + Product1 + "\n");

        p.Math1(100, 25, out int Sum2, out int Product2); //C# 7.0 Feature
        Console.WriteLine("Sum of the given number's is: " + Sum2);
        Console.WriteLine("Product of the given number's is: " + Product2 + "\n");

        (int Sum3, int Product3) = p.Math2(100, 25);
        Console.WriteLine("Sum of the given number's is: " + Sum3);
        Console.WriteLine("Product of the given number's is: " + Product3 + "\n");
        Console.ReadLine();
    }
}

```

Tuple: A **tuple** is a data structure in **C#**, often used when we want to return more than one value from a method. A **tuple** can be used to return a set of values as a result from a method and this feature was introduced in **C# 7.0**.

To test **InOut** parameters add a new class in the Project naming it as “**InOutParameters.cs**” and write the below code in it:

```

internal class InOutParameters
{
    public void Factorial(ref uint a)
    {
        if (a == 0 || a == 1)
        {
            a = 1;
        }
        else
        {
            uint result = 1;
            for(uint i=2;i<=a;i++)
            {
                result = result * i;
            }
            a = result;
        }
    }
}

```

```

}
static void Main()
{
    InOutParameters obj = new InOutParameters();
    uint f = 5;
    Console.WriteLine("Value of f before execution of the method: " + f);
    obj.Factorial(ref f);
    Console.WriteLine("Value of f after execution of the method: " + f);
    Console.ReadLine();
}
}

```

Params KeyWord: By prefixing this keyword before an **array** parameter of any method we get a chance to call that method without explicitly creating an array and pass to the method, but we can directly pass a set of values in a “Comma-Sepereated” list.

```
public void AddParams(params double[] args)
```

For example WriteLine method of Console class is defined as below:

```
public static void WriteLine(string format, params object[] args)
```

So we are able to call that method in our earlier program as following:

```
Console.WriteLine("{0} * {1} = {2}", x, i, x * i);
```

Note: while using the “**params**” keyword we have 2 restrictions:

1. We can use it only on 1 parameter of the method.
2. It can be used only on the last parameter of that method.

Default values to parameters: While defining methods we can assign default values to parameters of that method, so that those parameters will become “**optional**” and while calling that method it is not mandatory to pass values to those parameters. If the method is called without passing a value to those parameters then default value of that parameter will be used, for example:

```
public void AddNums(int x, int y = 50, int z = 25)
```

Note: In the above case x is a mandatory parameter whereas y and z are optional parameters and while defining methods with mandatory and optional parameters, mandatory parameters should be in the 1st place of parameter list, followed by optional parameters in the last.

To test “**params**” keyword and “**default valued parameters**” add a new class in the project naming it as “**MethodParameters.cs**” and write the below code in it:

```

internal class MethodParameters
{
    public void AddParams(params double[] args)
    {
        double Sum = 0;
        foreach (double arg in args)
        {

```

```

        Sum = Sum + arg;
    }
    Console.WriteLine($"Sum of {args.Length} no's in the array is: {Sum}");
}
public void AddNums(int x, int y = 50, int z = 25)
{
    Console.WriteLine($"Sum of given 3 no's is: {x + y + z}");
}
static void Main()
{
    MethodParameters obj = new MethodParameters();

    obj.AddParams(56.87);
    obj.AddParams(78, 12.35);
    obj.AddParams(12.34, 56.32, 87.21);
    obj.AddParams(10, 20, 30, 40, 50);
    Console.WriteLine();

    obj.AddNums(100);
    obj.AddNums(100, 100);
    obj.AddNums(100, z:100);
    obj.AddNums(100, 100, 100);
    Console.ReadLine();
}
}

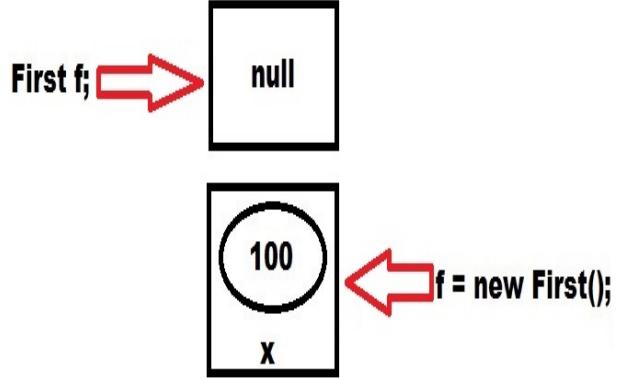
```

Understanding the difference between variable, instance, and reference of a class: to understand about a **variable**, **instance** and **reference** of a class add a new class in our **Project** naming it as “**First.cs**” and write below code in it:

```

internal class First
{
    public int x = 100;
    static void Main()
    {
        First f; //f is a variable of class
        f = new First(); //f is a instance of class
        Console.WriteLine(f.x);
        Console.ReadLine();
    }
}

```



Every **member** of a class, if it is **non-static** can be accessed from the **Main Method** only by using **instance** of that class. So, in the above case to print the value of "x", we created **instance** of class **First** under **Main method**.

A **variable** of class is a **copy** of class which is **not initialized** so it doesn't have any **memory allocation** and can't be used for **calling** or **accessing** the **members**.

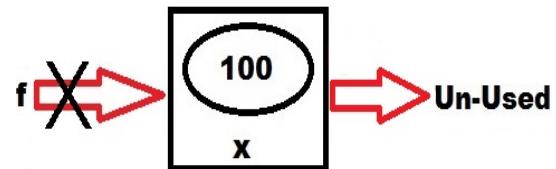
An **instance** of class is a **copy** of class which is initialized by using "**new**" keyword and for an **instance** **memory** is **allocated**, so by using this **instance** we can **access** or **call** members of that class.

De-referencing an Instance: it's possible to **de-reference** the **instance** of any class by assigning "**null**" to it and once "**null**" is assigned to **instance** we can't use that **instance** for calling **members** of class and if we try to do so, we get a **runtime error**. To test this, **re-write** the code under "**Main Method**" of class "**First**" as below:

```

First f = new First();
Console.WriteLine(f.x); //Valid
f = null;
Console.WriteLine(f.x); //Invalid (Causes runtime error)
Console.ReadLine();

```



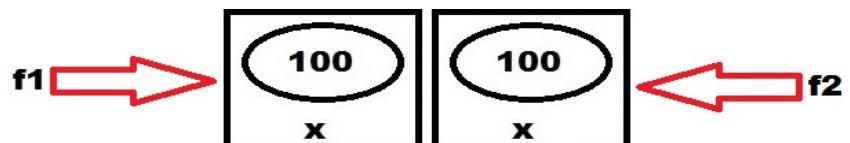
Note: once **null** is assigned to an **instance**, internally the **memory** which is **allocated** for that **instance** is **not** **de-allocated** immediately, but only gets **marked** as **un-used** and all those **un-used** objects **memory** will be **de-allocated** by "**Garbage Collector**" whenever it comes into **action**.

Creating multiple instances to a class: it is possible to create **multiple instances** to a class and each **instance** we **create** for the **class** will be having a **separate memory allocation** for its **members** as following:

```

First f1 = new First();
First f2 = new First();

```

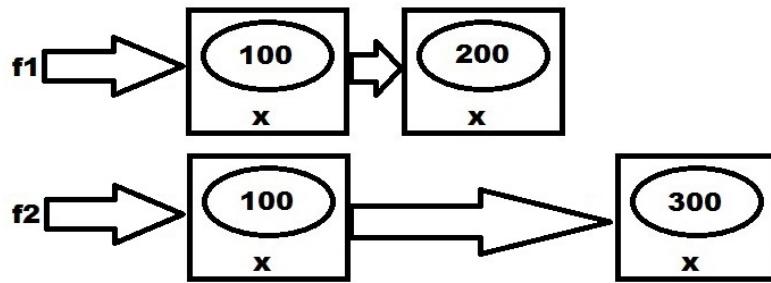


Instances are **unique** i.e., any **modifications** that we perform on the **members** of 1 **instance** will not **reflect** to the **members** of other **instances** of the class, and to test this **re-write** the code under **Main Method** of class **First** as below:

```

First f1 = new First();
First f2 = new First();
Console.WriteLine(f1.x + " " + f2.x);
f1.x = 200;
Console.WriteLine(f1.x + " " + f2.x);
f2.x = 300;
Console.WriteLine(f1.x + " " + f2.x);
Console.ReadLine();

```

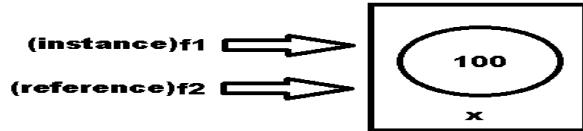


Reference of a class: we can initialize the **variable** of a class by using any **existing instance** of that class and we call it as a **reference** of the class. **References** of class will not have any **memory allocation** like **instances**, i.e., they will be **consuming** the **memory of instance** using which they are initialized, so a reference is just a **pointer** to an **instance**, as following:

```

First f1 = new First();
First f2 = f1; //f2 is a reference of class First

```

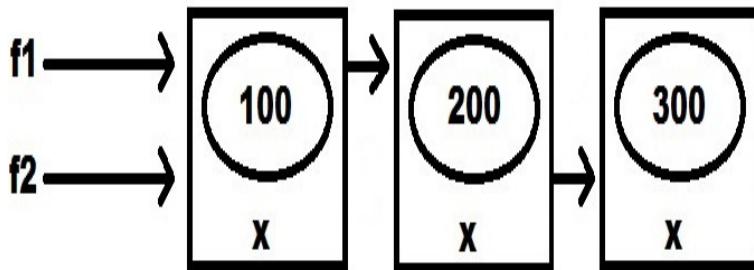


Because an **instance** and **reference** are accessing the **same memory**, changes that are performed on the **members** by using the **instance** will reflect when those **members** are **accessed** by using **reference** and **vice versa**. To test this, **re-write** code under **Main method** of class **First** as below:

```

First f1 = new First();
First f2 = f1;
Console.WriteLine(f1.x + " " + f2.x);
f1.x = 200;
Console.WriteLine(f1.x + " " + f2.x);
f2.x = 300;
Console.WriteLine(f1.x + " " + f2.x);
Console.ReadLine();

```

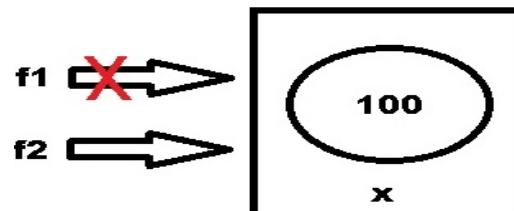


Note: when an **instance** and **references** are accessing the **same memory** and if **null** is assigned to any 1 of them, then the 1 to whom **null** is assigned can't access the **memory** anymore, but still the **others** can access it as is for calling the **members**. To test this, **re-write** code under **Main method** of class **First** as below:

```

First f1 = new First();
First f2 = f1;
f1 = null;
Console.WriteLine(f2.x); //Valid
Console.WriteLine(f1.x); //Invalid (Causes runtime error)
Console.ReadLine();

```



Variable of Class: this is a **copy** of **class** which is **not initialized**, so by using this we can't **call** any **members** of that **class**.

Instance of Class: this is a **copy** of **class** which is **initialized** by using the **new** keyword and by using this we can **call** **members** of that **class**.

Reference of Class: this is a **copy** of **class** which is **initialized** by using any **existing instance** of that class and this works **same** as an **instance**. By using the **reference** also, we can **call** **members** of that **class**.

What happens internally when we create the instance of a class?

Ans: When we **create** the **instance** of any **class** internally following **actions** will take place:

1. Reads the classes to identify their members.
 2. Invokes the constructors of all those classes.
 3. Allocates the memory that is required for execution.
-

Constructor

This is a **special method** present under a **class** responsible for **initializing** the **data members** (**fields**) of that **class**. This method is invoked **automatically** when we **create** the **instance** of **class**. The **name** of **constructor** method is the **same name** of the **class** and more over it's a **non-value** returning method. Every **class** requires a **constructor** in it, if we want to **create** the **instance** of that **class** or else, we can't **create** the **instance** of that **class**.

Note: While defining a **class** it's the **responsibility** of **developers** to define a **constructor explicitly** under his **class**, and if they **fail** to do so, **on-behalf** of the **developer** an **implicit constructor** gets defined in those **classes**; so, till now we are creating **instances** of the **classes** we defined, by using those **implicit constructors** only.

For example, if we define a class as below:

```
class Test
{
    int i = 10; string s; bool b;           //Fields
}
```

After compilation of the above class, it will be as below with an implicit constructor:

```
class Test
{
    int i = 10; string s; bool b;           //Fields
    public Test()                         //Implicit Constructor
    {
        i = 10; s = null; b = false;
    }
}
```

- Implicit constructors are **public**.
- While declaring a **field** if we assign any value to it, then **constructor** will initialize the field with that value only or else it will initialize the field with default value of that type.
- We can also define our own constructors in classes and if we do that **implicit constructor** will not be defined.

Syntax to define a Constructor Explicitly:

```
[<modifiers>] <Class_Name>( [<Parameter List>] )
```

```
{
```

-Statements to execute
}

To test defining a **constructor explicitly**, add a new class in the project naming it as “**ConDemo.cs**” and write the below code in it:

```
internal class ConDemo
{
    public ConDemo() //Explicit Constructor
    {
        Console.WriteLine("Constructor is called.");
    }
    public void Demo() //Method
    {
        Console.WriteLine("Method is called.");
    }
    static void Main()
    {
        ConDemo cd1 = new ConDemo();
        ConDemo cd2 = new ConDemo();
        ConDemo cd3 = cd2;
        cd1.Demo();
        cd2.Demo();
        cd3.Demo();
        Console.ReadLine();
    }
}
```

Constructor of a class must be **explicitly called** for execution and we do that while **creating the instance** of class as following:

Syntax: <Class_Name> <Instance_Name> = new <Constructor_Name>([<List of Values>])

Example: ConDemo obj = new **ConDemo()**;

Calling the constructor

If **constructor** is **called** then on **memory allocation** is performed, so **instances** of a class will have memory allocation because they **call** the **constructor** explicitly whereas **reference** of class will not have memory allocation because they do not **call** the **constructor**.

Constructors are defined implicitly or explicitly?

Ans: Either or.

Constructors must be called explicitly or called implicitly?

Ans: Must be called explicitly.

Constructors are of 2 types:

1. Default or Parameter-less
2. Parameterized

Constructors can also be parameterized i.e., just like a method can be defined with parameters; constructor can also be defined with parameters. If constructor is defined with parameters, we call it as “Parameterized Constructor” whereas a constructor without any parameters is called as “Default/Parameter-less Constructor”.

Default constructors can be defined either explicitly or will be defined implicitly provided there is no explicit constructor defined under that class, whereas implicit constructors will never be parameterized i.e., if a constructor is parameterized then it is very sure, that it is an explicit constructor.

Note: if Constructors of a class are parameterized then values to those parameters should be sent while creating instance of that class.

To test Parameterized Constructors, add a new class in our project naming it as “ParamConDemo.cs” and write the below code in it:

```
internal class ParamConDemo
{
    public ParamConDemo(int i)
    {
        Console.WriteLine($"Parameterized constructor is called: {i}");
    }
    static void Main()
    {
        ParamConDemo cd1 = new ParamConDemo(100);
        ParamConDemo cd2 = new ParamConDemo(200);
        ParamConDemo cd3 = new ParamConDemo(300);
        Console.ReadLine();
    }
}
```

Why to define a constructor explicitly in our class when there are implicit constructors?

Ans: We define constructors explicitly in our class for various reasons like:

1. Implicit constructors are parameter-less which will initialize fields of a class either with a default value of that type or a fixed given value, even if we create multiple instances of class, whereas if constructors are defined explicitly (parameterized), then we get a chance of passing new values to the fields every time the instance of class is created. To test this, add a new class under our project naming it as “Second.cs” and write the below code in it:

```
internal class Second
{
    public int x;           //Field
    public Second(int x)   //Variable
    {
        this.x = x;
    }
}
```

```
}
```

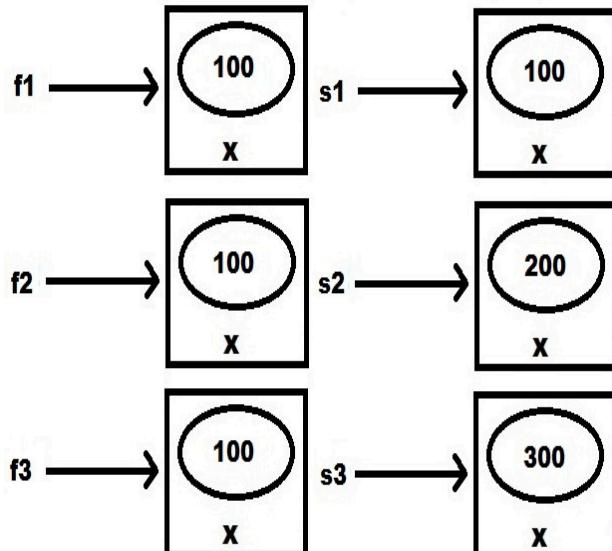
Note: “this” is a keyword which refers to the class and by using this we can access non-static members of a class from other non-static blocks when there is a **naming conflict**.

Earlier we have defined a class “First” with a public field “x” in it, and we have initialized it with a static value “100”, and in the above class also we have a public field “x” which was initialized thru a Constructor, so in the 1st case even if we create multiple instances of class First; under every instance the value of “x” will be “100” only whereas in case of class Second for each instance of class we create we can pass a new value for initialization because initialization is performed thru the **constructor**.

To test that add a new class in our project naming it as “TestClasses.cs”, and write the below code in it:

```
internal class TestClasses
{
    static void Main()
    {
        First f1 = new First();
        First f2 = new First();
        First f3 = new First();
        Console.WriteLine(f1.x + " " + f2.x + " " + f3.x);

        Second s1 = new Second(100);
        Second s2 = new Second(200);
        Second s3 = new Second(300);
        Console.WriteLine(s1.x + " " + s2.x + " " + s3.x);
        Console.ReadLine();
    }
}
```



2. Every class **requires** some values for **execution** and the values that are **required** for a class to **execute** should be **passed** to the class with the help of a **constructor**.
3. Just like **parameters** of a method will make a **method dynamic**, same as that **parameters** of **constructor** will make the whole class **dynamic**.

Static Modifier

It is a keyword using which we can declare a class and its members as static i.e., if static keyword is pre-fixed before a class or its members then they will become static or else by default every class and its member are non-static only.

Members of a class are divided into 2 categories, like:

1. Non-Static or Instance Members
2. Static Members

Members that require instance of a class for initialization and execution are known as **non-static** or **instance members**, whereas members that doesn't require instance of the class for **initialization** and **execution** are known as **static** members.

Non-Static Fields Vs Static Fields:

- ❖ If a field is explicitly declared by using static modifier it is a **static field**, whereas rest of every other **field** is **non-static** only.

```
class Test
{
    int x = 100;           //Non-Static
    static int y = 200;     //Static
    static void Main()
    {
        int z = 300;       //Static
    }
}
```

Note: variables declared under **static blocks** are also **static**.

- ❖ **Static** fields of a class are initialized **immediately** once the **execution** of that class starts whereas **non-static** fields are initialized only after **creating** the **instance** of that class as well as each and every time a **new instance** is created.
- ❖ In the **life cycle** of a class a **static** field gets initialized **1 & only 1** time whereas a **non-static** field gets initialized for "**0**" times if **no instances** are created & "**n**" times if "**n**" **instances** are created.
- ❖ The initialization of **non-static** fields is associated with a **constructor** call, so the best place to **initialize** **non-static** fields is a **constructor**.

Note: **static** fields can also be **initialized** thru **constructor** but still we never do that because, it's a **single copy** thru out the **life cycle** of a class and every new **instance** will override the **old values**.

Constant Fields: If a field is explicitly declared by using "const" keyword we call it as a constant field and those constant field can't be modified once after their declaration, so it is must to initialize them at the time of declaration only because they do not have a default value.

E.g.: **const float pi = 3.14f;**

The behavior of a constant field will be very similar to the behavior of a static field i.e., initialized immediately once the execution of class starts maintaining a single copy thru-out the life cycle of a class and the only difference between static and constant fields is static fields can be modified but not constant fields.

ReadOnly Fields: If a field is explicitly declared by using readonly keyword we call it as a readonly field and like constant fields, readonly fields also can't be modified, but after their initialization i.e., it's not mandatory to initialize readonly fields at the time of declaration because they can also be initialized after their declaration i.e., under a constructor.

E.g.: **readonly bool flag; //Declaration**

- ❖ The behavior of readonly fields will be like the behavior of non-static fields i.e., they are initialized only after creating the instance of class and maintains a separate copy for each instance that is created.
- ❖ The only difference between non-static and readonly fields is non-static fields can be modified but not readonly fields.
- ❖ The difference between constant and readonly fields is constant is a single fixed value for the whole class whereas readonly is a fixed value specific to each instance of the class.

To test all the above add a new class in our project naming it as “Fields.cs” and write the below code in it:

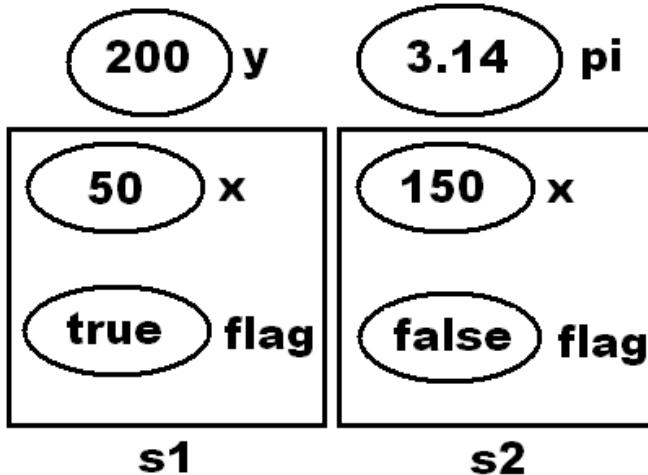
internal class Fields

```
{  
    int x;  
    static int y = 200;  
    const float pi = 3.14f;  
    readonly bool flag;  
    public Fields(int x, bool flag)  
    {  
        this.x = x;  
        this.flag = flag;  
    }  
    static void Main()  
    {  
        Console.WriteLine("Static field y is: " + y);  
        Console.WriteLine("Constant field pi is: " + pi);  
        y = 500; //Can be modified  
        //pi = 5.67f; //Can't be modified & error if un-commented  
        Console.WriteLine("Modified static field y is: " + y);  
        Console.WriteLine("-----");  
        //Creating instances of the class  
        Fields s1 = new Fields(50, true);  
        Fields s2 = new Fields(150, false);  
        Console.WriteLine("Non-Static Fields: " + (s1.x + " " + s2.x));  
        Console.WriteLine("ReadOnly Fields: " + (s1.flag + " " + s2.flag));  
        s1.x = 100; //Can be modified  
        s2.x = 300; //Can be modified  
        //s1.flag = false; //Can't be modified & Error if un-commented  
        //s2.flag = true; //Can't be modified & Error if un-commented  
        Console.WriteLine("Modified Non-Static Fields: " + (s1.x + " " + s2.x));
```

```

        Console.ReadLine();
    }
}

```



Note: While accessing fields of a class from other classes use class name for accessing static and constant fields whereas use instance of class for accessing non-static and readonly fields.

- Static field initializes immediately once the execution of class starts maintaining a single copy thru out the life cycle of class and its value is modifiable.
- Constant field also initializes immediately once the execution of class starts maintaining a single copy thru out the life cycle of class and its value is non-modifiable.
- Non-static field initializes only after creating the instance of class, as well as for each instance of the class that is created, maintaining a separate copy for each instance and its value is modifiable.
- Readonly field also initializes only after creating the instance of class, as well as for each instance of the class that is created, maintaining a separate copy for each instance and its value is non-modifiable.

Non-Static Methods Vs Static Methods:

If a method is explicitly declared by using **static** keyword, then it is a **static** method whereas rest of every other method is **non-static** only.

While defining methods, if a method is **non-static** and if we want to consume any **static** members of class in it, we can consume them directly whereas if the method is **static**, we can consume **non-static** members of class in that method only by using class **instance**.

Rules for consuming members within a class:

Static Member => Static Block	//Direct Access
Static Member => Non-Static Block	//Direct Access
Non-Static Member => Non-Static Block	//Direct Access
Non-Static Member => Static Block	//Can be accessed only by using the class instance

Rules for consuming members out of the class:

Static Members	//Using class name
Non-Static Members	//Using class instance

To test all the above add a new “Code File” in the project naming it as “**TestMethods.cs**” and write the below code in it:

```
namespace OOPSProject
{
    internal class Methods
    {
        int x = 200;
        static int y = 100;
        public void Add()
        {
            Console.WriteLine(x + y);
        }
        public static void Sub()
        {
            Methods m = new Methods();
            Console.WriteLine(m.x - y);
        }
    }
    internal class TestMethods
    {
        static void Main()
        {
            Methods obj = new Methods();
            obj.Add(); //Add is non-static so calling it with instance
            Methods.Sub(); //Sub is static so calling it with class name
            Console.ReadLine();
        }
    }
}
```

Non-Static Constructor Vs Static Constructor:

- A Constructor if explicitly declared by using static modifier is a static Constructor whereas rest of the other are non-static only and till now every Constructor, we defined is non-static only.
- Static Constructors are implicitly called whereas non-static Constructors must be explicitly called.
- As we are aware that Constructors are responsible for initializing fields in a class; Non-Static Constructor will initialize Non-Static and Readonly Fields, whereas Static Constructor will initialize Static and Constant fields.
- Static Constructor executes immediately once the execution of class starts and more over it is the first block of code to execute in a class, whereas Non-Static Constructor gets executed only after creating the instance of class as well as each and every time a new instance is created i.e., Static Constructor executes 1 and only 1

time in the life cycle of a class whereas Non-Static Constructor get executed for “0” times if no instances are created and “n” times if “n” instances are created.

- Static Constructor can't be parameterized because they are implicitly called and more over it's the first block of code to execute in a class, so we don't have any chance of sending values to its parameter's whereas parameterized Non-Static Constructors can be defined.

Note: We have already learnt earlier that, every class will contain an **implicit** constructor if not defined **explicitly** and those **implicit** constructors are defined based on the following criteria:

1. Non-static constructor will be defined in every class except in a static class.
2. Static constructor will be defined only if the class contains any static fields.

```
class Test           //Case 1
{
}
```

*After compilation there will be a non-static constructor in class.

```
class Test           //Case 2
{
    int i = 10;
}
```

*After compilation there will be a non-static constructor in class.

```
class Test           //Case 3
{
    static int i = 100;
}
```

*After compilation there will be both static and non-static constructors also.

```
static class Test     //Case 4
{}
```

*After compilation there will not be any constructor in class.

```
static class Test     //Case 5
{
    static int i = 100;
}
```

*After compilation there will be a static constructor in class.

To test all the above add a new class in the project naming it as “Constructors.cs” and write the below code in it:

internal class Constructors

```
{
    static Constructors()
    {
        Console.WriteLine("Static constructor is called.");
    }
    Constructors()
```

```

{
    Console.WriteLine("Non-static constructor is called.");
}
static void Main()
{
    Console.WriteLine("Main method is called.");
    Constructors c1 = new Constructors();
    Constructors c2 = new Constructors();
    Constructors c3 = new Constructors();
    Console.ReadLine();
}
}

```

Static Class: These are introduced in C# 2.0. If a class is explicitly declared by using static modifier, we call it as a static class and this class can contain only static members in it. We can't create the instance of static class and moreover it is not required also.

```

static class Class1
{
    //Define only static members.
}

```

Note: Console is a static class in our Libraries so every member of Console class is a static member only and to check that, right click on Console class in Visual Studio and choose the option “Go to definition” which will open “Metadata” or “Source Code” of that class.

Entity

Any living or non-living object that is associated with a set of attributes is known as an entity and application development is all about dealing and managing these entities only. To develop an application, we follow the below process:

Step 1: Identify each entity that is associated with the application.

- School Application: Student, Teacher, Book
- Retail Business Application: Customer, Employee, Product, Supplier

Step 2: Identify each attribute of that entity.

- Student: Id, Name, Address, Phone, Class, Section, Fees, Marks, Grade
- Teacher: Id, Name, Address, Phone, Qualification, Subject, Salary, Designation
- Customer: Id, Name, Address, Phone, Balance, Account Type, EmailId, PanCard, Aadhar
- Employee: Id, Name, Address, Phone, Job, Salary, Department, EmailId, PanCard

Step 3: Design a Database based on the following guidelines:

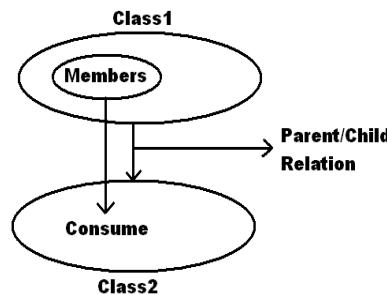
- Create a table representing each entity.
- Every column of the table should represent each attribute of the entity.
- Each record under table should be a unique representation for the entity.

Step 4: Design an application by using any **Programming Language** of your choice which should act as an **UI (User Interface)** between the **End User** and **Database** in managing the data present under **Database**, by adopting following **guidelines**:

- Define a class where each class should represent an entity.
- Define properties where each property should be a representation for each attribute.
- Each instance of the class we create will be a unique representation for each entity.

Inheritance

It is a process of consuming members that are defined in one class from other classes by establishing **parent/child** relationship between the classes, so that **child class** can consume members of its **parent class** as if they are **owner** of those members.



Note: Child class even if it can **consume** members of its parent class as an **owner**, still it can't access **private** members of their **parent** like **Constructors** and **Finalizers**

Syntax:

[<modifiers>] class <CC Name> : <PC Name>

Example:

```

class Class1
{
    -Define Members
}

class Class2 : Class1
{
    -Consume members of parent i.e., Class1 from here
}
  
```

To test inheritance, add a new class under the project naming it as “Class1.cs” and write the below code in it:

```

internal class Class1
{
    public Class1()
    {
        Console.WriteLine("Class1 constructor is called.");
    }
    public void Test1()
    {
        Console.WriteLine("Method 1");
    }
    public void Test2()
    {
        Console.WriteLine("Method 2");
    }
}

```

Now add another class in the project naming it as “Class2.cs” and write the below code in it:

```

internal class Class2 : Class1
{
    public Class2()
    {
        Console.WriteLine("Class2 constructor is called.");
    }
    public void Test3()
    {
        Console.WriteLine("Method 3");
    }
    public void Test4()
    {
        Console.WriteLine("Method 4");
    }
    static void Main()
    {
        Class2 c = new Class2();
        c.Test1(); c.Test2();      //Calling members of parent class
        c.Test3(); c.Test4();      //Calling members of current class
        Console.ReadLine();
    }
}

```

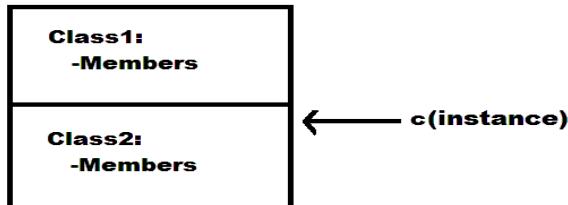
Rules and regulations that has to be followed while working with Inheritance:

Rule 1: In inheritance parent class **Constructor** must be **accessible** to child class or else **inheritance** will **not** be possible. The reason why parent's **Constructor** should be **accessible** to child is, because whenever **child** class instance is created, control first jumps to child class **Constructor** and child class **Constructor** will in turn call its parent class **Constructor** for execution and to test this, place a break point at child class's **Main** method and debug the code by hitting **F11**.

The reason why child Constructor calls its parent class Constructor is, because if child class wants to consume members of its parent class, they must be **initialized** first and then only child classes can consume them and we are already aware that members of a class are initialized by its own **Constructor**.

Note: Constructors are **never** inherited i.e., Constructors are **specific** to any class which can initialize members of that particular class only but not of parent or child classes.

When we **create** the **instance** of any class, it will first read all its parent classes to gather the information of members that are present under those classes, so in our previous case when the instance of **Class2** is created it gathers information of **Class1** also as following:



Rule 2: In inheritance child class can **access** members of their parent class whereas parent classes **can never access** members of their child class which are **purely defined** under the child class. To test this, re-write the code under **Main** method of child class i.e., **Class2** as following:

```

Class1 p = new Class1();
p.Test1(); p.Test2();           //Valid
//p.Test3(); p.Test4();         //Invalid and in-accessible
Console.ReadLine();
  
```

Rule 3: Earlier we have learnt that variable of a class can be initialized by using instance of same class to make it as a reference, for example:

```

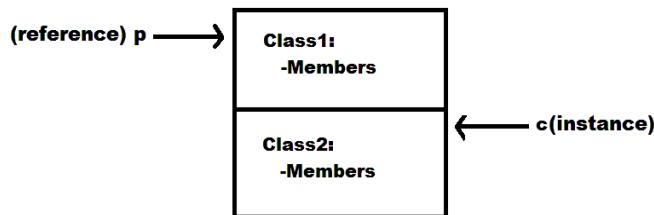
Class2 c1 = new Class2();
Class2 c2 = c1;
  
```

Same as the above we can also initialize variables of parent class by using its child classes instance as following:

```

Class2 c = new Class2();
Class1 p = c;
  
```

In this case both parent class reference and child class instance will be accessing the same memory, but owner of that memory is child class instance.



In the above case even if parent class reference is initialized by the child class instance and consuming the memory of child class instance, now also it is not possible to access the member's which are **purely defined** under the child class and to test that rewrite the code under Main method of child class i.e., **Class2** as following:

```

Class2 c = new Class2();
Class1 p = c;
p.Test1(); p.Test2();      //Valid
//p.Test3(); p.Test4();    //Invalid and in-accessible now also
Console.ReadLine();

```



Note: We can never initialize child class variables by using parent class instance either **implicitly** or **explicitly** also.

```

Class1 p = new Class1(); //Creating parent class instance
Class2 c = p;           //Invalid (Implicit conversion and compile time error)
Class2 c = (Class2)p;   //Invalid (Explicit conversion and runtime error)

```

We can **initialize** child class variables by using a parent class **reference** which is **initialized** by using the same child class instance by performing an **explicit** conversion.

Creating parent's reference by using child class instance:

```

Class2 c = new Class2();
Class1 p = c;

```

Initializing child's variable by using the above parent's reference:

```

Class2 obj = (Class2)p;           //Valid (Explicit)
Or
Class2 obj = p as Class2;        //Valid (Explicit)

```

Child Instance	=> Parent Reference	//Valid
Child Instance	=> Parent Reference	=> Child Reference //Valid
Parent Instance	=> Child Reference	//Invalid

Note: in the above case the new reference “**obj**” also starts accessing the same memory allocated for the instance “**c**” and with the new reference we call the members of both “**Class1**” and “**Class2**” also.



Rule 4: Every class that is **pre-defined** or **user-defined** has a **default** parent class i.e., **Object** class of **System** namespace. **Object** is the **ultimate parent** of all classes in **.NET** class hierarchy providing **low level services** to child classes. So, every class by default contains 4 methods that are inherited from the “**Object**” Class and those are

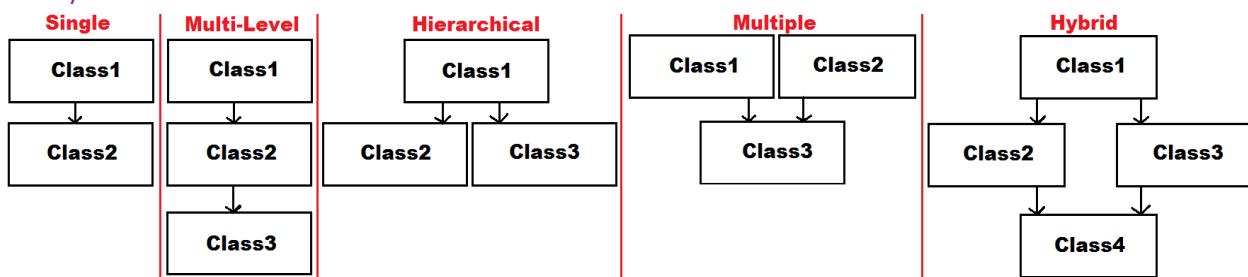
“`Equals`”, “`GetHashCode`”, “`GetType`” and “`ToString`”, and these 4 methods can be called or consumed from any class. To test this, re-write code under `Main` method of child class (`Class2`) as following:

```
Object obj = new Object();
Console.WriteLine(obj.GetType() + "\n");
Class1 p = new Class1();
Console.WriteLine(p.GetType() + "\n");
Class2 c = new Class2();
Console.WriteLine(c.GetType());
Console.ReadLine();
```

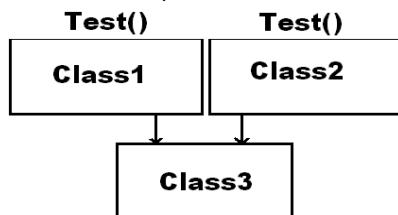
Types of Inheritance: This talks about no. of child classes a parent has or the no. of parent classes a child has.

According to the standards of **Object-Oriented Programming** we have **5** types of inheritances, and they are:

- i. Single
- ii. Multi-Level
- iii. Hierarchical
- iv. Multiple
- v. Hybrid



Rule 5: Both **Java** and **.NET Language's** doesn't provide the support for **Multiple** and **Hybrid** inheritances thru **classes**, and what they support is **Single**, **Multi-Level** and **Hierarchical** inheritances only because **Multiple Inheritance** suffers from **ambiguity** problem, for example:



Note: **C++ Language** supports all 5 types of **Inheritances** because it is the 1st **Object Oriented Programming Language** that came into **existence** and at the time of its **introduction**, this problem was not **anticipated**.

Rule 6: In the first rule of **inheritance**, we have discussed that whenever the **instance** of child class is created it will **implicitly** call its parent class constructor for execution, but this **implicit** calling will take place only if parent classes **Constructor** is “**default or parameter less**”, whereas if at all the parent classes **Constructor** is **parameterized** then child class **Constructor** can't implicitly call parent class **Constructor** for execution because it requires **parameter values**. To resolve the above problem developer needs to explicitly call parent classes **Constructor** from child class **Constructor** by using “**base**” keyword and pass all the required parameter values.

To test the above, re-write constructor of parent class i.e., Class1 as following:

```
public Class1(int i)
{
    Console.WriteLine("Class1 constructor is called: " + i);
}
```

Now when we run child class i.e., **Class2**, we get an error stating that there is no value sent to **formal parameter "i"** of **Class1** (Parent Class) and to resolve this problem re-write constructor of **Class2** as following:

```
public Class2(int x) : base(x)
{
    Console.WriteLine("Class2 constructor is called.");
}
```

In the above case child classes constructor is also **parameterized** so while creating the instance of child class we need to explicitly pass all the required values to its constructor and those values are first loaded into the constructor and from there those values are passed to parent classes constructor thru the "**base**" keyword, and to test this go to **Main** method of **Class2**, and re-write the code in it as below and **debug**:

```
Class2 c = new Class2(50);
```

How do we use inheritance in application development?

Ans: Inheritance is a process which comes into picture from the **initial stages** of an application development. As discussed earlier, if we want to develop an **application**, we need to follow the below process:

Step 1: Identification of the Entities.

E.g.: School Application: **Student, Teaching Staff, Non-Teaching Staff**

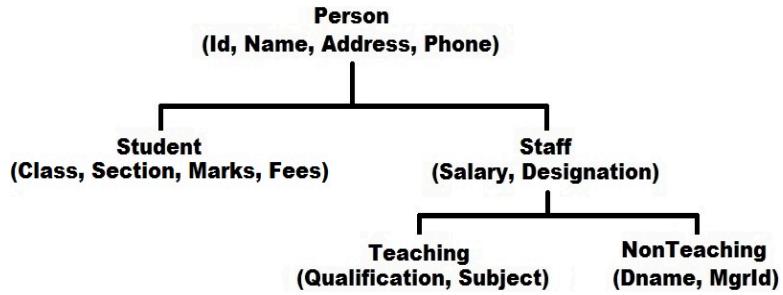
Step 2: Identification of Attributes for each Entity.

<u>Student</u>	<u>Teaching Staff</u>	<u>Non-Teaching Staff</u>
Id	Id	Id
Name	Name	Name
Phone	Phone	Phone
Address	Address	Address
Class	Designation	Designation
Section	Salary	Salary
Marks	Qualification	Dname
Fees	Subject	MgrId

Step 3: Designing the Database.

Step 4: Developing an **application** that works like an **UI**. While developing the **application**, to bring **re-usability** into the applications we use **inheritance** and to do that follow the below guidelines:

1. Identify all the **common attributes** between **entities** and put them in a **hierarchical order** as below:



2. Now define **classes** based on the above **hierarchy**:

```

public class Person
{
    public int Id;
    public string Name, Phone, Address;
}

public class Student : Person
{
    int Class;
    char Section;
    float Marks, Fees;
}

public class Staff : Person
{
    public double Salary;
    public string Designation;
}

public class Teaching : Staff
{
    string Subject, Qualification;
}

public class NonTeaching : Staff
{
    int MgrId; string Dname;
}
  
```

Polymorphism

Behaving in different ways depending upon the input received is known as **Polymorphism** i.e., whenever **input** changes then automatically the **output** or **behaviour** also changes accordingly. This can be implemented in our language in 3 different ways:

1. Overloading
2. Overriding
3. Hiding/Shadowing

Overloading: This is again of different types like **Method Overloading**, **Operator Overloading**, **Constructor Overloading**, **Indexer Overloading** and **De-constructor Overloading**.

Method Overloading: It is an approach of defining multiple methods in a class with the **same name** by changing their **parameters**. Changing **parameters** means we can change any of the **following**:

1. Change the no. of parameters passed to method.
2. Change the type of parameters passed to method.
3. Change the order of parameters passed to method.

- public void Show()
- public void Show(int i)
- public void Show(string s)
- public void Show(int i, string s)
- public void Show(string s, int i)

Note: in overloading a **return type** change without parameter change is not taken into **consideration**, for example:

public string Show() => Invalid

To test **method overloading**, add a new class in the project naming it as “**OverloadMethods.cs**” and write the following code in it:

```
internal class OverloadMethods
{
    public void Show()
    {
        Console.WriteLine(1);
    }
    public void Show(int i)
    {
        Console.WriteLine(2);
    }
    public void Show(string s)
    {
        Console.WriteLine(3);
    }
    public void Show(int i, string s)
    {
        Console.WriteLine(4);
    }
    public void Show(string s, int i)
    {
        Console.WriteLine(5);
    }
    static void Main()
    {
        OverloadMethods obj = new OverloadMethods();
        obj.Show();
        obj.Show(10);
        obj.Show("Hello");
        obj.Show(10, "Hello");
        obj.Show("Hello", 10);
        Console.ReadLine();
    }
}
```

What is Method Overloading?

Ans: It's an approach of defining a **method** with **multiple behaviors** and those **behaviors** will vary based on the **number, type and order of parameters**. For example, **IndexOf** is an overloaded **method** under **String** class which returns the index position of a **character** or **string** based on the **input** values of that method, for example:

```
string str = "Hello World";
str.IndexOf('o'); => 4 => Returns the first occurrence of a character
str.IndexOf('o', 5); => 7 => Returns the next occurrence of a character
```

Note: `WriteLine` method of `Console` class is also overloaded for printing any type of value that is passed as input to the method, as following:

- `WriteLine()`
- `WriteLine(int value)`
- `WriteLine(bool value)`
- `WriteLine(double value)`
- `WriteLine(string value)`
- `WriteLine(string format, params object[] values)`
- +12 more overloads

Inheritance based overloading: It's an approach of **overloading** parent classes' **methods** under the child class, and to do this child class doesn't require **taking any permission** from the parent class, for example:

```
Class1  
public void Test()
```

```
Class2 : Class1  
public void Test(int i)
```

Method Overriding: it's an approach of **re-implementing** parent classes' methods under child class exactly with the same **name** and **signature (parameters)**.

Difference between Method Overloading and Method Overriding:

Method Overloading	Method Overriding
It's all about defining multiple methods with the same name by changing their parameters.	It's all about defining multiple methods with the same name and same parameters.
This can be performed with-in a class or between parent-child classes also.	This can be performed only between parent-child classes but can't be performed with-in a class.
To overload parent's method under child, child doesn't require any permission from parent.	To override parent's method under child, parent should first grant the permission to child.
This is all about defining multiple behaviours to a method.	This is all about changing existing behaviour of a parent's method under child.

How to override a parent classes method under child class?

Ans: To override any parent classes' method under child class, first that method should be declared "**overridable**" by using "**virtual**" modifier in parent class as following:

Class1 =>
`public virtual void Show() //Overridable`

Every **virtual** method of parent class can be **overridden** by child class, if required by using "**override**" modifier as following:

Class2 : Class1 =>
`public override void Show() //Overriding`

Note: overriding **virtual** methods of parent class under child class is **not mandatory** for child class.

In **overriding**, parent class defines a method in it as **virtual** and gives it to the child class for **consumption**, so that it's giving a permission to the child class either to consume the method "**as is**" or **override** the method as per its requirement, if at all the original behavior of that method is not **satisfactory** to the child class.

To test **inheritance-based method overloading** and **method overriding**, add a new class in the project naming it as "**LoadParent.cs**" and write the following code in it:

```
internal class LoadParent
{
    public void Test()
    {
        Console.WriteLine("Parent Class Test Method Is Called.");
    }
    public virtual void Show() //Overridable
    {
        Console.WriteLine("Parent Class Show Method Is Called.");
    }
    public void Display()
    {
        Console.WriteLine("Parent Class Display Method Is Called.");
    }
}
```

Now add another class in the project naming it as "LoadChild.cs" and write the following code in it:

```
internal class LoadChild : LoadParent
{
    //Overloading parent's Test method in child
    public void Test(int i)
    {
        Console.WriteLine("Child Class Test Method Is Called.");
    }
    static void Main()
    {
        LoadChild c = new LoadChild();
        c.Test();      //Executes parent class Test method
        c.Test(10);   //Executes child class Test method
        c.Show();     //Executes parent class Show method
        c.Display();  //Executes parent class Display method
        Console.ReadLine();
    }
}
```

Inheritance-Based Overloading: In the above classes **Test** method of parent class has been **overloaded** in child class and then by using child class instance we are able to call both parent and child classes methods also, from the child class.

Method Overriding: In the above classes `Show` method of parent class is declared `virtual` which gives a chance for child classes to `override` that method but the child class did not `override` the method, so a call to that method by using child classes instance will invoke the parent classes `Show` method only and this proves us `overriding` is `optional` and to confirm that run the child class `LoadChild` and watch the output of `Show` method.

In this case if child class overrides the parent classes `virtual` method, then a call to that method by using child class `instance` will execute or invoke its own method but not of the parent classes, and to test that add a new method in class `LoadChild` as following:

```
//Overriding parent's Show method in child class
public override void Show()
{
    Console.WriteLine("Child Class Show Method Is Called.");
}
```

Now if we run the child class i.e., `LoadChild` and watch the output of `Show` method we will notice child classes `Show` method getting executed in place of parent classes `Show` method and this is what we call as changing the behavior.

Can we override any parent classes' methods under child classes without declaring them as virtual?

Ans: No.

Can we re-implement any parent classes' methods under the child classes without declaring them as virtual?

Ans: Yes.

We can re-implement a parent class method under the child class by using 2 different approaches:

- Overriding
- Hiding/Shadowing

Method Hiding/Shadowing: This is also an approach of `re-implementing` parent classes methods under child class exactly with the same `name` and `parameters` just like `overriding` but the difference between the 2 is; in `overriding` child class can `re-implement` only `virtual` methods of parent class where as in-case of `hiding/shadowing` child class can `re-implement` any method of the parent class i.e., even if the method is not declared as `virtual` also `re-implementation` can be performed.

```
Class1 =>
    public void Display()
```

```
Class2 : Class1 =>
    public [new] void Display()      //Hiding/Shadowing
```

In the above case using “`new`” keyword while `re-implementing` the method in child class is only `optional` and if we don’t use it, compiler gives a `warning` message at the time of `compilation`, saying that there is already a method with the same name in parent class and your new method in child class will `hide` that old method, so by using “`new`” keyword we are informing the `compiler` that we are `intentionally` defining a new method with the same `name` and `parameters` under our child class.

Before testing **hiding/shadowing** first run the child class i.e., **LoadChild** and watch the output of **Display** method and here we notice that parent classes **Display** method getting executed, now add a new method in the child class **LoadChild** as following:

```
//Hiding/Shadowing parent class Display method in child class
public new void Display()
{
    Console.WriteLine("Child Class Display Method Is Called.");
}
```

Now run the child class **LoadChild** again and watch the difference in **output** i.e., in this case child classes **Display** method is called in place of parent class **Display** method.

In the above 2 classes we have performed the following:

LoadParent

```
public void Test()
public virtual void Show()
public void Display()
```

LoadChild : LoadParent

public void Test(int i)	=> Overloading
public override void Show()	=> Overriding
public new void Display()	=> Hiding/Shadowing

In case of **Overriding** and **Hiding**, after **re-implementing** the parent classes methods under child class, instance of child class starts calling its own methods but not of parent class, whereas if required there is still a chance of calling those parent class methods from child class in 2 different ways:

1. By creating the parent classes instance under the child class, we can call parent class methods from child class and to test that re-write code under Main method of child class i.e., **LoadChild** as following:

```
LoadParent p = new LoadParent();
p.Show();                                //Executes parent class Show method
p.Display();                               //Executes parent class Display method
LoadChild c = new LoadChild();
c.Show();                                 //Executes child class Show method
c.Display();                               //Executes child class Display method
Console.ReadLine();
```

2. By using **base** keyword also, we can call parent class methods from child class, but keywords like "**this**" and "**base**" can't be used in **static blocks**.

To test this first add 2 new methods under the child class i.e., **LoadChild as following:**

```
public void PShow()
{
    base.Show();
```

```

    }

public void PDisplay()
{
    base.Display();
}

```

In the above case the 2 new methods we defined in child class, acts as an **interface** in calling parent classes methods from the child class, so now by using child class instance only we can call both parent and child classes methods also.

To test this, re-write code under Main method of child class i.e., LoadChild as following:

```

LoadChild c = new LoadChild();
c.PShow();                                //Executes parent class Show method
c.PDisplay();                               //Executes parent class Display method
c.Show();                                   //Executes child class Show method
c.Display();                                //Executes child class Display method
Console.ReadLine();

```

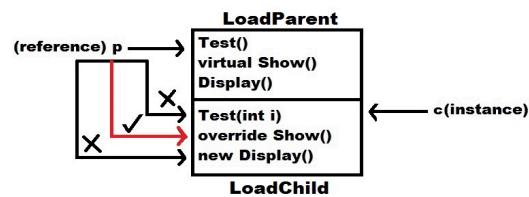
Note: Earlier in the 3rd rule of **inheritance** we have learnt that parent class **reference** even if created by using child class **instance** can't access any members of the child class which are **purely defined** under child class but we have an exemption for that rule, that is, parent's reference can call or access **overridden members** of the child class because **overridden** members are not considered as **pure child class members** because they have been **re-implemented** with permission from the parent class.

To test that re-write code under Main method of child class i.e., LoadChild as following:

```

LoadChild c = new LoadChild();
LoadParent p = c;
p.Show(); //Executes child class Show method
p.Display(); //Executes parent class Display method only
Console.ReadLine();

```



In the above case **Display** is considered as pure child class member only because it's re-implemented by child class without taking any permission from parent, so parent will never recognize it.

Polymorphism is divided into 2 types:

1. **Static or Compile-time Polymorphism (Early Binding)**
2. **Dynamic or Run-time Polymorphism (Late Binding)**

In static or compile-time polymorphism, the decision which polymorphic method must be executed for a method call is performed at compile time. Method overloading is an example for this and here compiler identifies which overloaded method it must execute for a particular method call at the time of program compilation by checking the type and number of parameters that are passed to the method and if no method matches the method call it will give an error.

In dynamic or run-time polymorphism, the decision which polymorphic method must be executed for a method call is made at runtime rather than compile time. Run-time polymorphism is achieved by method overriding because method overriding allows us to have methods in the parent and child classes with the same name and the same parameters. By runtime polymorphism, we can point to any child class by using the reference of the parent class, which is initialized by child class instance, so the determination of the method to be executed is based on the instance being referred to by reference.

Static Polymorphism	Dynamic Polymorphism
1. Occurs at compile-time.	1. Occurs at runtime.
2. Achieved through static binding.	2. Achieved through dynamic binding.
3. Method overloading should exist.	3. Method overriding should exist.
4. Inheritance is not involved.	4. Inheritance is involved.
5. Happens in the same class.	5. Happens between parent-child classes.
6. Reference creation thru instance is not required.	6. Requires parent class reference creation thru child class instance.

Operator Overloading

It's an approach of defining multiple behaviors to an operator, which varies based on the operands between which we use the operator. For example: “+” is an **addition operator** when used between **numeric operands** and it is a **concatenation operator** when used between **string operands**.

Number + Number => Addition

String + String => Concatenation

The behaviour for an **operator** is **pre-defined** i.e., developers or designers of the **language** have already implemented logic that must be executed when an **operator** is used between 2 **operands** under the **libraries** of the **language** with the help of a special method known as “**Operator Method**”.

Syntax of an Operator Method:

```
[<modifiers>] static <type> operator <opt>(<operand types>)
{
    -Logic
}
```

- Operator methods must be static only.
- <type> refers to the return type of method i.e., when the operator is used between 2 types what should be the result type.

- operator is name of the method, which should be in lower case and can't be changed.
- <opt> refers to the operator for which we want to write behaviour like “+” or “-” or “==”, etc.
- <operand types> refers to type of operands between which we want to use the operator.

Under libraries, operator methods have been defined as following:

```
public static int operator +(int a, int b)
public static int operator -(int a, int b)
public static string operator +(string a, string b)
public static string operator +(string a, int b)
public static bool operator >(int a, int b)
public static float operator +(int a, float b)
public static decimal operator +(double a, decimal b)
public static bool operator ==(string a, string b)
public static bool operator !=(string a, string b)
```

Note: same as the above we can also define **operator methods** for using an **operator** between new types of **operands**.

To test “**Operator Overloading**”, “**Method Overriding**” and “**Hiding/Shadowing**” add a new class naming it as “**Matrix.cs**” under the project and write the below code:

```
internal class Matrix
{
    //Declaring attributes for a 2 * 2 Matrix
    int a, b, c, d;
    //Initializing attributes of the Matrix in constructor
    public Matrix(int a, int b, int c, int d)
    {
        this.a = a; this.b = b; this.c = c; this.d = d;
    }
    //Overriding the ToString() method inherited from Object class to return values of the Matrix in 2 * 2 format
    public override string ToString()
    {
        return a + " " + b + "\n" + c + " " + d + "\n";
    }
    //Implementing the + operator so that it can be used between 2 Matrix operands
    public static Matrix operator +(Matrix obj1, Matrix obj2)
    {
        Matrix obj = new Matrix(obj1.a + obj2.a, obj1.b + obj2.b, obj1.c + obj2.c, obj1.d + obj2.d);
        return obj;
    }
    //Implementing the - operator so that it can be used between 2 Matrix operands
    public static Matrix operator -(Matrix obj1, Matrix obj2)
    {
        Matrix obj = new Matrix(obj1.a - obj2.a, obj1.b - obj2.b, obj1.c - obj2.c, obj1.d - obj2.d);
        return obj;
    }
}
```

```

}

//Re-Implementing the == operator using Hiding/Shadowing so that it can be used between 2 Matrix's to perform
    values equal comparison because original implementation is reference equal comparison
public static bool operator ==(Matrix obj1, Matrix obj2)
{
    if (obj1.a == obj2.a && obj1.b == obj2.b && obj1.c == obj2.c && obj1.d == obj2.d)
        return true;
    else
        return false;
}
//Re-Implementing the != operator using Hiding/Shadowing so that it can be used between 2 Matrix's to perform
    values not equal comparison because original implementation is reference not equal comparison
public static bool operator !=(Matrix obj2, Matrix obj1)
{
    if (obj1.a != obj2.a || obj1.b != obj2.b || obj1.c != obj2.c || obj1.d != obj2.d)
        return true;
    else
        return false;
}
}

```

ToString is a method defined in the parent class “Object” and by default that method returns “Name” of the type to which an **instance** belongs when we call it on any type’s instance. **ToString** method is declared as **virtual** under the class “Object” so any child class can **override** it as per their **requirements** as we performed it in our “Matrix” class to change the **behaviour** of that method, so the new method will return **values** that are associated with “Matrix” but not the **type name**.

The “==” and “!=” operators are also **implemented** in the parent class “Object”, but their original behaviour is to perform a **reference equal** and **reference not-equal** comparison between **type instances** but not **values equal** and **values non-equal** comparison. We can also change the **behaviour** of those **operator methods** by using the concept of **hiding** (but not **overriding** because they are not declared as **virtual**) as we have done in our **Matrix** class, so that now the 2 **operators** will now perform **values equal** and **values not-equal** comparison in place of **reference equal** and **reference not-equal** comparison.

To consume all the above add a new class TestMatrix.cs and write the below code:

```

internal class TestMatrix
{
    static void Main()
    {
        //Creating 4 instances of Matrix class with different values
        Matrix m1 = new Matrix(20, 19, 18, 17);
        Matrix m2 = new Matrix(15, 14, 13, 12);
        Matrix m3 = new Matrix(10, 9, 8, 7);
        Matrix m4 = new Matrix(5, 4, 3, 2);
        //Performing Matrix Arithmetic
        Matrix m5 = m1 + m2 + m3 + m4;
        Matrix m6 = m1 - m2 - m3 - m4;
        //Printing values of each Matrix:
    }
}

```

```

Console.WriteLine(m1);
Console.WriteLine(m2);
Console.WriteLine(m3);
Console.WriteLine(m4);
Console.WriteLine(m5);
Console.WriteLine(m6);

//Performing Matrix equal comparision
if (m1 == m2)
    Console.WriteLine("Yes, m1 is equal to m2.");
else
    Console.WriteLine("No, m1 is not equal to m2.");
//Performing Matrix not equal comparision
if (m1 != m2)
    Console.WriteLine("Yes, m1 is not equal to m2.");
else
    Console.WriteLine("No, m1 is equal to m2.");
Console.ReadLine();
}
}

```

In the above case when we call `WriteLine` method by passing `Matrix` class `instance` as a `parameter` to it, will internally invoke the overloaded `WriteLine` Method which takes “`Object`” as a `parameter` and that method will internally call `ToString` method on that `instance`, and because we have `overwritten` the `ToString` method in our `Matrix` class, a call to it in `WriteLine` method will invoke `Matrix` classes `ToString` method which returns the values that are associated with `Matrix` instance and prints them in a $2 * 2$ `Matrix` format (`Dynamic Polymorphism`).

Constructor Overloading

Just like `methods` in a class can be `overloaded`, `constructors` in a class also can be `overloaded` and it is called as “`Constructor Overloading`”. It’s an approach of defining `multiple constructors` under a class and if `constructors` of a class are `overloaded` then instance of that class can be created by using any available `constructor` i.e., it is not `mandatory` to call any `particular` constructor for `instance` creation. To test this, add a new code file under the project naming it as “`TestOverloadCons.cs`” and write the below code in it:

```

namespace OOPSProject
{
    internal class OverloadCons
    {
        int i; bool b;
        public OverloadCons()
        {
            //Initializes i & b with default values
        }
        public OverloadCons(int i)
        {
            //Initializes b with default value and i with given value
            this.i = i;
        }
    }
}

```

```

}

public OverloadCons(bool b)
{
    //Initializes i with default value and b with given value
    this.b = b;
}

public OverloadCons(int i, bool b)
{
    //Initializes both i & b with given values
    this.i = i;
    this.b = b;
}

public void Display()
{
    Console.WriteLine($"Value of i is: {i} and value of b is: {b}");
}

internal class TestOverloadCons
{
    static void Main()
    {
        OverloadCons c1 = new OverloadCons();
        c1.Display();

        OverloadCons c2 = new OverloadCons(10);
        c2.Display();
        OverloadCons c3 = new OverloadCons(true);
        c3.Display();

        OverloadCons c4 = new OverloadCons(10, true);
        c4.Display();
        Console.ReadLine();
    }
}
}

```

By overloading constructors in a class, we get a chance to initialize fields of that class in 3 different ways:

1. With a **default** or **parameter-less** constructor defined in class we can **initialize** all **fields** of that class with **default values**.
2. With a **parameterized** constructor defined in class we can **initialize** all **fields** of that class with **given values**.
3. With a **parameterized** constructor defined in class we can **initialize** some **fields** of that class with **default values** and some **fields** with **given values**.

Note: If a class contains **multiple attributes** in it and if we want to **initialize** them in a “**mix & match**” combination then we **overload** constructors, and the no. of **constructors** to be defined will be **2 power “n”** where “**n**” is the no. of attributes. In our above class we have **2** attributes, so we have defined **4** constructors.

Copy Constructor

It is a **constructor** using which we can create a new **instance** of the class with the help of an **existing instance** of the **same class**, which **copies** the **attribute** values from the **existing instance** into the **new instance** and the main purpose of this constructor is to **initialize a new instance** with the values from an **existing instance**. The “**Formal Parameter Type**” of a copy constructor will be the same “**Class Type**” in which it is defined.

To test Copy Constructors, add a new class under the project naming it as “**CopyConDemo.cs**” and write the following code:

```
internal class CopyConDemo
{
    int Id;
    string Name;
    double Balance;
    public CopyConDemo(int Id)
    {
        this.Id = Id;
        Name = "Vijay";
        Balance = 5000.00;
    }
    public CopyConDemo(CopyConDemo cd)
    {
        this.Id = cd.Id;
        this.Name = cd.Name;
        this.Balance = cd.Balance;
    }
    public void Display()
    {
        Console.WriteLine($"Id: {Id}; Name: {Name}; Balance: {Balance}");
    }
    static void Main()
    {
        CopyConDemo cd1 = new CopyConDemo(1005);
        cd1.Display();
        CopyConDemo cd2 = new CopyConDemo(cd1);
        cd2.Display();
        Console.WriteLine();
        cd1.Balance = 10000;
        cd1.Display();
        cd2.Display();
        Console.WriteLine();
        cd2.Balance = 20000;
        cd1.Display();
        cd2.Display();
        Console.ReadLine();
    }
}
```

}

In the above case “cd2” is a new **instance** of the class which is created by **copying** the values from “cd1” and here any **changes** that are **performed** on members of “cd1” will not **reflect** to members of “cd2” and **vice versa** because they have their own individual memory which is not accessible to others.

Types of Constructors: constructors are divided into **5 Categories** like:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

Default Constructor: a constructor defined without any parameters is known as a default constructor, which will initialize fields of a class with default values. If a class is not defined with any explicit constructor, then the class will contain an implicit default constructor.

Parameterized Constructor: if a constructor is defined with at least 1 parameter then we call it as parameterized constructor and these constructors must be explicitly defined but never implicitly defined. Parameterized constructors are used for initializing fields of a class with given set of values which we can pass while creating the instance of that class.

Copy Constructor: it's a constructor which takes the same type as its **“Parameter”** and initializes the fields of class by copying values from an existing instance of the same class. A Copy constructors will not create a reference to the class i.e., it will create a new instance for the class by allocating memory for all the members of that class and very importantly any changes made on the source will not reflect to the new instance and vice-versa.

Static Constructor: if a constructor is defined explicitly by using static modifier, we call it as a static constructor and this constructor is the first block of code which executes under the class, and they are responsible for initializing static fields and more over these constructors can't be overloaded because they can't be parameterized. This constructor is called implicitly before the first instance is created or any static members are referenced.

Private Constructor: If a constructor is explicitly declared by used private modifier, we call it as a private constructor. If a class contains only private constructors and no public constructors, other classes cannot create instances of that class. The use of private constructor is to serve singleton classes where a singleton class is one which limits the number of instances created to one.

Sealed Class: if a class is explicitly declared by using sealed modifier, we call it as a sealed class and these classes can't be inherited by other classes, for example:

```
sealed class Class1
{
    -Members
}
```

In the above case Class1 is a sealed class so it can't be inherited by any other class, for example:

class Class2 : Class1 => In-valid

Note: even if a sealed class can't be **inherited** it is still possible to consume the members of a **sealed** class by creating its **instance**, for example **String** is a sealed class in our **libraries**, so we can't inherit from that class but we can still **consume** it in all our classes by creating the **instance** of **String** class.

Sealed Method: If a parent class method can't be **overridden** under a child class, then we call that method as **sealed** method. By default, every method of a class is **sealed** because we can never **override** any method of parent class under the child class unless the method is declared as **virtual**. If a method is declared as **virtual** under a class, then any child class of it in a **linear hierarchy** (**Multi-Level Inheritance**) can override that method, for example:

```
Class1
public virtual void Show()

Class2 : Class1
public override void Show()           //Valid

Class3 : Class2
public override void Show()           //Valid
```

Note: in the above case even if **Class2** is not **overriding** the method also **Class3** can **override** the method.

When a **child** class is **overriding** parent classes' **virtual** methods, it can **seal** those methods by using **sealed** modifier on them, so that **further overriding** of those methods can't be performed by its child classes, for example:

```
Class1
public virtual void Show()

Class2 : Class1
public sealed override void Show()      //Valid

Class3 : Class2
public override void Show()             //In-valid
```

Note: in the above case **Class2** has **sealed** the method while **overriding**, so **Class3** can't **override** the method.

Abstract Class and Abstract Method

Abstract Method: a method without any body is known as abstract method i.e., an abstract method contains only declaration without any implementation. To declare a method as abstract it is must to use "**abstract**" modifier on that method explicitly.

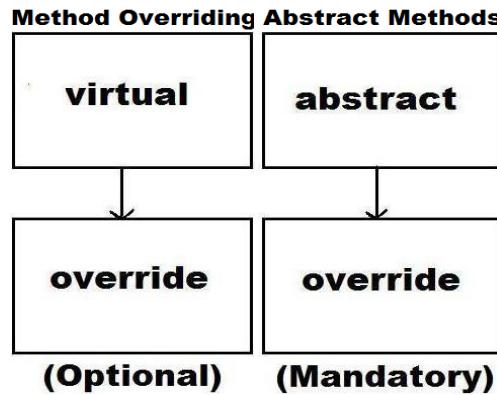
Abstract Class: a class under which we declare abstract members is known as abstract class and must be declared by using "**abstract**" modifier.

```
abstract class Math
{
    public abstract void Add(int x, int y);
}
```

Note: each and every abstract member of an abstract class **must be implemented** by the child class of abstract class without **fail** (**mandatory**).

The concept of abstract method's is near similar to **method overriding** i.e., in case of **overriding**, if at all a parent class contains any methods declared as **virtual** then child classes **can re-implement** those methods by using

`override` modifier whereas in case of abstract methods if at all a parent class contains any methods declared as `abstract` then every child class **must implement** all those methods by using the same `override` modifier only.



An abstract class can contain both `abstract` and `non-abstract (concrete)` members also, and if at all any child class of the abstract class wants to consume any `non-abstract` members of its parent, must first implement all the `abstract members` of its parent.

Abstract Class:

- Non-Abstract/Concrete Members
- Abstract Members

Child Class of Abstract Class:

- Implement each and every abstract member of parent class
- Now only we can consume concrete members of parent class

Note: we **can't create** the instance of an abstract class, so abstract classes are never useful to themselves, i.e., an abstract class is always a `parent` providing services to child classes.

To test an abstract class and abstract methods add a new class under the project naming it as "`AbsParent.cs`" and write the following code in it:

```
internal abstract class AbsParent
{
    public void Add(int a, int b)
    {
        Console.WriteLine(a + b);
    }
    public void Sub(int a, int b)
    {
        Console.WriteLine(a - b);
    }
    public abstract void Mul(int a, int b);
    public abstract void Div(int a, int b);
}
```

Now add another class “`AbsChild.cs`” to implement the above **abstract classes - abstract methods** and write the following code in it:

```
internal class AbsChild : AbsParent
{
    public override void Mul(int a, int b)
    {
        Console.WriteLine(a * b);
    }

    public override void Div(int a, int b)
    {
        Console.WriteLine(a / b);
    }

    public void Mod(int a, int b)
    {
        Console.WriteLine(a % b);
    }

    static void Main()
    {
        AbsChild c = new AbsChild();
        c.Add(100, 50); c.Add(75, 17);
        c.Mul(12, 13); c.Div(870, 15); c.Mod(121, 5);
        Console.ReadLine();
    }
}
```

Note: even if the instance of an **abstract** class can't be created it is still possible to create the **reference** of an **abstract class** by using its **child classes instance**, and with that **reference** we can call each and every **concrete method** of **abstract class** as well as its **abstract methods** which are implemented by **child class**, and to test this re-write code under **Main** method of the class “`AbsChild`” as following:

```
AbsChild c = new AbsChild();
AbsParent p = c;
p.Add(100, 50); p.Sub(75, 17);      //Methods defined in Parent class
p.Mul(12, 13); p.Div(870, 15);      //Methods implemented (override) by child class
//p.Mod(121, 5);                  //Invalid and In-accessible (Pure child class members are not accessible)
Console.ReadLine();
```

What is the need of Abstract Classes and Abstract Methods in Application development?

Ans: The concept of **Abstract Classes** and **Abstract Methods** is an extension to **inheritance** i.e., in **inheritance** we have already learnt that, we can **eliminate redundancy** between entities by identifying all the common attributes between the entities we wanted to implement, by putting them under a parent class.

For example, if we are designing a **Mathematical Application** then we follow the below process of implementation:

Step1: Identifying the **Entities** of Mathematical Application.

- Cone

- Circle
- Triangle
- Rectangle

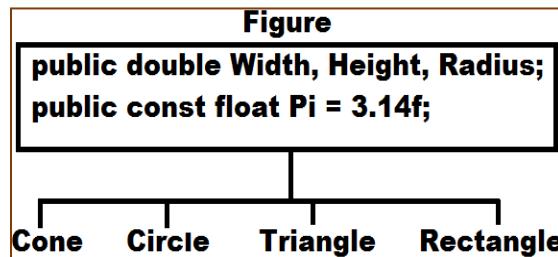
Step 2: Identifying the **Attributes** of each Entity.

- Cone: Height, Radius, Pi
- Circle: Radius, Pi
- Triangle: Base (Width), Height
- Rectangle: Length (Height), Breadth (Width)

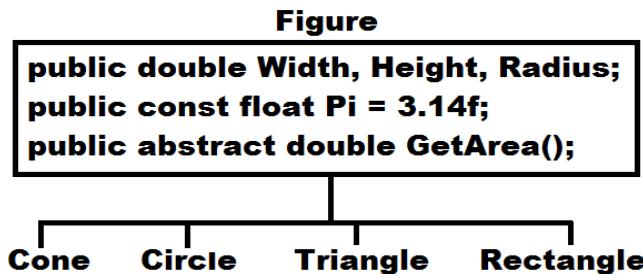
Step 3: Designing the Database by following the rules we learnt in entity implementations.

Step 4: Develop an application and define classes representing each and every entity.

Note: while defining classes representing entities, as learnt in **inheritance** first we need to define a parent class with all the common attributes as following:



In the above case, “**Figure**” is a Parent class containing all the common attributes between the 4 entities. Now we want a method that returns **Area** of each figure, and even if the method is common for all the classes, still we can't define it in the parent class **Figure**, because the **formula** to calculate area varies from figure to figure. So, to resolve the problem, without **defining** the method in parent class we need to **declare** it in the parent class **Figure** as abstract and restrict each child class to implement logic for that method as per their requirement as following:



In the above case because **GetArea()** method is declared as **abstract** in the parent class, so it is **mandatory** for all the child classes to implement that method under them, but logic can be varying from each other whereas signature of the method can't change and now all the child classes must do the following:

1. Define a constructor to initialize the attributes that are required for that entity.
2. Implement **GetArea()** method and write logic for calculating the Area of that corresponding figure.

To test the above add a “Code File” under project naming it as “TestFigures.cs” and write the following code:

```
namespace OOPSProject
{
    public abstract class Figure
    {
        public const float Pi = 3.14f;
        public double Width, Height, Radius;
        public abstract double GetArea();
    }

    public class Cone : Figure
    {
        public Cone(double Height, double Radius)
        {
            this.Height = Height;
            base.Radius = Radius; //Here this and base are same
        }

        public override double GetArea()
        {
            return Pi * Radius * (Radius + Math.Sqrt((Height * Height) + (Radius * Radius)));
        }
    }

    public class Circle : Figure
    {
        public Circle(double Radius)
        {
            this.Radius = Radius;
        }

        public override double GetArea()
        {
            return Pi * Radius * Radius;
        }
    }

    public class Triangle : Figure
    {
        public Triangle(double Base, double Height)
        {
            this.Width = Base;
            this.Height = Height;
        }

        public override double GetArea()
        {
            return 0.5 * Width * Height;
        }
    }

    public class Rectangle : Figure
    {
```

```

public Rectangle(double Length, double Breadth)
{
    this.Width = Length;
    this.Height = Breadth;
}
public override double GetArea()
{
    return Width * Height;
}
}
internal class TestFigures
{
    static void Main()
    {
        Cone cone = new Cone(18.92, 34.12);
        Console.WriteLine($"Area of Cone is: {cone.GetArea()}\n");

        Circle circ = new Circle(45.36);
        Console.WriteLine($"Area of Circle is: {circ.GetArea()}\n");

        Triangle trin = new Triangle(34.98, 27.87);
        Console.WriteLine($"Area of Triangle is: {trin.GetArea()}\n");

        Rectangle rect = new Rectangle(45.29, 76.12);
        Console.WriteLine($"Area of Rectangle is: {rect.GetArea()}\n");

        Console.ReadLine();
    }
}
}

```

Interface

Interface is also a **user-defined type** like a **class** but can contain only “**Abstract Members**” in it and all those abstract members should be **implemented** by a **child class** of the interface.

Non-Abstract Class:

Contains only **non-abstract/concrete** members

Abstract Class:

Contains both **non-abstract/concrete** and **abstract** members

Interface:

Contains only **abstract** members

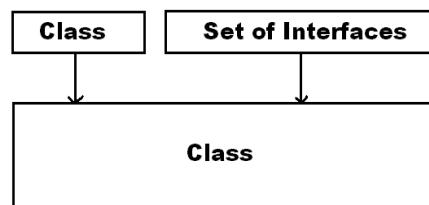
Just like a **class** can have another **class** as its **parent**, it can also have an **interface** as its **parent** but the main difference is if a **class** is a **parent**, we call it as **inheriting** whereas if an **interface** is a **parent**, we call it as **implementing**.

Inheritance is divided into 2 categories:

1. Implementation Inheritance
2. Interface Inheritance

If a class is inheriting from another **class**, we call it as **Implementation Inheritance** whereas if a class is implementing an **interface**, we call it as **Interface Inheritance**. **Implementation Inheritance** provides **re-usability** because by **inheriting** from a class we can **consume** members of **parent class** in **child class** whereas **Interface Inheritance** doesn't provide any **re-usability** because in this case we need to **implement abstract members** of a **parent** in **child class** without **fail**, but not **consume**.

Note: we have already discussed in the **5th rule of inheritance** that **Java** and **.NET Languages** doesn't support **multiple inheritance** thru **class**, because of **ambiguity problem** i.e., a class can have 1 and only 1 **immediate parent** class to it; but both in **Java** and **.NET languages** multiple inheritance is supported thru the **interfaces** i.e., a class can have more than 1 **interface** as its **immediate parent**.



Syntax to define a interface:

```
[<modifiers>] interface <Name>
{
    -Abstract member declarations.
}
```

- We can't declare any fields under an interface.
- Default scope for members of an interface is public whereas it is private in case of a class.
- Every member of an interface is by default abstract, so we again don't require using abstract modifier on it.
- Just like a class can inherit from another class, an interface can also inherit from another interface, but not from a class.

Adding an Interface under Project: Just like we have "**Class Item Template**" in "**Add New Item**" window to define a class we are also provided with an "**Interface Item Template**" to define an Interface. To test working with interfaces, add 2 interfaces under the project naming them as **IMath1.cs**, **IMath2.cs** and write the following code:

```
internal interface IMath1
{
    void Add(int x, int y);
    void Sub(int x, int y);
}

internal interface IMath2
{
    void Mul(int x, int y);
    void Div(int x, int y);
}
```

To implement methods of both the above interfaces add a new **class** under the **project** naming it as "**ClsMath.cs**" and write the following code:

```
internal class ClsMath : Program, IMath1, IMath2
{
```

```

public void Add(int a, int b)
{
    Console.WriteLine(a + b);
}
public void Sub(int a, int b)
{
    Console.WriteLine(a - b);
}
public void Mul(int a, int b)
{
    Console.WriteLine(a * b);
}
public void Div(int a, int b)
{
    Console.WriteLine(a / b);
}
static void Main()
{
    ClsMath obj = new ClsMath();
    obj.Add(100, 34); obj.Sub(576, 287);
    obj.Mul(12, 38); obj.Div(456, 2);
    Console.ReadLine();
}
}

```

Points to Ponder:

1. The implementation class can **inherit** from another class and implement “n” no. of interfaces, but class name must be first in the list followed by interface names.
E.g.: **internal class ClsMath : Program, IMath1, IMath2**
2. While declaring **abstract** members in an interface we don't require using “**abstract**” modifier on them and in the same way while implementing those abstract members we don't require to use “**override**” modifier also.

Just like we can't **create instance** of an **abstract class**, we can't **create instance** of an **interface** also; but here also we can create a **reference of interface** by using its **child class instance** and with that **reference** we can call all the members of **parent interface** which are implemented in child class and to test this **re-write** code under Main method of class “**ClsMath**” as following:

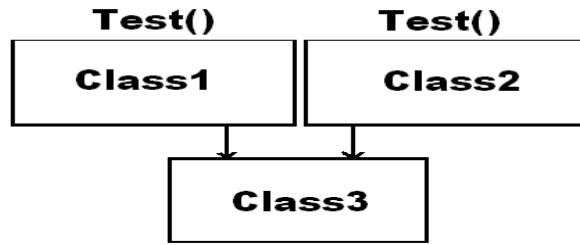
```

ClsMath obj = new ClsMath();
IMath1 i1 = obj; IMath2 i2 = obj;
i1.Add(150, 25); i1.Sub(97, 47);
i2.Mul(12, 17); i2.Div(870, 15);
Console.ReadLine();

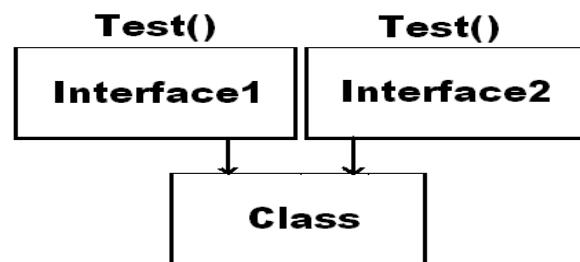
```

Multiple Inheritance with Interfaces:

Earlier in the **5th rule of inheritance** we have discussed that **Java** and **.NET Languages** doesn't support **multiple inheritances** thru **classes** because of **ambiguity** problem.



Whereas in **Java** and **.NET** Languages, **multiple inheritance** is supported thru **interfaces** i.e., a class can have any no. of interfaces as its immediate parent, but still we don't come across any **ambiguity** problems because child class of an interface is not **consuming** parent's members but **implements** them.



If we come across any situation as **above**, we can **implement** the **interface** methods under **class** by using 2 different approaches:

1. Implement the method of both **interfaces** only for 1 time under the **class** and both **interfaces** will assume the implemented **method** is of its only and in this case, we can call the method directly by using class **instance**.
2. We can also implement the method of both **interfaces** separately for each **interface** under the **class** by pre-fixing **interface** name before **method** name and we call this as **explicit implementation**, but in this case, we need to call the **method** by using **reference of interface** that is created with the help of a child class **instance**.

To test the above add 2 new **interfaces** under the **project** naming them as **Interface1.cs**, **Interface2.cs** and write the following code:

```

internal interface Interface1
{
    void Test();
    void Show();
}

internal interface Interface1
{
    void Test();
    void Show();
}
  
```

Now add a new class under the project naming it as "**ImplClass.cs**" for implementing both the above **interfaces** and write the following code:

```

internal class ImplClass : Interface1, Interface2
{
    //Implementing Test method using 1st approach
    public void Test()
    {
        Console.WriteLine("Method declared under 2 interfaces.");
    }
}
  
```

```

}

//Implementing Show method using 2nd approach
void Interface1.Show()
{
    Console.WriteLine("Method declared under Interface1.");
}

//Implementing Show method using 2nd approach
void Interface2.Show()
{
    Console.WriteLine("Method declared under Interface2.");
}

static void Main()
{
    ImplClass c = new ImplClass();
    c.Test();

    Interface1 i1 = c;
    Interface2 i2 = c;

    i1.Show();
    i2.Show();
    Console.ReadLine();
}
}

```

Structure

Structure is also a **user-defined type** like a **class** and interface which can contain only non-abstract members. A **structure** can contain all the **members** what a class can contain like **constructor**, **static constructor**, **constants**, **fields**, **methods**, **properties**, **indexers**, **operators**, and **events**.

Differences between Class and Structure

Class	Structure
This is a reference type.	This is a value type.
Memory is allocated for its instances on Managed Heap, so we get the advantage of Automatic Memory Management thru Garbage Collector.	Memory is allocated for its instances on Stack, so Automatic Memory Management is not available but faster in access.
Recommended for representing entities with larger volumes of data.	Recommended for representing entities with smaller volumes of data.

All pre-defined reference types in our Libraries like string (System.String) and object (System.Object) are defined as classes.	All pre-defined value types in our Libraries like int (System.Int32), float (System.Single), bool (System.Boolean), char (System.Char) and Guid (System.Guid) are defined as structures.
"new" keyword is mandatory for creating the instance and in this process, we need to call any constructor that is available in the class.	"new" keyword is optional for creating the instance and if "new" is not used it will call default constructor which is defined implicitly, whereas it is still possible to use "new" and call other constructors also.
Contains an implicit default constructor if no constructor is defined explicitly.	Contains a default constructor every time which can be implicitly or explicitly defined.
We can declare fields and those fields can be initialized at the time of declaration.	We can declare fields, but those fields can't be initialized at the time of declaration, if there is no explicit constructor.
Fields can also be initialized thru a constructor as well as referring thru instance also we can initialize them.	Fields can only be initialized thru a constructor as well as referring thru instance also we can initialize them.
Constructor is mandatory for creating the instance which can either be default or parameterized also.	Default constructor is mandatory for creating the instance without using new keyword and apart from that we can also define parameterized constructors.
Developers can define any constructor like default or parameterized also, or else implicit default constructor gets defined.	Developers can define parameterized constructors only up to C# 9.0 whereas from C# 10.0 developers can also define default constructors. Default constructor is mandatory if at all we want to create instance without using "new" keyword.
If defined with "0" constructors, after compilation there will be "1" constructor and if defined with "n" constructors, after compilation there will be "n" constructors only.	If defined with "0" constructors, after compilation here also there will be "1" constructor whereas if defined with "n" parameterized constructors, after compilation there will be "n + 1" constructors.
Supports both, implementation as well as interface inheritances also i.e., a class can inherit from another class as well as implement an interface also.	Supports only interface inheritance but not implementation inheritance i.e., a structure can implement an interface but can't inherit from another structure.

Syntax to define a structure:

```
[<modifiers>] struct <Name>
{
    -Define only non-abstract Members
}
```

Adding a Structure under Project: we are not provided with any Structure Item template in the add new item window, like we have class and interface item templates, so we need to use code file item template to define a structure under the project. Add a **Code File** under project, naming it as "**MyStruct.cs**" and write the following in it:

```
namespace OOPSProject
{
    internal struct MyStruct
    {
```

```

int x;
public MyStruct(int x)
{
    this.x = x;
}
public void Display()
{
    Console.WriteLine("Method defined under a structure: " + x);
}
static void Main()
{
    MyStruct m1 = new MyStruct(); m1.Display();
    MyStruct m2; m2.x = 10; m2.Display();
    MyStruct m3 = new MyStruct(20); m3.Display();
    Console.ReadLine();
}
}
}
}

```

Consuming a Structure: we can consume a **structure** and its members from another **structure** or a **class** also; but only by creating its **instance** because structure doesn't support **inheritance**. To test this, add a new **class** under the project naming it as “**TestStruct.cs**”, change the **class** keyword to **struct** and write the below code:

```

internal struct TestStruct
{
    static void Main()
    {
        MyStruct obj1 = new MyStruct(); obj1.Display();
        MyStruct obj2 = new MyStruct(30); obj2.Display();
        Console.ReadLine();
    }
}

```

Working with Multiple Projects and Solution

While developing an application sometimes code will be written under more than **1 project** also, where collection of all those **projects** is known as a **Solution**. Whenever we **create a new project** by default **Visual Studio** will create one **Solution** and under it the project gets added, where a **Solution** is a collection of **Projects** and **Project** is a collection of **Items** or **Files** and each **Item** or **File** is a collection of **Types** (**Class**, **Structure**, **Interface**, **Enum** and **Delegate**), and each **Type** is a collection of **Members** (**Fields**, **Methods**, **Constructors**, **Finalizers**, **Properties**, **Indexers**, **Events** and **Deconstructor**)

A **Solution** also requires a **Name**, which can be specified by us while creating a new **Project** or else it will take **Name** of the first **Project** that is created under **Solution**, if not specified. In our case **Solution Name** is “**OOPSProject**” because our **Project Name** is “**OOPSProject**”. A **Solution** can have **Projects** of different **.NET Languages** as well as can be of different **Project Templates** also like **Windows App's**, **Console App's**, **Class Library** etc. but a project cannot contain items of different **.NET Languages** i.e., they must be specific to **1 Language** only.

To add a new **Project** under our “**OOPSProject**” solution, right click on **Solution Node** in **Solution Explorer** and select add “**New Project**” which opens the new **Project Window**, under it select **Language as Visual C#, Template as Console Application**, name the **Project** as “**SecondProject**” and click **Ok** which adds the new **Project** under the “**OOPSProject**” solution.

By default, the new **Project** also comes with a class “**Program**” but under “**SecondProject Namespace**”, now write the below code in the file by deleting the existing code:

```
internal class Program
{
    static void Main()
    {
        Console.WriteLine("Second project under the solution.");
        Console.ReadLine();
    }
}
```

To run the above class, first we need to set a property i.e., “**StartUp Project**”, because there are multiple **Projects** under the **Solution** and **Visual Studio** by default runs first **Project** of the **Solution** only i.e., “**OOPSProject**” under the solution. To set the “**StartUp Project**” property and run classes under “**SecondProject**” open **Solution Explorer**, right click on “**SecondProject**”, select “**Set as StartUp Project**”, and then run the **Project**.

Note: if the new **Project** is added with new **Classes** we need to again set “**StartUp Object**” property under **Second Project’s** project file, because each project has its own property **Window**.

Saving Solution and Projects: The application what we have created right now is saved **physically on hard disk** in the same hierarchy as seen under **Solution Explorer** i.e., first a folder is created representing the **Solution** and under that a **separate folder** is created representing each **Project** and under that **Items** or **Files** corresponding to that **Project** gets saved and the path of the **Project** will be as following:

```
<drive>:\<our_personal_folder>\OOPSProject\OOPSProject => Project1  
<drive>:\<our_personal_folder>\OOPSProject\SecondProject => Project2
```

Note: A **Solution** will be having a file called **Solution file**, which gets saved with “**.sln**” extension and a **Project** also has a file called **Project file**, where a **C# Project** file gets saved with “**.csproj**” extension which can contain “**C#**” items only.

Compilation of Projects: whenever a **Project** is compiled it generates an output file known as “**Assembly**” that contains “**CIL Code**” of all the “**Types**” that are defined in the **Project**.

What is an Assembly?

- It’s an output file that is generated after compilation of a project which contains **CIL Code** in it.
- Assembly file contains the **CIL Code** of each type that is defined under the project.
- An **Assembly** is a **unit of deployment**, because when we need to install an application on client machines what we install is these **Assemblies** only and all the **.NET Libraries** are installed on our machines in the form of **Assemblies** when we install **Visual Studio**.

- The name of an **assembly** file is the same name of the **project** and can't be **changed**.
- In **.NET Framework** the assembly files of a project will be present under the project folder's "**bin\debug**" folder.
In **.NET Core**, assembly file of a project will be present under "**bin\debug\netcoreapp<Version>**" folder and here version represents the **Core Runtime** version. From **.NET 5**, assembly file of a project will be present under "**bin\debug\net<Version>**" folder and here also version represents the **Runtime** version.
- In **.NET Framework** the **extension** of an **assembly** file can either be a "**.exe**" or "**.dll**" which is based on the type of project we open, for example if the project is an "**Application Project**" then it will generate "**.exe**" assembly whereas if it is a "**Library Project**" then it will generate "**.dll**" assembly. From **.NET Core** every project will generate "**.dll**" assembly and apart from that "**Application Project's**" will generate an additional "**.exe**" assembly also i.e., "**Library Projects**" will be generating "**.dll**" only now also where as "**Application Project's**" will generate both "**.exe**" and "**.dll**" also.

.NET Framework:

- Application Projects => Generates only "**.exe**".
- Library Projects => Generates only "**.dll**".

.NET Core & above:

- Application Projects => Generates both "**.exe**" and "**.dll**" also.
- Library Projects => Generates only "**.dll**".

Note: Generally, "**.dll**" assemblies can't run but a "**.dll**" assembly that are generated by **Application Projects** can run or execute on **Linux** and **MAC** Machines also by using the tool: "**.NET Core CLI (Command Line Interface)**" as following:

```
dotnet <Assembly_Name>.dll
```

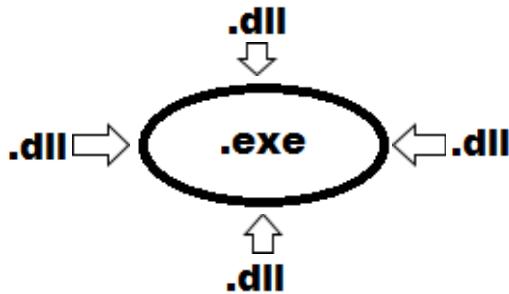
What is an ".exe" assembly?

Ans: In Windows OS "**.exe**" assemblies are known as **in-process** components i.e., these assemblies are physically loaded into the **memory** for execution and run-on **Windows Machines**.

What is a ".dll" assembly?

Ans: In Windows O.S. "**.dll**" assembly are known as **out-process** components i.e., these assemblies sit out of the memory providing support to the 1 who is running in the memory. In **.NET Framework**, "**.dll**" assemblies can never run on their own i.e., they can only be consumed from other projects, whereas from **.NET Core** and above the "**.dll**" assemblies that are generated by "**Application Projects**" can run on **Windows**, **Linux**, and **Mac** Machine with the help of **.NET Core CLI**.

Note: An assembly is a unit of deployment, because when we need to install or deploy an application on client machines what we install is these assemblies only and every application is a blend of "**.dll's**" and "**.exe**" assemblies combined to give better efficiency.



The assembly files of our 2 projects i.e., OOPSPProject and SecondProject will be at the following location:

<drive>:\<folder>\OOPSPProject\OOPSPProject\bin\Debug\net8.0\OOPSPProject.dll|.exe => Assembly1
<drive>:\<folder>\SecondProject\bin\Debug\net8.0\SecondProject.dll|.exe => Assembly2

ildasm: Intermediate Language Dis-Assembler. We use it to dis-assemble an Assembly file and view the contents of it. To check it out, open **Visual Studio Developer Command Prompt**, go to the location where the assembly files of the project are present and use it as following:
ildasm <name of the .dll assembly file>

Note: in **.NET Framework** we can dis-assemble both “.exe” and “.dll” assemblies also whereas from **.NET Core** we can dis-assemble only “.dll” assemblies.

E.g.: Open **Visual Studio Developer Command Prompt**, go to the below location and try the following:

```

<drive>:\<our_folder>\OOPSPProject\OOPSPProject\bin\Debug\net8.0> ildasm OOPSPProject.dll
<drive>:\<our_folder>\SecondProject\bin\Debug\net8.0> ildasm SecondProject.dll

```

Q. Can we consume classes of a project from other classes of same project?

Ans: Yes, we can consume them directly because all those classes were under the same project and will be considered as a family.

Q. Can we consume the classes of a project from other projects?

Ans: Yes, we can consume them, but not directly, as they are under different projects. To consume them first we need to add reference of the assembly in which the class is present to the project who wants to consume it.

Q. How to add the reference of an assembly to a project?

Ans: To add reference of an assembly to a project open solution explorer, right click on the project to whom reference must be added, select “Add => Project Reference” option, which opens a window “Reference Manager” and in that window select “Browse” option in LHS, then click on “Browse” button below, select the assembly we want to consume from its physical location and click ok. Now we can consume types of that assembly by prefixing with their namespace or importing the namespace.

Note: In **.NET Framework** we can add reference to “.exe” or “.dll” assemblies also and consume them in other projects, whereas from **.NET Core** onwards we can’t add reference to “.exe” assemblies i.e., we can add reference only to “.dll” assemblies.

To test this, go to “**OOPSPProject**” Solution, right click on the “**SecondProject**” we have newly added, select add reference and add the reference of “**OOPSPProject.dll**” assembly from its physical location

(<drive>:\<our_folder>\OOPSProject\OOPSProject\bin\Debug\net8.0>). Now add a new class under the "SecondProject" naming it as "Class1.cs" and write the below code in it:

```
using OOPSProject;
internal class Class1
{
    static void Main()
    {
        Cone cone = new Cone(18.92, 34.12);
        Console.WriteLine($"Area of Cone is: {cone.GetArea()}\n");

        Circle circ = new Circle(45.36);
        Console.WriteLine($"Area of Circle is: {circ.GetArea()}\n");

        Triangle trin = new Triangle(34.98, 27.87);
        Console.WriteLine($"Area of Triangle is: {trin.GetArea()}\n");

        Rectangle rect = new Rectangle(45.29, 76.12);
        Console.WriteLine($"Area of Rectangle is: {rect.GetArea()}\n");
        Console.ReadLine();
    }
}
```

Assemblies and Namespaces: An assembly is an output file which gets generated after compilation of a project and it is physical. The name of an assembly file will be same as project name and can't be changed at all.

Project: Source Code

Assembly: Compiled Code (IL Code)

A namespace is a logic container of types which are used for grouping types. By default, every project has a namespace, and its name is same as the project name, but we can change namespace names as per our requirements and more over a project can contain multiple namespaces in it also.

For Example: DBOperations (Console App.) when compiled generates an assembly with the names as DBOperations.exe and DBOperations.dll, under it, namespaces can be as following:

```
namespace SQL
{
    Class1
    Class2
}
namespace Oracle
{
    Class3
    Class4
}
namespace MySQL
{
```

```
Class5  
Class6  
}
```

Whenever we want to consume a type which is defined under 1 project from other projects, we need to follow the below process:

Step 1: add reference of the assembly corresponding to the project we want to consume.

Step 2: import the namespace under which the class is present.

Step 3: now either create the instance of the class or inherit from the class and consume it.

Access Specifier's

These are a special kind of modifiers using which we can define the scope of a type and its members i.e., who can access them and who cannot. C# supports 5 access specifiers in it, those are:

- 1. Private 2. Internal 3. Protected 4. Protected Internal 5. Public 6. Private Protected (C# 7.3)**

Note: members that are defined in a type with any scope or specifier are always accessible within the type, restrictions come into picture only when we try to access them outside of the type.

Private: members declared as **private** under a **class** or **structure** can be accessed only within the **type** in which they are defined and moreover their **default scope** is **private** only. **Interfaces** can't contain any **private** members and their **default scope** is **public**. **Types** can't be declared as **private**, so this applies only to **members**.

Protected: members declared as **protected** under a class can be accessed only from their **child** class i.e., **non-child** classes can't consume them. **Types** can't be declared as **protected**, so this applies only to **members**.

Internal: members and types that are declared as **internal** can be consumed only within the **project**, both from a **child** or **non-child**. The **default scope** for a type in **C#** is **internal** only.

Protected Internal: members declared as **protected internal** will have **dual scope** i.e., within the project they behave as **internal** providing access to whole project and outside the project they will change to **protected** and provide access to their child classes. **Types** can't be declared as **protected internal**, so this applies only to **members**.

Public: a **type** or **member** of a type if declared as **public** is **global** in scope which can be accessed from anywhere.

Private Protected (Introduced in C# 7.3 Version): members declared as **private protected** in a class are accessible only from the **child** classes that are defined in the same project. Types can't be declared as **private protected**, so this applies only to **members**.

To test access specifiers, create a new **C# Project** of type "**Console App.**", name the project as "**AccessDemo1**" and re-name the solution as "**MySolution**", click **Next** and choose the **Framework ".NET 8.0"** and Check the **CheckBox "Do not use top-level statements."**, so that it generates **Program class** and **Main method** also.

By default, the project comes with a class **Program** and its default scope is **internal**, so change it as **public** so that it can be **accessed** from other **projects** also and write the below code in the class:

```

//Consuming members of a class from the same class
public class Program
{
    private void Test1_Private()
    {
        Console.WriteLine("Private Method");
    }
    internal void Test2_Internal()
    {
        Console.WriteLine("Internal Method");
    }
    protected void Test3_Protected()
    {
        Console.WriteLine("Protected Method");
    }
    protected internal void Test4_ProtecedInternal()
    {
        Console.WriteLine("Protected Internal Method");
    }
    public void Test5_Public()
    {
        Console.WriteLine("Public Method");
    }
    private protected void Test6_PrivateProtected()
    {
        Console.WriteLine("Private Protected Method");
    }
    static void Main(string[] args)
    {
        Program p = new Program();
        p.Test1_Private();
        p.Test2_Internal();
        p.Test3_Protected();
        p.Test4_ProtecedInternal();
        p.Test5_Public();
        p.Test6_PrivateProtected();
        Console.ReadLine();
    }
}

```

Now add a new class “Two.cs” under the project and write the following:

```

//Consuming members of a class from child class of same project
internal class Two : Program
{
    static void Main()
    {
        Two t = new Two();
    }
}

```

```

        t.Test2_Internal();
        t.Test3_Protected();
        t.Test4_ProTECTEDInternal();
        t.Test5_Public();
        t.Test6_PrivateProtected();
        Console.ReadLine();
    }
}

```

Now add another new class “Three.cs” in the project and write the following:

```

//Consuming members of a class from non-child class of same project
internal class Three
{
    static void Main()
    {
        Program p = new Program();
        p.Test2_Internal();
        p.Test4_ProTECTEDInternal();
        p.Test5_Public();
        Console.ReadLine();
    }
}

```

Now Add a new “Console App” project under “MySolution”, name it as “AccessDemo2”, rename the default file “Program.cs” as “Four.cs” so that class name also changes to **Four**, add a reference to “AccessDemo1” assembly from its physical location to the new project and write the below code in the class **Four**:

```

//Consuming members of a class from child class of another project
internal class Four : AccessDemo1.Program
{
    static void Main(string[] args)
    {
        Four f = new Four();
        f.Test3_Protected();
        f.Test4_ProTECTEDInternal();
        f.Test5_Public();
        Console.ReadLine();
    }
}

```

Now add a new class under AccessDemo2 project, naming it as Five.cs and write the following:

```

//Consuming members of a class from non-child class of another project
internal class Five
{
    static void Main()
    {
        Program p = new Program();

```

```

        p.Test5_Public();
        Console.ReadLine();
    }
}

```

Cases	Private	Internal	Protected	Private Protected	Protected Internal	Public
Case 1: Same Class - Same Project	Yes	Yes	Yes	Yes	Yes	Yes
Case 2: Child Class - Same Project	No	Yes	Yes	Yes	Yes	Yes
Case 3: Non-Child Class - Same Project	No	Yes	No	No	Yes	Yes
Case 4: Child Class - Another Project	No	No	Yes	No	Yes	Yes
Case 5: Non-Child Class - Another Project	No	No	No	No	No	Yes

Language Interoperability

As discussed earlier the code written in **1 .NET Language** can be consumed from any other **.NET Languages** and we call this as **Language Interoperability**. Differences between **VB.NET** and **C#** languages:

- VB is not a case-sensitive language like C#.
- VB does not have any curly braces; the end of a block is represented with a matching End Stmt.
- VB does not have any semi colon terminators so each statement must be in a new line.
- The extension of source files will be “.vb” in VB.NET and “.cs” in C#.

To test this, add a new “Console App” project under “MySolution” choosing the language as **Visual Basic** and name the project as “**AccessDemo3**”. By default, the project comes with a file “**Module1.vb**”, so open solution explorer, delete that file under project and add a new class naming it as “**TestCS.vb**”. Now add the reference of “**AccessDemo1**” assembly to the current project, choosing it from its physical location and write the following code under the class **TestCS**:

```

Imports AccessDemo1
Public Class TestCS : Inherits Program
    Shared Sub Main()
        'Creating instance of the class
        Dim obj As New TestCS()
        obj.Test3_Protected()
        obj.Test4_ProTECTEDInternal()
        obj.Test5_Public()
        Console.ReadLine()
    End Sub
End Class

```

Note: to run the class set **Startup Project** as current project i.e., “**AccessDemo3**” and execute.

Consuming VB.Net Code in CSharp: Now to test consuming **VB.NET** code in **C#**, add a new project under “**MySolution**”, choosing the language as **Visual Basic**, project type as “**Class Library**” and name the project as “**AccessDemo4**”. A **Class Library** is a collection of types that can be **consumed** but not **executed**. After **compilation** the extension of project’s assembly will be “**.dll**”.

In VB.Net language methods are divided into 2 categories like:

1. Functions (Value returning methods)

2. Sub-Routines (Non-value returning methods)

By default, the project comes with a class Class1 within the file Class1.vb, write the below code in it:

```
Public Class Class1
    Public Function SayHello(Name As String) As String
        Return "Hello " & Name
    End Function
    Public Sub AddNums(x As Integer, y As Integer)
        Console.WriteLine($"Sum of given 2 no's is: {x + y}")
    End Sub
    Public Sub Math(a As Integer, b As Integer, ByRef c As Integer, ByRef d As Integer)
        c = a + b
        d = a * b
    End Sub
End Class
```

Now to compile the project open solution explorer, right click on “AccessDemo4” project and select “Build” option which compiles and generates an assembly “AccessDemo4.dll”.

Now add a new class under “AccessDemo2” project with the name “TestVB.cs”, add reference of “AccessDemo4.dll” assembly from its physical location and write the below code under the class TestVB:

```
using AccessDemo4;
internal class TestVB
{
    static void Main()
    {
        Class1 obj = new Class1();
        obj.AddNums(100, 50);
        string str = obj.SayHello("Raju");
        Console.WriteLine(str);
        int Sum = 0, Product = 0;
        obj.Math(100, 25, ref Sum, ref Product);
        Console.WriteLine("Sum of the given 2 no's is: " + Sum);
        Console.WriteLine("Product of the given 2 no's is: " + Product);
        Console.ReadLine();
    }
}
```

Note: to run the class set both Startup Project and Startup Object properties also.

Q. How to restrict a class not to be accessible for any other class to consume?

Ans: This can be done by declaring all the class constructors as private.

Q. How to restrict a class not to be inherited for any other class?

Ans: This can be done by declaring class as sealed.

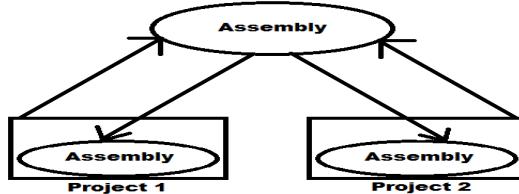
Q. How to restrict a class not to be accessible for any other class to consume by creating its instance?

Ans: This can be done by declaring all the class constructors as protected.

Assemblies are of 2 types:

1. Private Assembly
2. Shared Assembly

Private Assembly: By default, every assembly is private, if reference of these assemblies was added to any project; a copy of assembly is created and given to that project, so that each project maintains a private copy of assembly.



Creating an assembly to test it is by default private: create a new project of type **Class Library** and name it as “**PAssembly**”, which will by default come with a class **Class1** under the file **Class1.cs**. Now write the following code under the class:

```
public string SayHello()  
{  
    return "Hello from private assembly.";  
}
```

Now compile the project by opening the Solution Explorer, right click on the project and select “**Build**” which will compile and generate an assembly with the name as **PAssembly.dll**.

Note: we can find path of the assembly in output window present at bottom of the Visual Studio.

Consuming the assembly, we have created in multiple projects: We can consume an **assembly** under any no. of projects, but if the **assembly** is **private**, it will create multiple copies of the **assembly** i.e., in how many projects the reference is added that many no. of copies are also created for the **assembly**.

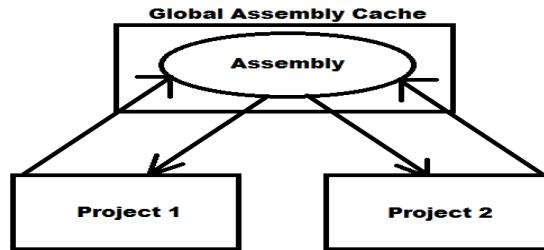
To test this, create 2 new projects of type **Console App.s**, naming it as “**TestPAssembly1**” and “**TestPAssembly2**”. Add the reference of “**PAssembly.dll**” we have created above to both the projects, from its physical location and write the below code under **Main** method of the default class **Program**:

```
PAssembly.Class1 obj = new PAssembly.Class1();  
Console.WriteLine(obj.SayHello());  
Console.ReadLine();
```

Run both the projects to test them and then go and verify under **bin/debug/net8.0** folder of both the projects where we can find a copy of “**PAssembly.dll**” because **PAssembly.dll** is a private assembly.

Note: the advantage of a private assembly is faster execution as it was in the local folder of consumer project, whereas the draw back was multiple copies gets created when multiple projects add the reference to consume it.

Shared Assemblies: If we intend to use an assembly among several applications, private assemblies are not feasible, in such cases we can install the Assembly into a centralized location known as the “Global Assembly Cache”. Each computer where the “.NET Runtime” is installed has this machine-wide code cache. The Global Assembly Cache stores assemblies specifically designated to be shared by several applications on the computer. All .NET Libraries are assemblies and are shared “.dll” assemblies only, so we can find them under GAC. If an assembly is shared multiple copies of the assembly will not be created even if being consumed by multiple projects i.e., only a single copy under GAC serves all the projects.



Note: administrators often protect the Windows directory using an access control list (ACL) to control write and execute access. Because the global assembly cache is installed in the Windows directory, it inherits that directory's ACL. It is recommended that only users with Administrator privileges be allowed to add or delete files from the global assembly cache.

Location of GAC Folder: <OS Drive>:\Windows\Microsoft.NET\assembly\GAC_MSIL

How to make an assembly as Shared?

Ans: to make an assembly as Shared we need to install the assembly into GAC.

How to install an assembly into GAC?

Ans: To manage assemblies in GAC like install, un-install and view we are provided with a tool known as Gacutil.exe (Global Assembly Cache Tool). This tool is automatically installed with VS. To run the tool, use the Visual Studio Command Prompt. These utilities enable you to run the tool easily, without navigating to the installation folder. To use Global Assembly Cache on your computer: On the taskbar, click Start, click All Programs, click Visual Studio, click Visual Studio Tools, and then click Visual Studio Command Prompt and type the following:

```
gacutil -i | -u | -l [<assembly name>]  
or  
gacutil /i | /u | /l [<assembly name>]
```

What assemblies can be installed into the GAC?

Ans: We can install only Strong Named Assemblies into the GAC.

What is a Strong Named Assembly?

Ans: assemblies deployed in the global assembly cache must have a strong name. When an assembly is added to the global assembly cache, integrity checks are performed on all files that make up the assembly.

Strong Name: A strong name consists of the assembly's identity - its simple text like: name, version number, and public key.

1. **Name:** it was the name of an assembly used for identification. Every assembly by default has name.
2. **Version:** software's maintain versions for discriminating changes that has been made from time to time. As an assembly is also a software component it will maintain versions, whenever the assembly is created it has a default version for it i.e., 1.0.0.0, which can be changed when required.

3. **Public Key:** as GAC contains multiple assemblies in it, to identify each assembly it will maintain a key value for the assembly known as public key, which should be generated by us and associate with the assembly to make it Strong Named.

You can ensure that a name is globally unique by signing an assembly with a strong name. Strong names satisfy the following requirements:

- Strong names guarantee name uniqueness by relying on unique key pairs. No one can generate the same assembly's name that you can. Strong names protect the version lineage of an assembly.
- A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version the application was built with.
- Strong names provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built.

Generating a Public Key: To sign an assembly with a strong name, you must have a public key pair. This public cryptographic key pair is used during compilation to create a strong-named assembly. You can create a key pair using the Strong Name tool (Sn.exe) from visual studio command prompt as following:

Syntax: sn -k <file name>

E.g.: sn -k Key.snk

Note: the above statement generates a key-value and writes it into the file “Key.snk”. Key/Value pair files usually have the extension of “.snk” (strong name key).

Creating a Shared Assembly:

Step 1: generate a public key. Open VS command prompt, go into your personal folder and generate a public key as following:

<drive>:\<CSharp> sn -k Key.snk

Step 2: create a new project and add the key file to it before compilation so that the assembly which is generated will be Strong Named. To do this open a new project of type **Class Library**, name it as “**SAssembly**” and write the below code under the class Class1:

```
public string SayHello1()
{
    return "Hello from shared assembly => 1.0.0.0";
}
```

To associate key file we have generated, with the project, open project properties window and to do that open Solution Explorer, right click on the Project and choose the option “**Properties**” and in the window opened select **Build** tab on LHS and under that click on “**Strong naming**” item, which displays a **Checkbox** as “**Sign the output assembly to give it a strong name.**” in RHS, select it, which display a **File Upload Control** with **Browse** button to select the file from its physical location, so click on **Browse** and select the “**Key.snk**” file from its physical location which adds the file to the project and we can find that information in “**.csproj**” file under the XML key “**<AssemblyOriginatorKeyFile>**”. Now compile the project using “**Build**” option that will generate “**SAssembly.dll**” which is **Strong Named**.

Step 3: installing assembly into GAC by using the “**Global Assembly Cache Tool**”. To install the assembly into GAC open Visual Studio - Developer Command Prompt in Administrator Mode, go to the location where “**SAssembly.dll**” is present and write the below statement:

```
<drive>:\<folder\>SAssembly\bin\Debug\net8.0> gacutil -i SAssembly.dll
```

Step 4: testing the Shared Assembly. Create a new project of type Console App., name it as “**TestSAssembly1**”. Add reference to “**SAssembly.dll**” from its physical location and write the below code under **Main** method of the default class **Program**:

```
SAssembly.Class1 obj = new SAssembly.Class1();
MessageBox.Show(obj.SayHello1());
```

Note: Run the project, test it, and now this project i.e., “**TestSAssembly1**” can run by using the “**SAssembly.dll**” which is present in “**GAC**” i.e., we don’t require a copy of the “**SAssembly.dll**” to be present under the local folder.

Versioning Assemblies: Every assembly is associated with a set of **attributes** that describes about general info of an assembly like **Title**, **Company**, **Description**, **Version** etc. These attributes will be under “**.csproj**” file of the project. To view the “**.csproj**” file right click on the **project** in **Solution Explorer** and select “**Edit Project File**” which will open the “**.csproj**” file of the project. This is an **XML** file and under this file right now we find code as below:

```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
<TargetFramework>net8.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
<SignAssembly>true</SignAssembly>
<AssemblyOriginatorKeyFile>Key.snk</AssemblyOriginatorKeyFile>
</PropertyGroup>
</Project>
```

We can also specify various other details regarding the **assembly** like **Company**, **Version**, **etc, etc**, under this file by using **XML Tags**, for example if we want to specify the name of the **Company** who designed and developed this assembly we can use “**<Company>**” tag and we need to use it inside of “**<PropertyGroup>**” tag as following:

```
<Company>NIT</Company>
```

Why do we maintain version numbers to an assembly?

Ans: Version no. is maintained for **discriminating** the **changes** that has been made from time to time. **Version No** is changed for an **assembly** if there are any **modifications** or **enhancements** made in the code. The default version of every assembly is **1.0.0.0** and version no is a combination of 4 values like:

1. Major Version
2. Minor Version
3. Build Number
4. Revision

What are the criteria for changing the version no. of an assembly?

Ans: we change version no. of an assembly basing on the following criteria:

1. Change the Major version value when we add new types under the assembly.
2. Change the Minor version value when we modify any existing types under the assembly.
3. Change the Build Number when we add new members under types.
4. Change the Revision value when we modify any existing members under types.

Where do we change the version no. of an assembly?

Ans: we need to change the version no. of an assembly under the project property file, and to do that open **Solution Explorer**, right click on the **project**, select the option “**Edit Project File**” which opens the **Project File** in **XML** format. Now under **<PropertyGroup>** tag we need to add the below statements in the last:

```
<AssemblyVersion>-Specify the new version no. here</AssemblyVersion>
<FileVersion>-Specify the new version no. here</FileVersion>
```

Testing the process of changing version no of an assembly: Open the **SAssembly** project we have developed earlier and add a new method under Class1 as below:

```
public string SayHello2()
{
    return "Hello from shared assembly => 1.0.1.0";
}
```

Open project property file and set Company, Description, Assembly Version & File Version attributes as below:

```
<Company>NIT</Company>
<Description>This is a shared assembly developed by Naresh I Technologies.</Description>
<AssemblyVersion>1.0.1.0</AssemblyVersion>
<FileVersion>1.0.1.0</FileVersion>
```

Now Re-build the project and add the new version of “**SAssembly.dll**” i.e., “**1.0.1.0**” also into **GAC** using the **Gacutil Tool**.

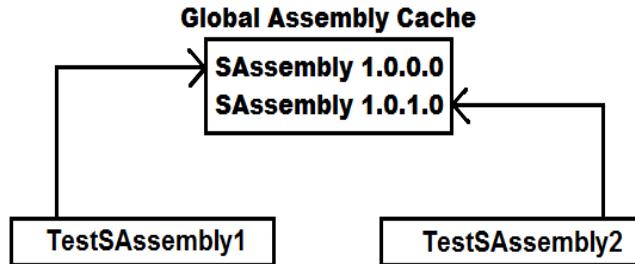
Note: GAC allows placing of multiple **versions** of an **assembly** in it and provides different **applications** using different **versions** of the assembly to **execute** correctly using their required version. Now if we open the GAC folder there, we will find 2 versions of “**SAssembly**” i.e., “**1.0.0.0**” and “**1.0.1.0**”.

Assemblies and Side-by-Side Execution: Side-by-side execution is the ability to store and execute **multiple versions** of an application or component on the same **computer**. Support for **side-by-side storage** and **execution** of different versions of the same assembly is an **integral part** of **strong naming** and is built into the infrastructure of the **.NET Runtime**. Because the strong-named assembly version number is part of its **identity**, the **.NET Runtime** can store multiple versions of the same assembly in the **Global Assembly Cache** and loads those assemblies at run time.

To test side-by-side execution, create a new project of type **Console App.**, name it as “**TestSAssembly2**”. Add reference to “**SAssembly.dll**” from its physical location and write the below code under **Main** method of default class **Program**:

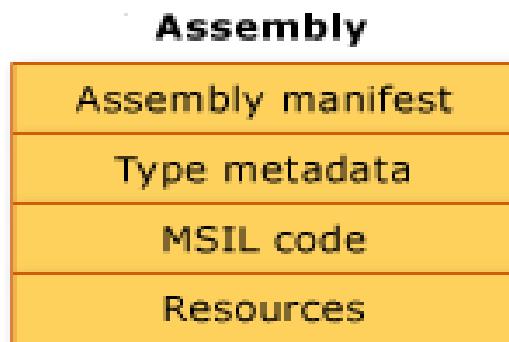
```
SAssembly.Class1 obj = new SAssembly.Class1();
MessageBox.Show(obj.SayHello2());
```

To check **side-by-side** execution of projects run the “**exe files**” of “**TestSAssembly1**” and “**TestSAssembly2**” projects at the same time, where each project will use its required version of “**SAssembly**” and execute, as below:



In general, an assembly is divided into four sections:

- The assembly manifest, which contains assembly metadata.
- Type metadata.
- Microsoft intermediate language (MSIL) Code or CIL Code that implements the types.
- A set of resources.



Assembly Manifest: contains information about the attributes that are associated with an assembly like **Assembly Name, Assembly Version, File Version, Company, Strong Name Information, List of files in the assembly etc.**

Type Metadata: describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

- Description of the assembly.
 - Identity (name, version, culture, public key).
 - Other assemblies that this assembly depends on.
 - Security permissions needed to run.
- Description of types.
 - Name, visibility, base class, and interfaces implemented.
 - Members (methods, fields, properties, events, nested types).

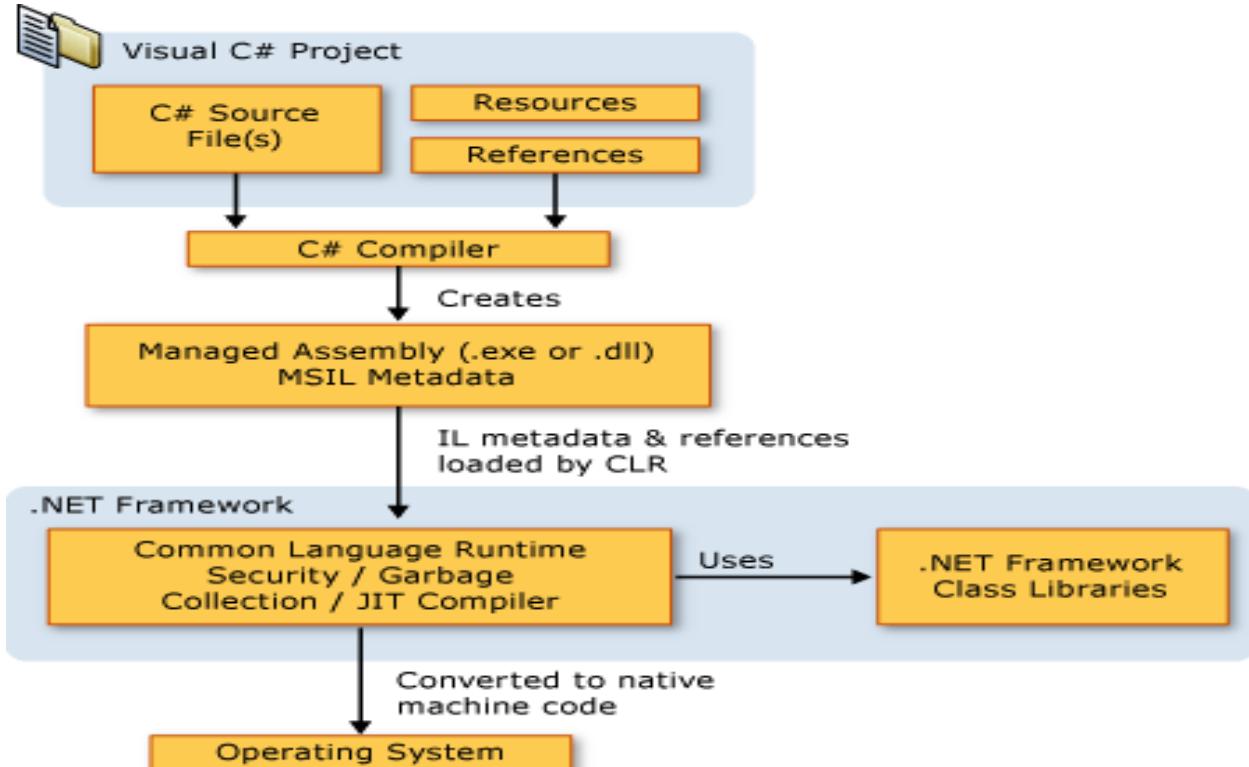
Metadata provides the following major benefits:

1. Self-describing files, common language runtime modules and assemblies are self-describing.

2. Language interoperability, metadata provides all the information required about compiled code for you to inherit a class from a file written in a different language.

MSIL Code or CIL Code: during compilation of any .NET programming languages, the source code is translated into CIL code rather than platform or processor-specific code. CIL is a CPU and platform-independent instruction set that can be executed in any environment supporting the Common Language Infrastructure, such as the .NET runtime on Windows, or the cross-platform Mono runtime.

Compilation and Execution Process of a C# Project



Finalizer

Finalizers are used to destruct objects (**instances**) of classes. A **Finalizer** is also a special method just like a **Constructor**, whereas **Constructors** are called when **instance** of a class is **created**, and **Finalizers** are called when **instance** of a class is **destroyed**. Both will have the same name i.e., the name of class in which they are defined, but to **differentiate** between each other we **prefix Finalizer** with a tilde (~) operator. For Example:

```

class Test
{
    Test()
    {
        //Constructor
    }
    ~Test()
    {
        //Finalizer
    }
}

```

```
}
```

Remarks:

- Finalizers cannot be defined in **structs**. They are only used with **classes**.
- A **finalizer** does not take **modifiers** or have **parameters**.
- A class can only have one **finalizer** and cannot be **inherited** or **overloaded**.
- Finalizers cannot be called. They are invoked **automatically**.

The **programmer** has no control over when the **finalizer** is called because this is determined by the **garbage collector**; **garbage collector** calls the **finalizer** in any of the following cases:

1. Called in the **end** of a programs execution and **destroys** all **instances** that are **associated** with the **program**.
2. In the **middle** of a **program's** execution also the **garbage collector** checks for **instances** that are no longer being used by the **application**. If it considers an **instance** is eligible for **destruction**, it calls the **finalizer** and reclaims the **memory** used to store the **instance**.
3. It is possible to **force - garbage collector** by calling **GC.Collect()** method to check for un-used **instances** and **reclaim** the **memory** used to **store** those **instances**.

Note: we can **force** the **garbage collector** to do clean up by calling the **GC.Collect** method, but in most cases, this should be **avoided** because it may create **performance issues**, i.e., when the **garbage collector** comes into **action** for reclaiming memory of un-used instances, it will **suspend** the execution of programs.

To test this, open a new **Console App. Project** in **.NET Framework** naming it as “**FinalizersProject**”, rename the default class “**Program.cs**” as “**DestDemo1.cs**” using **Solution Explorer** and write the below code over there:

```
internal class DestDemo1
{
    public DestDemo1()
    {
        Console.WriteLine("Instance1 is created.");
    }
    ~DestDemo1()
    {
        Console.WriteLine("Instance1 is destroyed.");
    }
    static void Main(string[] args)
    {
        DestDemo1 d1 = new DestDemo1();
        DestDemo1 d2 = new DestDemo1();
        DestDemo1 d3 = new DestDemo1();
        //d1 = null; d3 = null; GC.Collect(); //(Write all the 3 statements in the same line with comments)
        Console.ReadLine();
    }
}
```

Execute the above program by using **Ctrl + F5** and watch the output of program, first it will call **Constructor** for 3 times because 3 **instances** are created and then waits at **ReadLine** statement to execute; now press **enter** key to finish the execution of **ReadLine**, immediately **finalizer** gets called for 3 times because it is the end of programs **execution**, so all 3 instances associated with the program are **destroyed**. This proves that finalizer is called in the end of a **program's execution**.

Now **un-comment** the **commented** code in **Main** method of above **program** and **re-execute** the program again by using **Ctrl + F5** to watch the difference in output, in this case 2 **instances** are **destroyed** before execution of **ReadLine** because we have marked them as **un-used** by assigning “**null**” and called **Garbage Collector** explicitly, and the third instance is destroyed in end of **program's execution**.

Finalizers and Inheritance: As we are aware that whenever a child class **instance** is created, child class **constructor** will call its parents class **constructor** implicitly, same as this when a child class **instance** is destroyed it will also call its parent classes **finalizer**, but the difference is **constructor** are called in “**Top to Bottom**” hierarchy and **finalizer** are called in “**Bottom to Top**” hierarchy. To test this, add a new class “**DestDemo2.cs**” and write the below code:

```
internal class DestDemo2 : DestDemo1
{
    public DestDemo2()
    {
        Console.WriteLine("Instance2 is created.");
    }
    ~DestDemo2()
    {
        Console.WriteLine("Instance2 is destroyed.");
    }
    static void Main()
    {
        DestDemo2 obj = new DestDemo2();
        Console.ReadLine();
    }
}
```

Conclusion about Finalizers: In general, **C#** does not require as much **memory management**; it is needed when you develop with a **Language** that does not **target** a **runtime** with **garbage collection**, for example **CPP Language**. This is because the **.NET Garbage Collector** which implicitly manages the **allocation** and **release of memory** for your **instances**. However, when your application encapsulates **un-managed resources** such as **Files**, **Databases**, and **Network Connections**, you should use **finalizers** to free those **resources**.

Properties

A **property** is a **member** that provides a flexible mechanism to **read**, **write**, or **compute** the **value** of a **private field**. **Properties** can be used as if they are **public fields**, but they are **special methods** called **accessors**.

Suppose a class is **associated** with any **value** and if we want to **expose** that **value outside** of the **class**, access to that **value** can be given in **4 different ways**:

- I. By storing the **value** under a **public field**, access can be given to that value **outside** of the class, for Example:

```
public class Circle
{
    public double Radius = 12.34;
```

Now by creating the instance of the above class we can get or set a value to the field as following:

```
class TestCircle
{
    static void Main()
    {
        Circle c = new Circle();
        double Radius = c.Radius;           //Getting the old value of Radius
        c.Radius = 56.78;                  //Setting a new value for Radius
    }
}
```

Note: in this approach it will provide **Read/Write** access to the **value** i.e., anyone can get the **old value** of the **field** as well as **anyone** can set with a **new value** for the **field**, so we don't have any **control** on the value.

- II. By **storing** the **value** under a **private field** also we can provide **access** to the **value outside** of the **class** by defining a **property** on that **field**. The advantage in this **approach** is it can provide **access** to the value in **3 different ways**:

1. Only get access (Read Only Property)
2. Only set access (Write Only Property)
3. Both get and set access (Read/Write Property)

Syntax to define a property:

```
[<modifiers>] <type> Name
{
    [ get { -Stmts } ]      //Get Accessor
    [ set { -Stmts } ]      //Set Accessor
}
```

- A **property** is **one** or **two code blocks**, representing a **get** access or **and/or** a **set** accessor.
- The code block for the **get accessor** is executed when the property is **read** and the body of the **get** accessor resembles that of a **method**. It must **return** a **value** of the **property type**. The **get accessor** resembles a **value returning method** without any **parameters**.
- The code block for the **set accessor** is executed when the property is **assigned** with a **new value**. The **set accessor** resembles a **non-value returning method** with parameter, i.e., it uses an **implicit parameter** called "**value**", whose **type** is the same as **property type**.
- A property without a **set accessor** is considered as **read-only**. A property without a **get accessor** is considered as **write-only**. A property that has **both accessors** is considered as **read-write**.

Remarks:

- Properties can be marked as **public**, **private**, **protected**, **private protected**, **internal**, or **protected internal**. These access modifiers define how users of the class can access the property. The get and set accessors for the same property may have different access modifiers. For example, the get may be public to allow read-only access from outside the class, and the set may be private or protected.
 - A property may be declared as a **static** property by using the **static** keyword. This makes the property available to callers at any time, even if no **instance** of the class exists.
 - A property may be marked as a **virtual** property by using the **virtual** keyword, which enables derived classes to **override** the property behavior by using the **override** keyword. A property **overriding** a **virtual** property can also be **sealed**, specifying that for derived classes it is no longer **virtual**.
 - A property can be declared as **abstract** by using the **abstract** keyword, which means that there is no implementation in the class, and **derived classes** must write their own **implementation**.
-

To test properties first add a new code file Cities.cs and write the following code:

```
namespace OOPSProject
{
    public enum Cities
    {
        Bengaluru, Chennai, Delhi, Hyderabad, Kolkata, Mumbai
    }
}
```

Now add a new class Customer.cs and write the following code:

```
public class Customer
```

```
{
    int _Custid;
    bool _Status;
    string _Name, _State;
    double _Balance;
    Cities _City;
    public Customer(int Custid)
    {
        _Custid = Custid;
        _Status = false;
        _Name = "John";
        _Balance = 5000.00;
        _City = 0;
        _State = "Karnataka";
        Country = "India";
    }
    //Read Only Property
    public int Custid
    {
        get { return _Custid; }
    }
}
```

```
}

//Read-Write Property
public bool Status
{
    get { return _Status; }
    set { _Status = value; }
}

//Read-Write Property (With a condition in Set Accessor)
public string Name
{
    get { return _Name; }
    set
    {
        if (_Status)
        {
            _Name = value;
        }
    }
}

//Read-Write Property (With a condition in Get & Set Accessor)
public double Balance
{
    get
    {
        if (_Status)
        {
            return _Balance;
        }
        else
        {
            return 0;
        }
    }
    set
    {
        if (_Status)
        {
            if (value >= 500)
            {
                _Balance = value;
            }
        }
    }
}

//Read-Write Property (Enumerated Property)
```

```

public Cities City
{
    get { return _City; }
    set
    {
        if(_Status)
        {
            _City = value;
        }
    }
}

//Read-Write Property (With a different scope to each property accessor (C# 2.0))
public string State
{
    get { return _State; }
    protected set
    {
        if(_Status)
        {
            _State = value;
        }
    }
}

//Read-Write Property (Automatic or Auto-Implemented property (C# 3.0))
public string Country
{
    get;
    private set;
}

//Read-Only Property (Auto property initializer (C# 6.0))
public string Continent { get; } = "Asia";
}

```

Note: The contextual keyword **value** is used in the **set accessor** in ordinary property declarations. It is like an **input** parameter of a **method**. The word **value** references the value that client code is attempting to assign to the property.

Enumerated Property: It is a property that provides with a set of constants to choose from, for example **BackgroundColor** property of the **Console** class that provides with a list of **constant** colors to choose from, under an **Enum ConsoleColor**. E.g.: `Console.BackgroundColor = ConsoleColor.Blue;` An **Enum** is a distinct type that consists of a set of **named constants** called the **enumerator** list. Usually, it is best to define an **Enum** directly within a **namespace** so that all classes in the **namespace** can access it with equal convenience. However, an **Enum** can also be nested within a **class** or **structure**.

Syntax to define an Enum:

```
[<modifiers>] enum <Name>
{
```

```

<List of named constant values>
}
public enum Days
{
    Monday, Tuesday, Wednesday, Thursday, Friday
}

```

Note: By default, the first value is represented with an index 0, and the value of each successive enumerator is increased by 1. For example, in the above enumeration, Monday is 0, Tuesday is 1, Wednesday is 2, and so forth.

To define an Enumerated Property, adopt the following process:

Step 1: define an **Enum** with the list of constants we want to provide for the property to choose.

E.g.: public enum Days { Monday, Tuesday, Wednesday, Thursday, Friday };

Step 2: declare a **field** of type **Enum** on which we want to define a property.

E.g.: Days _Day = 0; or Days _Day = Days.Monday; or Days _Day = (Days)0;

Step 3: now define a **property** on the **Enum** field for providing access to its values.

```

public Days Day
{
    get { return _Day; }
    set { _Day = value; }
}

```

Auto-Implemented properties: In C# 3.0 and later, auto-implemented properties make property-declaration more concise when **no additional logic is required in the property accessors**. E.g.: **Country** property in our customer class, but up to **CSharp 5.0** it is important to remember that auto-implemented properties must contain both **get** and **set blocks** either with the same access modifier or different also whereas from **CSharp 6.0** it's not mandatory because of a new feature called "**Auto Property Initializer**", which allows to initialize a property at declaration time.

In our Customer class the Country property we have defined can be implemented as below also:

E.g.: public string Country { get; } = "India";

To consume the properties, we have defined above add a new class TestCustomer.cs and write the following:

```

internal class TestCustomer
{
    static void Main()
    {
        Customer obj = new Customer(1001);
        Console.WriteLine("Custid: " + obj.Custid + "\n");
        //obj.Custid = 1005; //Invalid, because the property is defined read only

        if (obj.Status)
            Console.WriteLine("Customer Status: Active");
        else
    }
}

```

```

Console.WriteLine("Customer Status: In-Active");
Console.WriteLine("Customer Name: " + obj.Name);
obj.Name += " Smith"; //Update fails because status is in-active
Console.WriteLine("Name when update failed: " + obj.Name);
Console.WriteLine("Balance when status is in-active: " + obj.Balance + "\n");

obj.Status = true; //Activating the status
if (obj.Status)
    Console.WriteLine("Customer Status: Active");
else
    Console.WriteLine("Customer Status: In-Active");
Console.WriteLine("Customer Name: " + obj.Name);
obj.Name += " Smith"; //Update succeds because status is in-active
Console.WriteLine("Name when update succeeded: " + obj.Name);
Console.WriteLine("Balance when status is active: " + obj.Balance + "\n");

obj.Balance -= 4600; //Transaction failed
Console.WriteLine("Balance when transaction failed: " + obj.Balance);
obj.Balance -= 4500; //Transaction succeds
Console.WriteLine("Balance when transaction succeeded: " + obj.Balance + "\n");

Console.WriteLine($"Customer City: {obj.City}");
obj.City = Cities.Hyderabad;
Console.WriteLine($"Modified City: {obj.City}");

Console.WriteLine("Customer State: " + obj.State);
//obj.State = "Telangana"; //Invalid because set accessor is accessible only to child classes

Console.WriteLine("Customer Country: " + obj.Country);
Console.WriteLine("Customer Continent: " + obj.Continent);
Console.ReadLine();
}
}

```

Object Initializers (Introduced in C# 3.0)

Object initializers let you assign values to any accessible **properties** of an **instance** at creation time without having to **explicitly invoke a parameterized constructor**. You can use **object initializers** to initialize type objects in a **declarative manner** without explicitly invoking a **constructor** for the **type**. Object Initializers will use the **default constructor** for initializing **fields** thru **properties**.

To test these, add a new Code File naming it as “TestStudent.cs” and write the following code in it:

```

namespace OOPSProject
{
    public class Student
    {
        int? _Id, _Class;

```

```

string? _Name;
float? _Marks, _Fees;
public int? Id
{
    get { return _Id; }
    set { _Id = value; }
}
public int? Class
{
    get { return _Class; }
    set { _Class = value; }
}
public string? Name
{
    get { return _Name; }
    set { _Name = value; }
}
public float? Marks
{
    get { return _Marks; }
    set { _Marks = value; }
}
public float? Fees
{
    get { return _Fees; }
    set { _Fees = value; }
}
public override string ToString()
{
    return "Id: " + _Id + "\nName: " + _Name + "\nClass: " + _Class + "\nMarks: " + _Marks + "\nFees: " + _Fees;
}
}

internal class TestStudent {
static void Main() {
    Student s1 = new Student { Id = 101, Name = "Raju", Class = 10, Marks = 575.00f, Fees = 5000.00f };
    Student s2 = new Student { Id = 102, Name = "Vijay", Class = 10 };
    Student s3 = new Student { Id = 103, Marks = 560.00f, Fees = 5000.00f };
    Student s4 = new Student { Id = 104, Class = 10, Fees = 5000.00f };
    Student s5 = new Student { Id = 105, Name = "Raju", Marks = 575.00f };
    Student s6 = new Student { Id = 106, Class = 10, Marks = 575.00f };

    Console.WriteLine(s1); Console.WriteLine(s2); Console.WriteLine(s3);
    Console.WriteLine(s4); Console.WriteLine(s5); Console.WriteLine(s6);
    Console.ReadLine();
}
}

```

```
}
```

Indexers

Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble properties except that their accessors take parameters. Indexers are syntactic conveniences that enable you to create a class or struct that client applications can access just as an array. Defining an indexer allows you to create classes that act like “virtual arrays”. Instances of that class or structure can be accessed using the [] array access operator. Defining an indexer in C# is like defining operator “[]” in C++ but is considerably simpler and more flexible. For classes or structure that encapsulate array or collection - like functionality, defining an indexer allows the users of that class or structure to use the array syntax to access the class or structure. An indexer doesn't have a specific name like a property it is defined by using “this” keyword.

Syntax to define Indexer:

```
[<modifiers>] <type> this[<Parameter List>]
{
    [ get { -Stmts } ] //Get Accessor
    [ set { -Stmts } ] //Set Accessor
}
```

Indexers Overview:

- “this” keyword is used to define the indexers.
- The out and ref keyword are not allowed on parameters.
- A get accessor returns a value. A set accessor assigns a value.
- The value keyword is only used to define the value being assigned by the set indexer.
- Indexers do not have to be indexed by integer value; it is up to you how to define the look-up mechanism.
- Indexers can be overloaded.
- Indexers can't be defined as static.
- Indexers can have more than one formal parameter, for example, accessing a two-dimensional array.

To test indexers, add a Code File under the project naming it as “TestEmployee.cs” and write the below code in it:

```
namespace OOPSProject
{
    public class Employee
    {
        int? _Id;
        string? _Name, _Job;
        double? _Salary;
        bool? _Status;
        public Employee(int Id)
```

```

{
    _Id = Id;
    _Name = "Nicholas";
    _Job = "Manager";
    _Salary = 50000.00;
    _Status = true;
}
public object? this[int Index]
{
    get
    {
        if (Index == 1)
            return _Id;
        else if (Index == 2)
            return _Name;
        else if (Index == 3)
            return _Job;
        else if (Index == 4)
            return _Salary;
        else if (Index == 5)
            return _Status;
        else
            return null;
    }
    set
    {
        if(Index == 2)
            _Name = (string?)value;
        else if(Index == 3)
            _Job = (string?)value;
        else if(Index == 4)
            _Salary = (double?)value;
        else if(Index == 5)
            _Status = (bool?)value;
    }
}
public object? this[string Key]
{
    get
    {
        if (Key.ToUpper() == "ID")
            return _Id;
        else if (Key.ToUpper() == "NAME")
            return _Name;
        else if (Key.ToUpper() == "JOB")
            return _Job;
    }
}

```

```

        else if (Key.ToUpper() == "SALARY")
            return _Salary;
        else if (Key.ToUpper() == "STATUS")
            return _Status;
        else
            return null;
    }
    set
    {
        if (Key.ToLower() == "name")
            _Name = (string?)value;
        else if (Key.ToLower() == "job")
            _Job = (string?)value;
        else if (Key.ToLower() == "salary")
            _Salary = (double?)value;
        else if (Key.ToLower() == "status")
            _Status = (bool?)value;
    }
}
}

internal class TestEmployee
{
    static void Main()
    {
        Employee Emp = new Employee(1005);
        Console.WriteLine("Employee ID: " + Emp[1]);
        Console.WriteLine("Employee Name: " + Emp[2]);
        Console.WriteLine("Employee Job: " + Emp[3]);
        Console.WriteLine("Employee Salary: " + Emp[4]);
        Console.WriteLine("Employee Status: " + Emp[5]);
        Console.WriteLine();

        Emp["Id"] = 1010; //Can't assigned with new value, because we have not defined setter for ID
        Emp[3] = "Sr. Manager";
        Emp["Salary"] = 75000.00;

        Console.WriteLine("Employee ID: " + Emp["Id"]);
        Console.WriteLine("Employee Name: " + Emp["name"]);
        Console.WriteLine("Employee Job: " + Emp["JOB"]);
        Console.WriteLine("Employee Salary: " + Emp["SaLaRy"]);
        Console.WriteLine("Employee Status: " + Emp["Status"]);
        Console.ReadLine();
    }
}

```

Deconstructor

These are newly introduced in C# 7.0 which can also be used to provide access to the values or expose the values associated with a class to the outside environment, apart from **public fields**, **properties**, and **indexers**. **Deconstructor** is a special method with the name “**Deconstruct**” that is defined under the class to expose (**Read Only**) the attributes of a class and this will be defined with a code that is **reverse** to a **constructor**.

To understand **Deconstructor**, add a code file in our project naming it as “**TestTeacher.cs**” and write the below code in it:

```
namespace OOPSProject
{
    public class Teacher
    {
        int? Id;
        string? Name, Subject, Designation;
        double? Salary;
        public Teacher(int? Id, string? Name, string? Subject, string? Designation, double? Salary)
        {
            this.Id = Id;
            this.Name = Name;
            this.Subject = Subject;
            this.Designation = Designation;
            this.Salary = Salary;
        }
        public void Deconstruct(out int? Id, out string? Name, out string? Subject, out string? Designation, out double? Salary)
        {
            Id = this.Id;
            Name = this.Name;
            Subject = this.Subject;
            Designation = this.Designation;
            Salary = this.Salary;
        }
    }
    class TestTeacher
    {
        static void Main()
        {
            Teacher obj = new Teacher(1005, "Suresh", "English", "Lecturer", 25000.00);
            (int? Id1, string? Name1, string? Subject1, string? Designation1, double? Salary1) = obj;
            Console.WriteLine("Teacher Id: " + Id1);
            Console.WriteLine("Teacher Name: " + Name1);
            Console.WriteLine("Teacher Subject: " + Subject1);
            Console.WriteLine("Teacher Designation: " + Designation1);
            Console.WriteLine("Teacher Salary: " + Salary1 + "\n");
            Console.ReadLine();
        }
    }
}
```

```
    }
}
}
```

In the above case “**Deconstruct**” (name cannot be changed) is a special method which will expose the attributes of **Teacher** class. We can capture the values exposed by “**Deconstructors**” by using **Tuples**, through the instance of class we have created.

We can even capture the values as below:

E.g.: `(var Id2, var Name2, var Subject2, var Designation2, var Salary2) = obj;`

The above statement can be implemented as following also:

E.g.: `var (Id2, Name2, Subject2, Designation2, Salary2) = obj;`

Now you print the above values as below and to test that add the below code in the **Main** method of “**TestTeacher**” class just above the **ReadLine** method.

```
var (Id2, Name2, Subject2, Designation2, Salary2) = obj;
Console.WriteLine("Teacher Id: " + Id2);
Console.WriteLine("Teacher Name: " + Name2);
Console.WriteLine("Teacher Subject: " + Subject2);
Console.WriteLine("Teacher Designation: " + Designation2);
Console.WriteLine("Teacher Salary: " + Salary2 + "\n");
```

Note: Deconstructor will provide **read-only** access to the **attributes** of a **class**.

We can also overload **Deconstructors** to access specific values from the list of attributes and to test that add the following **Deconstructor** in the **Teacher** class.

```
public void Deconstruct(out int? Id, out string? Name, out string? Subject)
{
    Id = this.Id;
    Name = this.Name;
    Subject = this.Subject;
}
```

Now we can capture only those 3 values and to test that add the below code in the **Main** method of “**TestTeacher**” class just above the **ReadLine** method.

```
var (Id3, Name3, Subject3) = obj;
Console.WriteLine("Teacher Id: " + Id3);
Console.WriteLine("Teacher Name: " + Name3);
Console.WriteLine("Teacher Subject: " + Subject3 + "\n");
```

Without Overloading the **Deconstructors** also we can access required attribute values by just putting “_” at the place whose values we don’t want to access, and to test this Add the below code in the **Main** method of **TestTeacher** class just above the **ReadLine** method.

```
var (Id4, _, Subject4, _, Salary4) = obj;
Console.WriteLine("Teacher Id: " + Id4);
```

```

Console.WriteLine("Teacher Subject: " + Subject4);
Console.WriteLine("Teacher Salary: " + Salary4 + "\n");

var(Id5, __ Designation5, Salary5) = obj;
Console.WriteLine("Teacher Id: " + Id5);
Console.WriteLine("Teacher Designation: " + Designation5);
Console.WriteLine("Teacher Salary: " + Salary5 + "\n");

```

Exceptions and Exception Handling

In the development of an application, we will be coming across 2 different types of errors, like:

- **Compile time errors.**
- **Runtime errors.**

Errors which occur in a program due to syntactical mistakes at the time of program compilation are known as compile time errors and these are not considered to be dangerous.

Errors which occur in a program while the execution of a program is taking place are known as runtime errors, which can occur due to various reasons like wrong implementation of logic, wrong input supplied to the program, missing of required resources etc. Runtime errors are dangerous because when they occur under the program, the program terminates abnormally at the same line where the error got occurred without executing the next lines of code. To test this, add a new class naming it as **ExceptionDemo.cs** and write the following code:

```

internal class ExceptionDemo
{
    static void Main()
    {
        Console.Write("Enter 1st number: ");
        int x = int.Parse(Console.ReadLine());
        Console.Write("Enter 2nd number: ");
        int y = int.Parse(Console.ReadLine());
        int z = x / y;
        Console.WriteLine("The result of division is: " + z);
        Console.WriteLine("End of the Program.");
    }
}

```

Execute the above program by using Ctrl + F5, and here there are chances of getting few runtime errors under the program, to check them enter the value for y as '0' or enter character input for x or y values, and in both cases when an error got occurred program gets terminated abnormal on the same line where error got occurred.

Exception: In C#, errors in the program at run time are caused through the program by using a mechanism called Exceptions. Exceptions are classes derived from class `Exception` of `System` namespace. Exceptions can be thrown by the .NET Framework CLR (Common Language Runtime) when basic operations fail or by code in a program. Throwing an exception involves creating an instance of an Exception-derived class, and then throwing that instance by using the `throw` keyword. There are so many Exception classes under the Framework Class Library where each class is defined representing a different type of error that occurs under the program, for example: `FormatException`, `NullReferenceException`, `IndexOutOfRangeException`, `ArithmetException` etc.

Exceptions are basically 2 types like SystemExceptions and ApplicationExceptions. System Exceptions are pre-defined exceptions that are fatal errors which occur on some pre-defined error conditions like DivideByZero, FormatException, and NullReferenceException etc. ApplicationExceptions are non-fatal errors i.e. these are errors that are caused by the programs explicitly. Whatever the exception it is every class is a sub class of class Exception only and the hierarchy of these exception classes will be as following:

```
② Exception
  ② SystemException
    ② FormatException
    ② NullReferenceException
    ② IndexOutOfRangeException
    ② ArithmeticException
      ② DivideByZeroException
      ② OverflowException
  ② ApplicationException
```

Exception Handling: It is a process of stopping the abnormal termination of a program whenever a runtime error occurs under the program; if exceptions are handled under the program, we will be having the following benefits:

1. As abnormal termination is stopped, statements that are not related with the error can be still executed.
2. We can also take any corrective actions which can resolve the problems that may occur due to the errors.
3. We can display user friendly error messages to end users in place of pre-defined error messages.

How to handle an Exception: to handle an exception we need to enclose the code of the program under some special blocks known as try and catch blocks which should be used as following:

```
try
{
  -Statement's where there is a chance of getting runtime errors.
  -Statement's which should not execute when the error occurs.
}
catch(<Exception Class Name> [<Variable>])
{
  -Statement's which should execute only when the error occurs.
}
[---<multiple catch blocks if required>---]
```

To test handling exceptions, add a new class TryCatchDemo.cs and write the following code:

```
internal class TryCatchDemo
{
  static void Main()
  {
    try
    {
      Console.Write("Enter 1st number: ");
```

```

int x = int.Parse(Console.ReadLine());
Console.Write("Enter 2nd number: ");
int y = int.Parse(Console.ReadLine());
int z = x / y;
Console.WriteLine("The result of division is: " + z);
}
catch(DivideByZeroException)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Value of divisor can't be zero.");
    Console.ForegroundColor = ConsoleColor.White;
}
catch (FormatException)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Input values must be integers.");
    Console.ForegroundColor = ConsoleColor.White;
}
catch(Exception ex)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(ex.Message);
    Console.ForegroundColor = ConsoleColor.White;
}
Console.WriteLine("End of the Program.");
}
}

```

If we enclose the code under try and catch blocks the execution of program will take place as following:

- If all the statements under try block are successfully executed (i.e., no error in the program), from the last statement of try the control directly jumps to the first statement which is present after all the catch blocks.
- If any statement under try causes an error from that line, without executing any other lines of code in try, control directly jumps to catch blocks searching for a catch block to handle the error:
- If a catch block is available that can handle the exception, then exceptions are caught by that catch block, executes the code inside of that catch block and from there again jumps to the first statement which is present after all the catch blocks.
- If a catch block is not available to handle that exception which got occurred, abnormal termination takes place again on that line.

Note: `Message` is a property under the `Exception` class which gets the error message associated with the exception that got occurred under the program, this property was defined as `virtual` under the class `Exception` and `overridden` under all the child classes of class `Exception` as per their requirement, that is the reason why when we call `ex.Message` under the last catch block, even if “`ex`” is the reference of parent class, it will get the error message that is associated with the child exception class but not of itself because we have already learnt in overriding that “parent’s reference which is created by using child classes instance will call child classes overridden members” i.e., nothing but `dynamic polymorphism`.

Finally Block: this is another block of code that can be paired with try along with catch or without catch also and the speciality of this block is, code written under this block gets executed at **any cost** i.e., when an exception got occurred under the program or an exception did not occur under the program. All statements under try gets executed only when there is no exception under the program and statements under catch block will be executed only when there is exception under the program whereas code under finally block gets executed in both the cases.

To test finally block add a new class “FinallyDemo.cs” and write the following code:

```
internal class FinallyDemo
{
    static void Main()
    {
        try
        {
            Console.Write("Enter 1st number: ");
            int x = int.Parse(Console.ReadLine());
            Console.Write("Enter 2nd number: ");
            int y = int.Parse(Console.ReadLine());
            if(y == 1)
            {
                return;
            }
            int z = x / y;
            Console.WriteLine("The result of division is: " + z);
        }
        catch (Exception ex)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(ex.Message);
            Console.ForegroundColor = ConsoleColor.White;
        }
        finally
        {
            Console.ForegroundColor = ConsoleColor.Blue;
            Console.WriteLine("Finally block got executed.");
            Console.ForegroundColor = ConsoleColor.White;
        }
        Console.WriteLine("End of the Program.");
    }
}
```

Execute the above program for 2 times, first time by giving input which doesn't cause any error and second time by giving the input which causes an error and check the output where in both the cases **finally block** is executed.

In both the cases not only finally block along with it “**End of the program.**” statement also gets executed, now test the program for the third time by giving the divisor value i.e., value to **y** as 1, so that, the if condition in

the try block gets satisfied and return statement gets executed. As we are aware that `return` is a `jump` statement which jumps out of the method in execution, but in this case, it will jump out only after executing the `finally` block of the method because once the control enters `try`, we cannot stop the execution of `finally` block.

Note: try, catch, and finally blocks can be used in 3 different combinations like:

- I. **try and catch**: in this case exceptions that occur in the program are caught by the catch block so abnormal termination will not take place.
- II. **try, catch and finally**: in this case behavior will be same as above but along with it finally block keeps executing in any situation.
- III. **try and finally**: in this case exceptions that occur in the program are not caught because there is no catch block so abnormal termination will take place but still the code under finally block gets executed.

Note: to test `try & finally`, comment the `catch block` in the previous program and execute the program again, now when there is any `runtime error`, exception occurs and program gets `abnormally terminated` but in this case also we see `finally block` getting `executed`.

Application Exceptions: these are non-fatal application errors i.e.; these are errors that are caused by the programs explicitly. Application exceptions are generally raised by programmers under their programs basing on their own error conditions, for example in a division program we don't want the divisor value to be an odd number. If a programmer wants to raise an exception explicitly under his program, he needs to do 2 things under the program.

1. Create the instance of any exception class.
2. Throw that instance by using throw statement. E.g.: `throw <instance of exception class>`

While creating an Exception class instance to throw explicitly we are provided with different options in choosing which exception class instance must be created to throw, like:

1. If any pre-defined `Exception` class is matching with our requirement, then we can create `instance` of that class and `throw`.
2. We can create instance of a pre-defined class i.e., `ApplicationException` by passing the error message that has to be displayed when the error got occurred as a `parameter` to the class `constructor` and then throw that instance.

E.g., `ApplicationException ex = new ApplicationException ("<error message>");`

`throw ex;`

or

`throw new ApplicationException ("<error message>");`

3. We can also define our own exception class, create instance of that class, and throw it when required. If we want to define a new exception class, we need to follow the below process:

- I. Define a new class inheriting from any pre-defined `Exception` class (but `ApplicationException` is preferred choice as we are dealing with application exceptions) so that the new class also is an exception.
- II. Override the `Message` property inherited from parent by providing the required error message.

To test this first add a new class under the project naming it DivideByOddNoException.cs and write the below:

```
public class DivideByOddNoException : ApplicationException
{
    public override string Message
    {
        get
        {
            return "Attempted to divide by odd number.";
        }
    }
}
```

Add a new class ThrowDemo.cs and write the below code:

```
internal class ThrowDemo
{
    static void Main()
    {
        Console.Write("Enter 1st number: ");
        int x = int.Parse(Console.ReadLine());
        Console.Write("Enter 2nd number: ");
        int y = int.Parse(Console.ReadLine());
        if(y % 2 > 0)
        {
            throw new ApplicationException("Divisor can't be an odd number.");
            //throw new DivideByOddNoException();
        }
        int z = x / y;
        Console.WriteLine("The result of division is: " + z);
        Console.WriteLine("End of the Program.");
    }
}
```

Test the above program for the first time by giving the divisor value as an odd number and now **ApplicationException** will raise and displays the error message “Divisor value should not be an odd number.”.

Now comment the **first throw statement** and **uncomment** the **second throw statement** so that when the divisor value is an **odd** number **DivideByOddNoException** will raise and displays the error message “Attempted to divide by odd number.”.

Delegates

Delegate is a **type** which holds the method(s) reference in an **object**. It is also referred to as a **type safe function pointer**. Delegates are roughly like **function-pointers** in **C++**; however, delegates are **type-safe** and **secure**.

A delegate `instance` can encapsulate a `static` or a `non-static` method also and `call` that method for execution. Effective use of a `delegate` improves the `performance` of applications.

Methods can be called in 2 different ways in C#, those are:

1. Using instance of a class if it is non-static and name of the class if it is static.
2. Using a delegate (either static or non-static).

To call a method by using delegate we need to adopt the following process:

1. Define a delegate.
2. Instantiate the delegate.
3. Call the delegate by passing required parameter values.

Syntax to define a Delegate:

```
[<modifiers>] delegate void | <type> Name([<Parameter List>])
```

Note: while defining a `delegate` you should follow the same `signature` of the method i.e., `parameters` of `delegate` should be same as the `parameters` of method and `return types` of `delegates` should be same as the `return types` of method, we want to call by using the `delegate`.

```
public void AddNums(int x, int y)
{
    Console.WriteLine(x + y);
}
public delegate void AddDel(int x, int y);

public static string SayHello(string name)
{
    return "Hello " + name;
}
public delegate string SayDel(string name);
```

Instantiate the delegate: In this process we create the `instance` of the `delegate` and bind the `method` we want to call by using the `delegate` to the `delegate`.

```
AddDel ad = new AddDel(AddNums);      or      AddDel ad = AddNums;
SayDel sd = new SayDel(SayHello);      or      SayDel sd = SayHello;
```

Calling the delegate: Call the `delegate` by passing required parameter values, so that the `method` which is bound with `delegate` gets executed.

```
ad(10, 20); ad(30, 40) ad(50, 60)
```

```
string s1 = sd("Raju"); string s2 = sd("Suneetha"); string s3 = sd("Srinivas");
```

Where to define a delegate?

Ans: Delegates can be defined either in a `class/structure` or with in a `namespace` just like we define other `types`.

Add a code file under the project naming it as Delegates.cs and write the following code:

```

namespace OOPSProject
{
    public delegate void MathDelegate(int x, int y);
    public delegate string WishDelegate(string str);
    public delegate void CalculatorDelegate(int a, int b, int c);
}

Add a class DelDemo1.cs under the project and write the following code:
internal class DelDemo1
{
    public void AddNums(int x, int y, int z)
    {
        Console.WriteLine($"Sum of given 3 no's is: {x + y + z}");
    }
    public static string SayHello(string Name)
    {
        return $"Hello {Name}, have a nice day!";
    }
    static void Main()
    {
        DelDemo1 obj = new DelDemo1();

        CalculatorDelegate cd = obj.AddNums;
        cd(10, 20, 30); cd(40, 50, 60); cd(70, 80, 90);

        WishDelegate wd = DelDemo1.SayHello;
        Console.WriteLine(wd("Raju"));
        Console.WriteLine(wd("Vijay"));
        Console.WriteLine(wd("Naresh"));

        Console.ReadLine();
    }
}

```

Multicast Delegate: It is a **delegate** who holds the **reference** of more than **one method**. Multicast delegates must contain only methods that return **void**. If we want to call multiple methods using a single delegate all the methods should have the same **Parameter types**. To test this, add a new class **DelDemo2.cs** under the project and write the following code:

```

internal class DelDemo2
{
    public void Add(int x, int y)
    {
        Console.WriteLine($"Add: {x + y}");
    }
    public void Sub(int x, int y)
    {

```

```

        Console.WriteLine($"Sub: {x - y}");
    }
    public void Mul(int x, int y)
    {
        Console.WriteLine($"Mul: {x * y}");
    }
    public void Div(int x, int y)
    {
        Console.WriteLine($"Div: {x / y}");
    }
    static void Main()
    {
        DelDemo2 obj = new DelDemo2();
        MathDelegate md = obj.Add;
        md += obj.Sub; md += obj.Mul; md += obj.Div;

        md(100, 25);
        Console.WriteLine();
        md(760, 20);
        Console.WriteLine();
        md -= obj.Mul;
        md(930, 15);
        Console.ReadLine();
    }
}

```

Anonymous Methods (Introduced in C# 2.0): In versions of C# before 2.0, the only way to instantiate a delegate was to use **named** methods. C# 2.0 introduced **anonymous methods** which provide a **technique** to pass a code block as a **delegate parameter**. **Anonymous** methods are basically methods without a **name**. An **anonymous** method is **in-line unnamed** method in the code. It is created using the **delegate** keyword and doesn't require **modifiers**, **name**, and **return type**. Hence, we can say an **anonymous** method has only **body** without **name**, **return type** and **optional parameters**. An **anonymous** method behaves like a regular method and allows us to write **in-line** code in place of **explicit named** methods. To test this, add a new class **DelDemo3.cs** and write the following code:

```

internal class DelDemo3
{
    static void Main()
    {
        CalculatorDelegate cd = delegate (int a, int b, int c)
        {
            Console.WriteLine($"Product of given numbers: {a * b * c}");
        };
        cd(10, 20, 30); cd(40, 50, 60); cd(70, 80, 90);

        WishDelegate wd = delegate (string user)
        {

```

```

        return $"Hello {user}, welcome to the application.";

    };

    Console.WriteLine(wd("Raju"));
    Console.WriteLine(wd("Pooja"));
    Console.WriteLine(wd("Praveen"));
    Console.ReadLine();
}
}

```

Lambda Expression (Introduced in CSharp 3.0): While **Anonymous Methods** were a new feature in 2.0; **Lambda Expressions** are simply an improvement to syntax when using **Anonymous** method. Lambda Operator “=>” was introduced so that there is no longer a need to use the **delegate** keyword or provide the type of the parameters. The types can usually be **inferred** by **compiler** from usage based on the **delegate**. To test this, add a new class **DelDemo4.cs** and write the following code:

```

internal class DelDemo4
{
    static void Main()
    {
        CalculatorDelegate cd = (a, b, c) =>
        {
            Console.WriteLine($"Product of given numbers: {a * b * c}");
        };
        cd(10, 20, 30);
        cd(40, 50, 60);
        cd(70, 80, 90);

        WishDelegate wd = user =>
        {
            return $"Hello {user}, welcome to the application.";
        };
        Console.WriteLine(wd("Raju"));
        Console.WriteLine(wd("Pooja"));
        Console.WriteLine(wd("Praveen"));
        Console.ReadLine();
    }
}

```

Expression Bodied Members (Introduced in C# 6.0 & 7.0): Expression body definitions let you provide a member’s implementation in a very concise, readable form. You can use an expression body definition whenever the logic for any supported member consists of a **single** expression.

An expression body definition has the following general syntax:

member => expression;

To test this, add a new class DelDemo5.cs and write the following code:

```
internal class DelDemo5
{
    static void Main()
    {
        CalculatorDelegate cd = (a, b, c) => Console.WriteLine($"Product of given numbers: {a * b * c}");
        cd(10, 20, 30);
        cd(40, 50, 60);
        cd(70, 80, 90);

        WishDelegate wd = user => $"Hello {user}, welcome to the application.";
        Console.WriteLine(wd("Raju"));
        Console.WriteLine(wd("Pooja"));
        Console.WriteLine(wd("Praveen"));
        Console.ReadLine();
    }
}
```

Why would we need to write a method without a name is **convenience** i.e., it's a **shorthand** that allows you to write a method in the same place you are going to use it. Especially useful in places where a method is being used **only once** and the method definition are **short**. It saves you the effort of declaring and writing a separate method to the containing class. Benefits are like **reduced typing**, i.e., no need to specify the name of the method, its return type, and its access modifier as well as when reading the code, you don't need to look **elsewhere** for the method definition. Anonymous methods should be **short**; a **complex** definition makes calling code difficult to read.

Support for **expression body definitions** was introduced for **methods** and **read-only properties** in **C# 6.0** and was expanded in **C# 7.0**. Expression body definitions can be used with type members listed in following table:

<u>Member</u>	<u>Supported as of...</u>
Method	C# 6.0
Read-only property	C# 6.0
Property	C# 7.0
Constructor	C# 7.0
Finalizer	C# 7.0
Indexer	C# 7.0
Deconstructor	C# 7.0

For example, below is a class defined without using expression bodied members:

```
internal class Circle1
{
    double _Radius;
    const float _Pi = 3.14f;
    public Circle1(double Radius)
    {
        _Radius = Radius;
```

```

}

public void Deconstruct(out double Radius)
{
    Radius = _Radius;
}
~Circle1()
{
    Console.WriteLine("Instance is destroyed.");
}
public float Pi
{
    get { return _Pi; }
}
public double Radius
{
    get { return _Radius; }
    set { _Radius = value; }
}
public double GetRadius()
{
    return _Pi * _Radius * _Radius;
}
public double GetPerimeter()
{
    return 2 * _Pi * _Radius;
}
}

```

Above class can be defined as following by using expression bodied members:

```

internal class Circle2
{
    const float _Pi = 3.14f;
    double _Radius;

    public Circle2(double Radius) => _Radius = Radius;           //C# 7.0

    public void Deconstruct(out double Radius) => Radius = _Radius; //C# 7.0

    ~Circle2() => Console.WriteLine("Instance is destroyed.");      //C# 7.0

    public float Pi => _Pi;                                         //C# 6.0

    public double Radius                                            //C# 7.0
    {
        get => _Radius;
        set => _Radius = value;
    }
}
```

```

}

public double GetRadius() => _Pi * _Radius * _Radius;           //C# 6.0

public double GetPerimeter() => 2 * _Pi * _Radius;            //C# 6.0
}

```

Anonymous Types (Introduced in C# 3.0): Anonymous type, as the name suggests, is a type that doesn't have any name. C# allows you to create an instance with the new keyword without defining a class. The implicitly typed variable - "var" or "dynamic" is used to hold the reference of anonymous types.

```

var Emp = new { Id = 1001, Name = "Raju", Job = "Manager", Salary = 25000.00, Status = true };
dynamic Emp = new { Id = 1001, Name = "Raju", Job = "Manager", Salary = 25000.00, Status = true };

```

In the above example, "Emp" is an instance of the anonymous type which is created by using the new keyword and object initializer syntax. It includes 5 properties of different data types. An anonymous type is a temporary type that is inferred based on the data that you include in an object initializer. Properties of anonymous types will be **read-only** properties so you cannot change their values.

Notice that the compiler applies the appropriate type to each property based on the value assigned. For example, **Id** is of **integer** type, **Name** and **Job** are of **string** type, **Salary** is of **double** type and **Status** is of **boolean** type. Internally, the compiler automatically generates the new type for anonymous types. You can check that by calling **GetType()** method on an anonymous type instance which will return the following value:

```
<>f__AnonymousType0`5[System.Int32,System.String,System.String,System.Double,System.Boolean]
```

Remember that Anonymous Types are derived from the **Object** class, and they are **sealed** classes, and all the properties are created as **read only** properties. An anonymous type will always be local to the method where it is defined. Usually, you cannot pass an anonymous type to another method; however, you can pass it to a method that accepts a parameter of **dynamic** type. Anonymous types can be **nested** i.e., an anonymous type can have another anonymous type as a property.

Points to Remember:

- Anonymous type can be defined using the new keyword and object initializer syntax.
- The implicitly typed variable - "var" or "dynamic" keyword, is used to hold an anonymous type.
- Anonymous type is a reference type, and all the properties are read-only.
- The scope of an anonymous type is local to the method where it is defined.

To test anonymous types, add a new **Code File** under the project naming it as "**TestAnonymousTypes.cs**" and write the following code in it:

```

namespace OOPSProject
{
    internal class TestAnonymousTypes
    {
        static void Main()
        {
            var Emp = new { Id = 1001, Name = "Raju", Job = "Manager", Salary = 50000.00, Status = true,

```

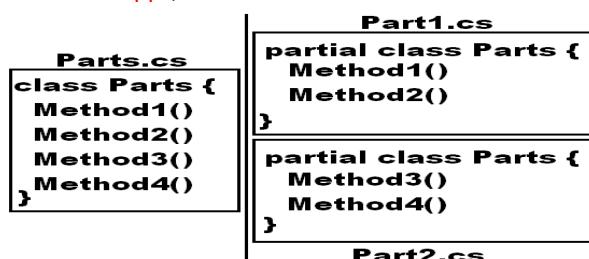
```

        Dept = new { Id = 10, Name = "Sales", Location = "Hyderabad" } );
Console.WriteLine(Emp.GetType() + "\n");
Printer.Print(Emp);
Console.ReadLine();
}
}
internal class Printer
{
    public static void Print(dynamic d)
    {
        Console.WriteLine($"Employee Id: {d.Id}");
        Console.WriteLine($"Employee Name: {d.Name}");
        Console.WriteLine($"Employee Job: {d.Job}");
        Console.WriteLine($"Employee Salary: {d.Salary}");
        Console.WriteLine($"Employee Status: {d.Status}");
        Console.WriteLine($"Department Id: {d.Dept.Id}");
        Console.WriteLine($"Department Name: {d.Dept.Name}");
        Console.WriteLine($"Department Location: {d.Dept.Location}");
    }
}
}

```

Partial Types/Partial Classes (Introduced in C# 2.0): It is possible to split the definition of a class or struct or interface over two or more source files. Each source file contains a section of the type definition, and all parts are combined when the application is compiled. There are several situations when splitting a class definition is desirable like:

- When working on large projects, spreading a type over separate files enable multiple programmers to work on it at the same time.
- Visual Studio uses these partial classes for auto generation of source code in the development of Windows Forms Apps, WPF Apps, Web Forms Apps, and Web Services and so on.



Points to Remember:

- The partial keyword indicates that other parts of the class, struct, or interface can be defined in the namespace.
- All the parts must use the partial keyword.
- All the parts must be available at compile time to form the final type.
- All the parts must have the same accessibility, such as public or internal.
- If any part is declared abstract, then the whole type is considered abstract.
- If any part is declared sealed, then the whole type is considered sealed.

- If any part declares a base type, then the whole type inherits that class.
- Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations.
- Any class, struct, or interface members declared in a partial definition are available to all the other parts.
- The final type is the combination of all the parts at compile time.
- The partial modifier is not available on delegate or enumeration declarations.

To test partial classes, add 2 new code files under the project Part1.cs and Part2.cs and write the below code:

```
namespace OOPSProject
{
    partial class Parts
    {
        public void Method1()
        {
            Console.WriteLine("Part1 - Method1");
        }

        public void Method2()
        {
            Console.WriteLine("Part1 - Method2");
        }
    }
}

namespace OOPSProject
{
    partial class Parts
    {
        public void Method3()
        {
            Console.WriteLine("Part2 - Method3");
        }

        public void Method4()
        {
            Console.WriteLine("Part2 - Method4");
        }
    }
}
```

Now to test the above partial class, add a new class TestParts.cs under the project and write the below code:

```
internal class TestParts
{
    static void Main()
    {
        Parts p = new Parts();
        p.Method1();
        p.Method2();
        p.Method3();
    }
}
```

```
p.Method4();
Console.ReadLine();
}
}
```

Collections

Arrays are simple **data structures** used to store data items of a specific type. Although commonly used, arrays have **limited capabilities**. For example, you must specify an array's **size** at the time of **declaration** and if at **execution** time, you wish to modify it, you must do so manually by **creating** a new **array** or by using **Array** class's **Resize** method, which creates a new **array** and **copies** the **existing** elements into the **new** **array**.

Collections are a set of **pre-packaged data structures** that offer greater **capabilities** than traditional **arrays**. They are **reusable**, **reliable**, **powerful**, and **efficient** and have been carefully **designed** and **tested** to ensure **quality** and **performance**. Collections are like **arrays** but provide **additional functionalities**, such as **dynamic resizing** - they automatically **increase** their size at **execution** time to **accommodate** additional elements, **inserting** of new elements and **removing** of existing elements.

Initially in 2002 .NET introduced so many collection classes under the namespace **System.Collections** (which is defined in the assembly **System.Collections.NonGeneric.dll**) like **Stack**, **Queue**, **LinkedList**, **SortedList**, **ArrayList**, **Hashtable** etc. and you can work out with these classes in your **application** where you need the appropriate behavior.

To work with **Collection** classes, create a new project of type "**Console App**." naming it as "**CollectionsProject**", now under the first-class **Program** write the following code to use the **Stack** class which works on the principle **First in Last Out (FILO)** or **Last in First Out (LIFO)**:

```
using System.Collections;
internal class Program
{
    static void Main(string[] args)
    {
        Stack s = new Stack();

        s.Push('A'); s.Push(100); s.Push(false); s.Push(34.56); s.Push("Hello");

        foreach(object obj in s) {
            Console.Write(obj + " ");
        }
        Console.WriteLine();

        Console.WriteLine(s.Pop());
        foreach (object obj in s) {
            Console.Write(obj + " ");
        }
        Console.WriteLine();
```

```

Console.WriteLine(s.Peek());
foreach (object obj in s) {
    Console.Write(obj + " ");
}
Console.WriteLine();
Console.WriteLine($"No. of items in the Stack: {s.Count}");
s.Clear();
Console.WriteLine($"No. of items in the Stack: {s.Count}");
Console.ReadLine();
}
}

```

Using Queue class which works on the principle First in First Out (FIFO): Add a new class in the project naming it as “**Class1.cs**” and write the below code in it:

```

using System.Collections;
internal class Class1
{
    static void Main()
    {
        Queue q = new Queue();

        q.Enqueue('A'); q.Enqueue(100); q.Enqueue(false); q.Enqueue(34.56); q.Enqueue("Hello");

        foreach(object obj in q) {
            Console.Write(obj + " ");
        }
        Console.WriteLine();

        Console.WriteLine(q.Dequeue());
        foreach (object obj in q) {
            Console.Write(obj + " ");
        }
        Console.ReadLine();
    }
}

```

Auto-Resizing of Collections: The **capacity** of a collection increases dynamically i.e., when we add new **elements** to a Collection the **size** keeps on **incrementing** automatically. Every **collection** class has 3 **constructors** to it and the behavior of **collections** will be as following when the instance is created using different **constructor**:

- i. **Default Constructor:** initializes a new **instance** of the **collection** class that is **empty** and has the **default initial capacity** as **zero** which becomes **4** after adding the **first** element and from then when ever needed the current capacity **doubles**.

- ii. **Collection(int Capacity)**: Initializes a new **instance** of the **collection** class that is **empty** and has the specified **initial capacity**, here also when requirement comes the current capacity **doubles**.
 - iii. **Collection(Collection c)**: This is a **Copy Constructor**. Initializes a new **instance** of the **collection** class that contains elements copied from an **old collection** and that has the same **initial capacity** as the number of elements **copied**, here also when requirement comes current capacity **doubles**.
-

ArrayList: this collection class works same as an **array** but provides **auto resizing**, **inserting**, and **deleting** of items. To work with an **ArrayList**, add a new class in the project naming it as “**Class2.cs**” and write the below code in it:

```
using System.Collections;
internal class Class2
{
    static void Main()
    {
        ArrayList Coll1 = new ArrayList();
        Console.WriteLine($"Initial capacity: {Coll1.Capacity}");

        Coll1.Add('A');
        Console.WriteLine($"Capacity of the collection after adding 1st item: {Coll1.Capacity}");

        Coll1.Add(100); Coll1.Add(false); Coll1.Add(34.56);
        Console.WriteLine($"Capacity of the collection after adding 4th item: {Coll1.Capacity}");

        Coll1.Add("Hello");
        Console.WriteLine($"Capacity of the collection after adding 5th item: {Coll1.Capacity}");

        for (int i=0;i< Coll1.Count;i++ )
        {
            Console.Write(Coll1[i] + " ");
        }
        Console.WriteLine();

        //Coll1.Remove(false);
        //Coll1.RemoveAt(2);
        Coll1.RemoveRange(2, 1);
        foreach(object obj in Coll1)
        {
            Console.Write(obj + " ");
        }
        Console.WriteLine();

        Coll1.Insert(2, true);
        foreach (object obj in Coll1)
        {
            Console.Write(obj + " ");
        }
        Console.WriteLine("\n");

        ArrayList Coll2 = new ArrayList(Coll1);
        foreach (object obj in Coll2)
        {
            Console.Write(obj + " ");
        }
    }
}
```

```

        }
        Console.WriteLine();
        Console.WriteLine($"Initial capacity of new collection: {Coll2.Capacity}");
        Coll2.Add(false);
        Console.WriteLine($"Capacity of new collection after adding new item: {Coll2.Capacity}");
        Coll2.TrimToSize();
        Console.WriteLine($"Capacity of new collection after calling TrimToSize: {Coll2.Capacity}");
        Console.ReadLine();
    }
}

```

Hashtable: it is a **collection** which stores elements in it as "**Key/Value Pairs**" i.e., **ArrayList** also has a **key** to access the **values** under them which is the **index** that starts at "**0**" to number of **elements - 1**, whereas in case of **Hashtable** these **keys** can also be defined by us and can be of any **data type**. To work with Hashtable add a new class in the project naming it as "**Class3.cs**" and write the below code in it:

```

using System.Collections;
internal class Class3
{
    static void Main()
    {
        Hashtable Emp = new Hashtable();
        Emp.Add("Emp-Id", 1001);
        Emp.Add("Emp-Name", "Scott");
        Emp.Add("Job", "CEO");
        Emp.Add("Mgr-Id", null);
        Emp.Add("Salary", 50000.00);
        Emp.Add("Commission", 0.00f);
        Emp.Add("Dept-Id", 10);
        Emp.Add("Dept-Name", "Administration");
        Emp.Add("Location", "Mumbai");
        Emp.Add("Status", true);
        Emp.Add("PAN", "AKYPM 1234K");
        Emp.Add("Aadhar No.", "1234 5678 9012");
        Emp.Add("Mobile", "98392 14256");
        Emp.Add("Home Phone", "2718 6547");
        Emp.Add("Email", "Scott@gmail.com");

        foreach(object key in Emp.Keys)
        {
            Console.WriteLine($"{key}: {Emp[key]}");
        }
        Console.ReadLine();
    }
}

```

Generics: Generics are added in **C# 2.0** introducing to the **.NET Framework** the concept of **type parameters**, which make it possible to design classes, and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter “T” you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, in simple words Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at consumption time. To understand these, add a class naming it as “**GenericMethods.cs**” and write the following code:

```
internal class GenericMethods
{
    public bool AreEqual<T>(T a, T b)
    {
        if (a.Equals(b))
            return true;
        else
            return false;
    }
    static void Main()
    {
        GenericMethods obj = new GenericMethods();
        Console.WriteLine(obj.AreEqual<int>(100, 200));
        Console.WriteLine(obj.AreEqual<bool>(true, true));
        Console.WriteLine(obj.AreEqual<double>(34.56, 87.12));
        Console.WriteLine(obj.AreEqual<string>("Hello", "Hello"));
        Console.ReadLine();
    }
}
```

Just like we are passing Type parameter to methods it is possible to pass them to a class also, to test this add a code file naming it as “**TestGenericClass.cs**” and write the following:

```
namespace CollectionsProject
{
    class Math<T>
    {
        public T Add(T a, T b)
        {
            dynamic d1 = a;
            dynamic d2 = b;
            return d1 + d2;
        }
        public T Sub(T a, T b)
        {
            dynamic d1 = a;
            dynamic d2 = b;
            return d1 - d2;
        }
    }
}
```

```

public T Mul(T a, T b)
{
    dynamic d1 = a;
    dynamic d2 = b;
    return d1 * d2;
}

public T Div(T a, T b)
{
    dynamic d1 = a;
    dynamic d2 = b;
    return d1 / d2;
}
}

internal class TestGenericClass
{
    static void Main()
    {
        Math<int> mi = new Math<int>();
        Console.WriteLine(mi.Add(100, 200));
        Console.WriteLine(mi.Sub(234, 123));
        Console.WriteLine(mi.Mul(12, 46));
        Console.WriteLine(mi.Div(900, 45));
        Console.WriteLine();

        Math<double> md = new Math<double>();
        Console.WriteLine(md.Add(145.35, 12.5));
        Console.WriteLine(md.Sub(45.6, 23.3));
        Console.WriteLine(md.Mul(15.67, 3.4));
        Console.WriteLine(md.Div(168.2, 14.5));
        Console.ReadLine();
    }
}
}

```

Generic Collections: these are also introduced in C# 2.0 which are extension to collections we have been discussing above, in case of collection classes the elements being added in them are of type object, so we can store any type of values in them which requires boxing and un-boxing, whereas in case of generic collections we can store specified type of values which provides type safety. Microsoft has re-implemented all the existing collection classes under a new namespace **System.Collections.Generic** but the main difference is while creating instance of generic collection classes we need to explicitly specify the type of values we want to store under them. In this namespace we have been provided with many classes like classes in **System.Collections** namespace as following:

Stack<T>, Queue<T>, LinkedList<T>, SortedList<T>, List<T>, Dictionary<TKey, TValue>

Note: <T> refers to the **type** of values we want to store under them. For example:

```
Stack<int> si = new Stack<int>();           //Stores integer values only
Stack<float> sf = new Stack<float>();        //Stores float values only
Stack<string> ss = new Stack<string>();       //Stores string values only
```

List: this class is same as **ArrayList** we have discussed under collections above. To work with this **List**, add a new class in the project naming it as “**Class4.cs**” and write the below code in it:

```
internal class Class4
{
    static void Main()
    {
        List<int> Coll = new List<int>();
        Coll.Add(10); Coll.Add(20); Coll.Add(30); Coll.Add(40); Coll.Add(50);

        for(int i=0;i<Coll.Count;i++) {
            Console.Write(Coll[i] + " ");
        }
        Console.WriteLine();

        Coll.Insert(3, 35);
        foreach(int i in Coll) {
            Console.Write(i + " ");
        }
        Console.WriteLine();

        Coll.Remove(30);
        foreach (int i in Coll) {
            Console.Write(i + " ");
        }
        Console.WriteLine();

        Console.ReadLine();
    }
}
```

Dictionary: this class is same as **Hashtable** we have discussed under collections but here while creating the object we need to specify the type for keys as well as for values also, as following:

```
Dictionary< TKey, TValue>
```

To work with Hashtable add a new class in the project naming it as “Class5.cs” and write the below code in it:

```
internal class Class5
{
    static void Main()
    {
        Dictionary<string, object?> Emp = new Dictionary<string, object?>();
        Emp.Add("Emp-Id", 1001);
```

```

        Emp.Add("Emp-Name", "Scott");
        Emp.Add("Job", "CEO");
        Emp.Add("Mgr-Id", null);
        Emp.Add("Salary", 50000.00);
        Emp.Add("Commission", 0.00f);
        Emp.Add("Dept-Id", 10);
        Emp.Add("Dept-Name", "Administration");
        Emp.Add("Location", "Mumbai");
        Emp.Add("Status", true);
        Emp.Add("PAN", "AKYPM 1234K");
        Emp.Add("Adhar No.", "1234 5678 9012");
        Emp.Add("Mobile", "98392 14256");
        Emp.Add("Home Phone", "2718 6547");
        Emp.Add("Email", "Scott@gmail.com");

    foreach(string key in Emp.Keys) {
        Console.WriteLine($"{key}: {Emp[key]}");
    }
    Console.ReadLine();
}
}

```

Collection Initializers: this is a new feature added in C# 3.0 which allows to initialize a collection directly at the time of declaration like an array, as following:

```

List<int> Coll1 = new List<int>() { 10, 20, 30, 40, 50 };
List<string> Coll2 = new List<string>() { "Red", "Blue", "Green", "White", "Yellow" };

```

Add a new class in the project naming it as Class6.cs and write the below code in it:

```

internal class Class6
{
    static void Main()
    {
        //Copying values > 40 from 1 list to another list and arranging them in descending order
        List<int> coll1 = new List<int>() { 13,56,29,98,24,54,79,39,8,42,22,93,6,73,35,67,48,18,61,32,86,15,21,81,2 };
        List<int> coll2 = new List<int>();

        foreach(int i in coll1)
        {
            if(i > 40)
            {
                coll2.Add(i);
            }
        }
        coll2.Sort();
        coll2.Reverse();
    }
}

```

```

        Console.WriteLine(String.Join(", ", coll2));
        Console.ReadLine();
    }
}

```

The above program if used an array, code will be as following (Add a new class Class7.cs and write the below):

```

internal class Class7
{
    static void Main()
    {
        //Copying values > 40 from 1 array to another array and arranging them in descending order
        int[] arr = { 13, 56, 29, 98, 24, 54, 79, 39, 8, 42, 22, 93, 6, 73, 35, 67, 48, 18, 61, 32, 86, 15, 21, 81, 2 };

        int Count = 0, Index = 0;
        foreach(int i in arr)
        {
            if(i > 40)
            {
                Count += 1;
            }
        }
        int[] brr = new int[Count];
        foreach(int i in arr)
        {
            if(i > 40)
            {
                brr[Index] = i;
                Index += 1;
            }
        }

        Array.Sort(brr);
        Array.Reverse(brr);
        Console.WriteLine(String.Join(", ", brr));
        Console.ReadLine();
    }
}

```

In the above 2 programs we are **filtering** the values of a **List** and **Array** which are greater than **40** and then **arranging** them in **descending** order; to do this we have written a **substantial** amount of code which is the traditional process of performing **filters** on **Arrays** and **Collections**.

In **C# 3.0** Microsoft has introduced a new language known as “**LINQ**” much like **SQL** (which we use universally with **Relational Databases** to perform queries). **LINQ** allows you to write query expressions (similar to **SQL Queries**) that can retrieve information from a wide variety of **Data Sources** like **Objects**, **Databases** and **XML**.

Introduction to LINQ: LINQ stands for **Language Integrated Query**. LINQ is a data querying methodology which provides querying capabilities to **.NET** languages with syntax like an **SQL Query**.

LINQ has a great power of **querying** on any **source** of data, where the **Data Source** could be collections of objects (**arrays & collections**), **Database** or **XML Source** and it is divided into 3 parts:

LINQ to Objects:

- Used to perform queries against the in-memory data like an **Array or Collection**.

LINQ to XML (XLinq):

- Used to perform queries against an **XML Source**.

LINQ to Databases: under this we again have 2 options like,

- **LINQ to SQL** is used to perform queries against a **Relation Database**, but only **Microsoft SQL Server**.
- **LINQ to Entities** is used to perform queries against any **Relation Database** like **SQL Server**, **Oracle**, etc.

Advantages of LINQ:

- **LINQ** offers an object-based, language-integrated way to **Query** over data, no matter where that data came from. So, through **LINQ** we can query **Database**, **XML** as well as **Collections & Arrays**.
- **Compile-time** syntax checking.
- It allows to **Query - Collections**, **Arrays**, and **classes** etc. in the **native language** of your application like **VB** or **C#**.

LINQ to Objects

This is designed to write **queries** against the **in-memory data** like an **array** or **collection** and **filter** or **sort** the information present under them. **Syntax** of the query we want to use on **objects** will be as following:

from <alias> in <array name | collection name> [<clauses>] select <alias> | new {<Column List>}

- A LINQ-Query **starts** with **from** and **ends** with **select**.
- In **clauses** we need to use the **alias name** just like we use **column names** in case of **SQL** in case of scalar types.
- Clauses in **LINQ** are **where**, **group by** and **order by**.
- To use **LINQ** in your application first we need to import “**System.Linq**” namespace.

We can write our previous 2 programs where we have **filtered** the data of a **List** or **Array** and arranged in **sorted order** by using **LINQ** and to test that add a new class with the name “**Class8.cs**” and write the below code:

```

internal class Class8
{
    static void Main()
    {
        List<int> coll1 = new List<int>() { 13,56,29,98,24,54,79,39,8,42,22,93,6,73,35,67,48,18,61,32,86,15,21,81,2 };
        var coll2 = from i in coll1 where i > 40 orderby i descending select i;
        Console.WriteLine(String.Join(", ", coll2));

        int[] arr = { 13, 56, 29, 98, 24, 54, 79, 39, 8, 42, 22, 93, 6, 73, 35, 67, 48, 18, 61, 32, 86, 15, 21, 81, 2 };
        var brr = from i in arr where i > 40 orderby i descending select i;
        Console.WriteLine(String.Join(", ", brr));
        Console.ReadLine();
    }
}

```

Note: the values that are returned by a **LINQ** query can be captured by using **implicitly typed local variables**, so in above code “**coll2**” & “**brr**” are implicitly declared **collection/array** that stores the values retrieved by the **Query**.

In **traditional** process of **filtering** data of an **array** or **collection** we have **repetition** statements that filter arrays focusing on the **process** of getting the results i.e., **iterating** through the elements and checking whether they satisfy the desired **criteria**, whereas **LINQ** specifies, not the steps necessary to get the results, but rather the conditions that selected elements must satisfy and this is known as **declarative programming** - as opposed to **imperative programming** (which we've been using so far) in which we specify the **actual steps** to perform a **task**. **Procedural** and **Object-Oriented** Languages are a **subset of imperative**.

The **queries** we have used above specifies that the result should consist of all the **int's** in the **List** or **Array** that are greater than **40**, but it does not specify how to **obtain** the **result**, **C#** compiler **generates** all the necessary code **automatically**, which is one of the great **strengths of LINQ**.

LINQ Providers: The syntax of **LINQ** is built into the **language**, and **LINQ** can be used in many different contexts because of the **libraries** known as **providers**. A **LINQ provider** is a set of classes that implement **LINQ** operations and enable programs to interact with **Data Sources** to perform tasks such as **sorting**, **grouping**, and **filtering** elements. **System.Linq** is the **LINQ Provider** or **Library** that we need for writing **Queries** in our code.

To test writing queries on a Collection, add a new Class naming it as Class9.cs and write the below code in it:

```

internal class Class9
{
    static void Main()
    {
        string[] colors = { "Red", "Blue", "Green", "Black", "White", "Brown", "Orange", "Purple", "Yellow", "Aqua" };

        //Gets the list of all colors as is
        var coll1 = from s in colors select s;
        Console.WriteLine(String.Join(" ", coll1) + "\n");

        //Gets the list of all colors in ascending order
        var coll2 = from s in colors orderby s select s;
    }
}

```

```

Console.WriteLine(String.Join(" ", coll2) + "\n");

//Gets the list of all colors in descending order
var coll3 = from s in colors orderby s descending select s;
Console.WriteLine(String.Join(" ", coll3) + "\n");

//Gets the list of colors whose length is 5 characters
var coll4 = from s in colors where s.Length == 5 select s;
Console.WriteLine(String.Join(" ", coll4) + "\n");

//Getting the list of colors whose name starts with character "B":
var coll5 = from s in colors where s[0] == 'B' select s;
Console.WriteLine(String.Join(" ", coll5));
var coll6 = from s in colors where s.IndexOf("B") == 0 select s;
Console.WriteLine(String.Join(" ", coll6));
var coll7 = from s in colors where s.StartsWith("B") select s;
Console.WriteLine(String.Join(" ", coll7));
var coll8 = from s in colors where s.Substring(0, 1) == "B" select s;
Console.WriteLine(String.Join(" ", coll8) + "\n");

//Getting the list of colors whose name ends with character "e":
var coll9 = from s in colors where s[s.Length - 1] == 'e' select s;
Console.WriteLine(String.Join(" ", coll9));
var coll10 = from s in colors where s.IndexOf("e") == s.Length - 1 select s;
Console.WriteLine(String.Join(" ", coll10));
var coll11 = from s in colors where s.EndsWith("e") select s;
Console.WriteLine(String.Join(" ", coll11));
var coll12 = from s in colors where s.Substring(s.Length - 1) == "e" select s;
Console.WriteLine(String.Join(" ", coll12) + "\n");

//Getting the list of colors whose name contains character "a" at 3rd place:
var coll13 = from s in colors where s[2] == 'a' select s;
Console.WriteLine(String.Join(" ", coll13));
var coll14 = from s in colors where s.IndexOf("a") == 2 select s;
Console.WriteLine(String.Join(" ", coll14));
var coll15 = from s in colors where s.Substring(2, 1) == "a" select s;
Console.WriteLine(String.Join(" ", coll15) + "\n");

//Getting the list of colors whose name contains character "O or o" in it:
var coll16 = from s in colors where s.Contains('O') || s.Contains('o') select s;
Console.WriteLine(String.Join(" ", coll16));
var coll17 = from s in colors where s.IndexOf('O') >= 0 || s.IndexOf('o') >= 0 select s;
Console.WriteLine(String.Join(" ", coll17));
var coll18 = from s in colors where s.ToUpper().Contains('O') select s;
Console.WriteLine(String.Join(" ", coll18));
var coll19 = from s in colors where s.ToLower().IndexOf('o') >= 0 select s;

```

```

Console.WriteLine(String.Join(" ", coll19) + "\n");

//Getting the list of colors whose name doesn't contains character "O or o" in it:
var coll20 = from s in colors where s.Contains('O') == false && s.Contains('o') == false select s;
Console.WriteLine(String.Join(" ", coll20));
var coll21 = from s in colors where s.IndexOf('O') == -1 && s.IndexOf('o') == -1 select s;
Console.WriteLine(String.Join(" ", coll21));
var coll22 = from s in colors where s.ToUpper().Contains('O') == false select s;
Console.WriteLine(String.Join(" ", coll22));
var coll23 = from s in colors where s.ToLower().IndexOf('o') == -1 select s;
Console.WriteLine(String.Join(" ", coll23) + "\n");
Console.ReadLine();
}
}

```

Note: The type of values being stored in a generic collection can be of user-defined type values also like a class type or structure type that is defined to represent an entity as following:

```
List<Customer> Customers = new List<Customer>();
```

In the above code assume **Customer** is a user-defined class type that represents an entity **Customer**, so we can store objects of **Customer** type under the List where each object can internally represent different attributes of Customer like **Id**, **Name**, **City**, **Balance**, **Status** etc.

To test this above add a class in the project with the name Customer.cs and write the below code in it:

```

public class Customer
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public string? City { get; set; }
    public double Balance { get; set; }
    public bool Status { get; set; }
    public override string ToString() => $"Id: {Id}; Name: {Name}; City: {City}; Balance: {Balance}; Status: {Status}";
}

```

Add another class in the project with the name Class10.cs and write the below code in it:

```

internal class Class10
{
    static void Main()
    {
        //Creating instance of Customer class using Object Initializers.
        Customer c1 = new Customer { Id = 101, Name = "Scott", City = "Delhi", Balance = 15000.00, Status = true };
        Customer c2 = new Customer { Id = 102, Name = "Dave", City = "Mumbai", Balance = 10000.00, Status = true };
        Customer c3 = new Customer { Id = 103, Name = "Sunitha", City = "Chennai", Balance = 15600.00, Status = false };
        Customer c4 = new Customer { Id = 104, Name = "David", City = "Delhi", Balance = 22000.00, Status = true };
        Customer c5 = new Customer { Id = 105, Name = "John", City = "Kolkata", Balance = 34000.00, Status = true };
    }
}

```

```

Customer c6 = new Customer { Id = 106, Name = "Jane", City = "Hyderabad", Balance = 19000.00, Status = true };
Customer c7 = new Customer { Id = 107, Name = "Kavitha", City = "Mumbai", Balance = 16500.00, Status = true };
Customer c8 = new Customer { Id = 108, Name = "Steve", City = "Bengaluru", Balance = 34600.00, Status = false };
Customer c9 = new Customer { Id = 109, Name = "Sophia", City = "Chennai", Balance = 6300.00, Status = true };
Customer c10 = new Customer { Id = 110, Name = "Rehman", City = "Delhi", Balance = 9500.00, Status = true };
Customer c11 = new Customer { Id = 111, Name = "Raj", City = "Hyderabad", Balance = 9800.00, Status = false };
Customer c12 = new Customer { Id = 112, Name = "Rupa", City = "Kolkata", Balance = 13200.00, Status = true };
Customer c13 = new Customer { Id = 113, Name = "Ram", City = "Bengaluru", Balance = 47700.00, Status = true };
Customer c14 = new Customer { Id = 114, Name = "Joe", City = "Hyderabad", Balance = 26900.00, Status = false };
Customer c15 = new Customer { Id = 115, Name = "Peter", City = "Delhi", Balance = 17400.00, Status = true };

```

//Created a List of Customers and added all the Customer instances into the List

```
List<Customer> Customers = new List<Customer>()
```

```
{
```

```
    c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15
};
```

//Implementing LINQ Queries for fetching the data from the List using LINQ to Objects.

//Fetching all rows and columns from the List un-conditionally:

```
//var Coll = from c in Customers select c;
```

//Fetching selected columns and giving alias names to columns:

```
//var Coll = from c in Customers select new { c.Id, c.Name, IsActive = c.Status };
```

//Order By Clause:

```
//var Coll = from c in Customers orderby c.Name select c;
```

```
//var Coll = from c in Customers orderby c.Balance descending select c;
```

//Where Clause:

```
//var Coll = from c in Customers where c.Balance > 25000 select c;
```

```
//var Coll = from c in Customers where c.City == "Hyderabad" select c;
```

```
//var Coll = from c in Customers where c.City == "Bengaluru" && c.Balance > 40000 select c;
```

```
//var Coll = from c in Customers where c.City == "Chennai" || c.Balance > 30000 select c;
```

//Group By Clause:

```
//var Coll = from c in Customers group c by c.City into G select new { City = G.Key, Customers = G.Count() };
```

```
//var Coll = from c in Customers group c by c.City into G select new {
```

```
    City = G.Key, MaxBalance = G.Max(c => c.Balance) };
```

```
//var Coll = from c in Customers group c by c.City into G select new {
```

```
    City = G.Key, MinBalance = G.Min(c => c.Balance) };
```

```
//var Coll = from c in Customers group c by c.City into G select new {
```

```
    City = G.Key, AvgBalance = G.Average(c => c.Balance) };
```

```
//var Coll = from c in Customers group c by c.City into G select new {
```

```
    City = G.Key, TotalBalance = G.Sum(c => c.Balance) };
```

//Having (Where) Clause:

```
//var Coll = from c in Customers group c by c.City into G where G.Count() > 2 select new {
```

```

        City = G.Key, Customers = G.Count() };
//var Coll = from c in Customers group c by c.City into G where G.Max(c => c.Balance) > 25000 select new {
        City = G.Key, MaxBalance = G.Max(c => c.Balance) };
var Coll = from c in Customers group c by c.City into G where G.Min(c => c.Balance) < 10000 select new {
        City = G.Key, MinBalance = G.Min(c => c.Balance) };

foreach (var customer in Coll) {
    Console.WriteLine(customer);
}
Console.ReadLine();
}
}

```

Note: We don't have "having" clause in LINQ, so wherever you need the functionality of "having", use "where" overthere, because both are used for filtering only. LINQ has not given us 2 separate clauses i.e., if we use "where" before "group by" it works like "where" clause whereas if we use "where" after "group by" it works like "having".

Task Parallel Library (TPL)

The Task Parallel Library (TPL) is a set of public types "System.Threading" and "System.Threading.Tasks" namespaces. The purpose of TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications. The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. In addition, the TPL handles the partitioning of the work, the scheduling of Threads on the Thread Pool, cancellation support, state management, and other low-level details. By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish.

Starting with .NET Framework 4, the TPL is the preferred way to write multithreaded and parallel code. However, not all code is suitable for parallelization. For example, if a loop performs only a small amount of work on each iteration, or it doesn't run for many iterations, then the overhead of parallelization can cause the code to run more slowly. Furthermore, parallelization like any multithreaded code adds complexity to your program execution. Although the TPL simplifies multithreaded scenarios, it is recommended that you have a basic understanding of threading concepts, for example, locks, deadlocks, and race conditions, so that you can use the TPL effectively.

To test the examples given below create a new "Console Application" Project naming it as "TPLProjectConsole" and choose the Target Framework as: ".NET 8.0 (Long-term support)", check the Checkbox => "Do not use top-level statements" and click on the "Create" button. First let's write a program without using multi-Threading and to do that write the below code in the default class "Program" which is present under "Program.cs" file by deleting the existing code in the class:

```

internal class Program
{
    static void Print1()
    {
        for (int i = 1; i <= 100; i++)
        {
            Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print1 Method: {i}");
        }
    }
}

```

```

    }
}

static void Print2()
{
    for (int i = 1; i <= 100; i++)
    {
        Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print2 Method: {i}");
    }
}

static void Print3()
{
    for (int i = 1; i <= 100; i++)
    {
        Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print3 Method: {i}");
    }
}

static void Main(string[] args)
{
    Print1(); Print2(); Print3();
}
}

```

Note: in the above code we have defined 3 methods in the class and called them in a **single threaded model** so each method is executed 1 after the other and all the methods are executed by the **Main Thread**, and we can see the **Id** of that **Thread** which will be printed by the statement “**Thread.CurrentThread.ManagedThreadId**”.

Now let's re-write the above program using **multi-Threading**, and to do that add a new class in the project naming it as “**Class1.cs**” and write the below code in the class:

```

internal class Class1
{
    static void Print1()
    {
        for (int i = 1; i <= 100; i++) {
            Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print1 Method: {i}");
        }
    }

    static void Print2()
    {
        for (int i = 1; i <= 100; i++) {
            Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print2 Method: {i}");
        }
    }

    static void Print3()
    {
        for (int i = 1; i <= 100; i++) {
            Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print3 Method: {i}");
        }
    }
}

```

```

    }
}

static void Main()
{
    Thread t1 = new Thread(Print1);
    Thread t2 = new Thread(Print2);
    Thread t3 = new Thread(Print3);
    t1.Start(); t2.Start(); t3.Start();
    t1.Join(); t2.Join(); t3.Join();
    Console.WriteLine($"Main thread with Id: {Thread.CurrentThread.ManagedThreadId} is exiting.");
}
}

```

Note: in the above code we have defined 3 methods and called them by using 3 **separate threads** so each thread will execute 1 method **concurrently** and, in the program, we will be having 4 threads along with the **Main thread** and we can see the Id of those **Threads** which will be printed by the statement **"Thread.CurrentThread.ManagedThreadId"**.

Now let's re-write the above program using **Task Parallelism**, and to do that add a new class in the project naming it as "**Class2.cs**" and write the below code in the class:

```

internal class Class2
{
    static void Print1()
    {
        for (int i = 1; i <= 100; i++) {
            Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print1 Method: {i}");
        }
    }

    static void Print2()
    {
        for (int i = 1; i <= 100; i++) {
            Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print2 Method: {i}");
        }
    }

    static void Print3()
    {
        for (int i = 1; i <= 100; i++) {
            Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; Print3 Method: {i}");
        }
    }

    static void Main()
    {
        Task t1 = new Task(Print1);
        Task t2 = new Task(Print2);
        Task t3 = new Task(Print3);
        t1.Start(); t2.Start(); t3.Start();
        t1.Wait(); t2.Wait(); t3.Wait();
    }
}

```

```

        Console.WriteLine($"Main thread with Id: {Thread.CurrentThread.ManagedThreadId} is exiting.");
    }
}

```

In the above case in place of **Threads** we have used **Tasks** and these **Tasks** will internally use **Threads** to execute the code and the “**Wait**” method we called here is same as the “**Join**” method we use in **Threads**. The process of creating **Tasks**, calling **Start** and **Wait** methods can be simplified and implemented i.e., we can implement the code in Main method of the above program as following also:

```

Task t1 = Task.Factory.StartNew(Print1);
Task t2 = Task.Factory.StartNew(Print2);
Task t3 = Task.Factory.StartNew(Print3);
Task.WaitAll(t1, t2, t3);
Console.WriteLine($"Main thread with Id: {Thread.CurrentThread.ManagedThreadId} is exiting.");

```

In the above code **Factory** is a **static** property of the **Task** class which will refer to **TaskFactory** class and the **StartNew** method of **TaskFactory** class will create a new **Thread**, starts it, and returns the reference of it.

Note: in the above code also, we have defined 3 **methods** and called them by using 3 **separate tasks**, so each **task** will execute 1 **method** concurrently. In this program also we will be having 4 **threads** along with the **Main** thread and we can see the **Id** of those **Threads** in the output.

Calling value returning methods by using Tasks: in the above programs the methods that we called by using **Tasks** are all **non-value returning** as well as they **do not take any parameters** also. Now let's learn how to call **value returning methods** by using **Task** and to do that add a new class in the **project** naming it as “**Class3.cs**” and write the below code in it:

```

internal class Class3
{
    static int GetLength()
    {
        string str = "";
        for (int i = 1; i <= 100000; i++) {
            str += i;
        }
        return str.Length;
    }
    static string ToUpper()
    {
        string str = "Hello World";
        return str.ToUpper();
    }
    static void Main()
    {
        Task<int> t1 = new Task<int>(GetLength);
        Task<string> t2 = new Task<string>(ToUpper);
        t1.Start(); t2.Start();
    }
}

```

OR

```

Task<int> t1 = Task.Factory.StartNew(GetLength);
Task<string> t2 = Task.Factory.StartNew(ToUpper);

int Result1 = t1.Result;
string Result2 = t2.Result;
Console.WriteLine($"Value of Result1 is: {Result1}");
Console.WriteLine($"Value of Result2 is: {Result2}");
}
}

```

Note: in the above program the `GetLength1` and `GetLength2` method of the class are value returning. `GetLength1` method concatenates from 1 to 100000 and then returns the length of that string and `GetLength2` method converts a given string to Upper Case and returns the converted string. So, in this case to capture the values we need to use the `Task` class which takes the generic parameter `<T>` and in this case `<T>` is of type `integer` for `GetLength1` and of type `string` for `GetLength2` and after execution of the method we can capture the result by calling "Result" property of `Task` class which returns the result as `integer` for `GetLength1` and `string` for `GetLength2`.

Calling value returning method with parameters by using Tasks: in the above program the methods that we called by using `Task` are value returning methods and now let's learn how to call value returning methods which takes parameters also by using `Task` and to do that add a new class in the project naming it as "`Class4.cs`" and write the below code in it:

```

internal class Class4
{
    static int GetLength(int ub)
    {
        string str = "";
        for (int i = 1; i <= ub; i++)
            str += i;
        return str.Length;
    }
    static string ToUpper(string str)
    {
        return str.ToUpper();
    }
    static void Main()
    {
        Task<int> t1 = new Task<int>(() => GetLength(50000));
        Task<string> t2 = new Task<string>(() => ToUpper("Hello India"));
        t1.Start(); t2.Start();
        OR
        Task<int> t1 = Task.Factory.StartNew(() => GetLength(50000));
        Task<string> t2 = Task.Factory.StartNew(() => ToUpper("Hello India"));

        int Result1 = t1.Result;
        string Result2 = t2.Result;
    }
}

```

```

        Console.WriteLine($"Value of Result1 is: {Result1}");
        Console.WriteLine($"Value of Result2 is: {Result2}");
    }
}

```

Note: in the above program GetLength method of class concatenates 1 to a number that is passed to the method as parameter value and then returns the length of that string, so in this case to pass values to the method we need to take the help of a delegate.

Thread Synchronization: synchronization is a technique that allows only one thread to access the resource for the time. No other thread can interrupt until the assigned thread finishes its task. In multithreading program, threads are allowed to access any resource for the required execution time. Threads share resources and executes asynchronously. Accessing shared resources (data) is critical task that sometimes may halt the system. We deal with it by making threads synchronized. It is mainly used in case of transactions like deposit, withdraw etc.

We can use C# **lock** keyword to execute program synchronously. It is used to get lock for the current thread, execute the task and then release the lock. It ensures that other thread does not interrupt the execution until the execution finish. To test this, add a new class in the project naming it as “**Class5.cs**” and write the below code in it:

```

class Class5
{
    public static void Print()
    {
        lock (typeof(Class5))
        {
            Console.Write("[CSharp Is ");
            Thread.Sleep(5000);
            Console.WriteLine("Object Oriented]"));
        }
    }

    static void Main()
    {
        Thread t1 = new Thread(Print);
        Thread t2 = new Thread(Print);
        Thread t3 = new Thread(Print);
        t1.Start(); t2.Start(); t3.Start();
        t1.Join(); t2.Join(); t3.Join();
    }
}

```

If we want to perform synchronization with Tasks then here also the process is same and to test this, add a new class in the project naming it as “**Class6.cs**” and write the below code in it:

```

class Class6
{
    public static void Print()
    {

```

```

lock (typeof(Class6))
{
    Console.Write("[CSharp Is ");
    Task.Delay(5000).Wait();
    Console.WriteLine("Object Oriented]");
}
}

static void Main()
{
    Task t1 = new Task(Print);
    Task t2 = new Task(Print);
    Task t3 = new Task(Print);
    t1.Start(); t2.Start(); t3.Start();

    Or

    Task t1 = Task.Factory.StartNew(Print);
    Task t2 = Task.Factory.StartNew(Print);
    Task t3 = Task.Factory.StartNew(Print);
    Task.WaitAll(t1, t2, t3);
}
}

```

Data Parallelism: this refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source like an array or collection. In data parallel operations, the source is partitioned so that multiple threads can operate on different segments concurrently. The **Task Parallel Library (TPL)** supports data parallelism through “Parallel” class which is present under **System.Threading.Tasks** namespace. This class provides method-based parallel implementations of for and foreach loops. You write the loop logic for a “Parallel.For” or “Parallel.ForEach” loops much as you would write a sequential loop. You do not have to create threads or queue the work items i.e., **TPL** handles all the low-level work for you.

Sequential Version:

```

foreach (var item in Source_Collection)
{
    Process(item);
}

```

Parallel Equivalent:

```
Parallel.ForEach(Source_Collection, item => Process(item));
```

Let's now write a program to understand the difference between sequential for loop and parallel for loop and to do that add a new class in the project naming it as “**Class7.cs**” and write the below code in it:

```

using System.Diagnostics;
class Class7
{
    static void Main()
    {

```

```

Stopwatch sw1 = new Stopwatch();
sw1.Start();
string str1 = "";
for (int i = 1; i < 200000; i++)
{
    str1 = str1 + i;
}
sw1.Stop();
Console.WriteLine("Time taken to execute the code by using sequential for loop: " + sw1.ElapsedMilliseconds);

Stopwatch sw2 = new Stopwatch();
sw2.Start();
string str2 = "";
Parallel.For(1, 200000, i =>
{
    str2 = str2 + i;
});
sw2.Stop();
Console.WriteLine("Time taken to execute the code by using parallel for loop: " + sw2.ElapsedMilliseconds);
}
}

```

Let's now write another program to understand the difference between **sequential foreach loop** and **parallel foreach loop** and to do that add a new class in the project naming it as "**Class8.cs**" and write the below code in it:

```

using System.Diagnostics;
class Class8
{
    static void Main()
    {
        int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
                     31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50 };

        Stopwatch sw1 = new Stopwatch();
        sw1.Start();
        foreach(int i in arr) {
            Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; i value: {i}");
        }
        sw1.Stop();

        Console.WriteLine();

        Stopwatch sw2 = new Stopwatch();
        sw2.Start();
        Parallel.ForEach(arr, i => {

```

```

        Console.WriteLine($"Thread Id: {Thread.CurrentThread.ManagedThreadId}; i value: {i}");
    });
    sw2.Stop();
}

Console.WriteLine("Time taken to execute code by using sequential foreach loop: " + sw1.ElapsedMilliseconds);
Console.WriteLine("Time taken to execute code by using parallel foreach loop: " + sw2.ElapsedMilliseconds);
}
}

```

Note: If you observe the above 2 programs in the first code parallel for loop executed much faster than a sequential for loop whereas in the second case sequential foreach loop executed faster than parallel foreach loop because when we are doing any bulk task inside the loop then parallel loops are faster whereas if you are just iterating and doing a small task inside a loop then sequential loops are faster.

Chaining Tasks using Continuation Tasks: in **asynchronous programming**, it's common for one asynchronous operation, on completion, to invoke a second operation. Continuations allow descendant operations to consume the results of the first operation. A continuation task (also known just as a continuation) is an asynchronous task that's invoked by another task, known as the antecedent, when the antecedent finishes. To test this, add a new class in the project naming it as "**Class9.cs**" and write the below code in it:

```

class Class9
{
    static void Method1(int x, int ub) {
        for (int i = 1; i <= ub; i++)
            Console.WriteLine($"{x} * {i} = {x * i}");
    }
    static void Method2(int x, int ub) {
        for (int i = ub; i > 0; i--)
            Console.WriteLine($"{x} * {i} = {x * i}");
    }
    static void Main() {
        Task t = Task.Factory.StartNew(() => Method1(5, 12)).ContinueWith((antecedent) =>
            Console.WriteLine()).ContinueWith((antecedent) => Method2(5, 12));
        t.Wait();
        Console.ReadLine();
    }
}

```

Asynchronous programming with async and await: **async** and **await** in C# are the code markers, which marks code positions from where the control should resume after a task completes. When we are dealing with UI, and on a button click we called a long-running method like reading a large file or something else which will take a long time and, in that case, the entire application must wait to complete the task. In other words, if a process is blocked in a synchronous application, the whole application gets blocked and stops responding until the whole task completes.

```

class Class10
{

```

```

static async void Test1() {
    Console.WriteLine("Started reading values from DB.....");
    await Task.Delay(10000);
    Console.WriteLine("Completed reading values from DB.....");
}

static void Test2() {
    Console.Write("Please enter your name: ");
    string Name = Console.ReadLine();
    Console.WriteLine($"Name you entered is: {Name}");
}

static void Main() {
    Test1();
    Test2();
    Console.ReadLine();
}
}

```

Logical Programs

Write a program to print the given no is a prime number or not?

```

class PrimeNumberTest
{
    static void Main()
    {
        Console.Write("Enter a number to check it's a prime: ");
        uint Number = uint.Parse(Console.ReadLine());
        if(Number == 0 || Number == 1)
        {
            Console.WriteLine("Please enter a number other than 0 & 1");
            return;
        }
        bool IsPrime = true;
        uint HalfNumber = Number / 2;
        for(uint i = 2;i<=HalfNumber;i++)
        {
            if(Number % i == 0)
            {
                IsPrime = false;
                break;
            }
        }
        if(IsPrime == true)
            Console.WriteLine("Given number is a prime.");
        else
            Console.WriteLine("Given number is not a prime.");
        Console.ReadLine();
    }
}

```

```
}
```

Write a program to swap 2 numbers without using 3rd variable?

```
class SwapNumbers1          //Solution 1
{
    static void Main()
    {
        int a = 342, b = 784;
        Console.WriteLine($"Numbers Before Swap: a => {a}; b => {b}");
        a = a * b;    b = a / b;    a = a / b;
        Console.WriteLine($"Numbers After Swap: a => {a}; b => {b}");
        Console.ReadLine();
    }
}

class SwapNumbers2          //Solution 2
{
    static void Main()
    {
        Console.Write("Enter 1st number: ");
        int a = int.Parse(Console.ReadLine());
        Console.Write("Enter 2nd number: ");
        int b = int.Parse(Console.ReadLine());
        Console.WriteLine($"Numbers Before Swap: a => {a}; b => {b}");
        a = a + b;    b = a - b;    a = a - b;
        Console.WriteLine($"Numbers After Swap: a => {a}; b => {b}");
        Console.ReadLine();
    }
}
```

Write a program to print the reverse of a given number?

```
class ReverseNumber
{
    static void Main()
    {
        Console.Write("Enter a number: ");
        int Number = int.Parse(Console.ReadLine());
        int Reminder, Reverse = 0;
        while(Number != 0)
        {
            Reminder = Number % 10;
            Reverse = Reverse * 10 + Reminder;
            Number = Number / 10;
        }
        Console.WriteLine("Reversed Number is: " + Reverse);
        Console.ReadLine();
    }
}
```

Write the program to print the binary value of a given number?

```
class NumberToBinary
{
    static void Main()
    {
        Console.Write("Enter a number to convert into binary: ");
        int Number = int.Parse(Console.ReadLine());
        int[] arr = new int[16];
        int i;
        for(i = 0;Number > 0;i++)
        {
            arr[i] = Number % 2;
            Number = Number / 2;
        }
        Console.Write("Binary value of the given number is: ");
        for(i = i - 1;i >= 0;i--)
        {
            Console.Write(arr[i]);
        }
        Console.ReadLine();
    }
}
```

Write a program to check whether a given number is a palindrome?

```
class PalindromeNumber
{
    static void Main()
    {
        Console.Write("Enter a Number: ");
        int Number = int.Parse(Console.ReadLine());
        int OldNumber = Number;
        int Reminder, Reverse = 0;
        while(Number != 0)
        {
            Reminder = Number % 10;
            Reverse = (Reverse * 10) + Reminder;
            Number = Number / 10;
        }
        if (OldNumber == Reverse)
            Console.WriteLine("Given number is a palindrome");
        else
            Console.WriteLine("Given number is not a palindrome");
        Console.ReadLine();
    }
}
```

Write a program to print the Fibonacci series up to a given upper bound?

```
class FibonacciSeries
```

```

{
    static void Main()
    {
        Console.Write("Enter the number of elements for Fibanocci Series: ");
        int Number = int.Parse(Console.ReadLine());
        int Num1 = 0, Num2 = 1, Num3;
        Console.Write(Num1 + " " + Num2 + " ");
        for(int i = 2;i < Number;i++)
        {
            Num3 = Num1 + Num2;
            Console.Write(Num3 + " ");
            Num1 = Num2;
            Num2 = Num3;
        }
        Console.ReadLine();
    }
}

```

Write a program to print the factorial of a given number?

```

class Factorial
{
    static void Main()
    {
        Console.Write("Enter a number to find it's factorial: ");
        uint Number = uint.Parse(Console.ReadLine());
        uint Result = 1;
        for(uint i=1;i<=Number;i++)
        {
            Result = Result * i;
        }
        Console.WriteLine("Factorial of given number is: " + Result);
        Console.ReadLine();
    }
}

```

Write a program to find whether the give number is an Armstrong number or not?

```

class ArmstrongNumber
{
    static void Main()
    {
        Console.Write("Enter a number to find it is Armstrong: ");
        int Number = int.Parse(Console.ReadLine());
        int Original = Number;
        int Reminder, Sum = 0;
        while(Number > 0)
        {
            Reminder = Number % 10;
            Sum = Sum + (Reminder * Reminder * Reminder);
        }
    }
}

```

```
Number = Number / 10;
}
if (Original == Sum)
    Console.WriteLine($"{Original} is an armstrong number");
else
    Console.WriteLine($"{Original} is not an armstrong number");
Console.ReadLine();
}
}
```

Write a program to find the sum of digits of a given number?

```
class SumOfDigits1
{
    static void Main()
    {
        Console.Write("Enter a number to find sum of its digits: ");
        int Number = int.Parse(Console.ReadLine());
        int Reminder, Sum = 0;
        while(Number > 0)
        {
            Reminder = Number % 10;
            Sum = Sum + Reminder;
            Number = Number / 10;
        }
        Console.WriteLine("Sum of the digits of given no is: " + Sum);
        Console.ReadLine();
    }
}
```

Write a program to find the sum of digits of a given number until single digit?

```
class SumOfDigits2
{
    static void Main()
    {
        Console.Write("Enter a number to find sum of it's digits: ");
        int Number = int.Parse(Console.ReadLine());
        int Reminder, Sum = 0;
        do
        {
            if(Sum != 0)
            {
                Number = Sum;
                Sum = 0;
            }
            while (Number > 0)
            {
                Reminder = Number % 10;
                Sum = Sum + Reminder;
            }
        }
```

```

        Number = Number / 10;
    }
}
while (Sum > 9);
Console.WriteLine("Sum of the digits of given no is: " + Sum);
Console.ReadLine();
}
}

```

Write a program to print the given number in words?

```

class NumberToString
{
    static void Main()
    {
        Console.Write("Enter a number: ");
        int Number = int.Parse(Console.ReadLine());
        int Reminder, Reverse = 0;
        while(Number > 0)
        {
            Reminder = Number % 10;
            Reverse = Reverse * 10 + Reminder;
            Number = Number / 10;
        }
        while(Reverse > 0)
        {
            Reminder = Reverse % 10;
            switch (Reminder)
            {
                case 1:
                    Console.Write("one ");
                    break;
                case 2:
                    Console.Write("two ");
                    break;
                case 3:
                    Console.Write("three ");
                    break;
                case 4:
                    Console.Write("four ");
                    break;
                case 5:
                    Console.Write("five ");
                    break;
                case 6:
                    Console.Write("six ");
                    break;
                case 7:

```

```

        Console.Write("seven ");
        break;
    case 8:
        Console.Write("eight ");
        break;
    case 9:
        Console.Write("nine ");
        break;
    case 0:
        Console.Write("zero ");
        break;
    }
    Reverse = Reverse / 10;
}
Console.ReadLine();
}
}

```

Write a program to find the given year is a leap year or not?

```

class LeapYear
{
    static void Main()
    {
        Console.Write("Enter the year in 4 digits: ");
        int Year = int.Parse(Console.ReadLine());
        if ((Year % 4 == 0 && Year % 100 != 0) || (Year % 400 == 0))
            Console.WriteLine($"{Year} is a leap year.");
        else
            Console.WriteLine($"{Year} is not a leap year.");
        Console.ReadLine();
    }
}

```

Write a program to print the larger number in an array?

```

class LargerNumberInArray
{
    static void Main()
    {
        Console.Write("Specify the no of items to compare: ");
        int UB = int.Parse(Console.ReadLine());
        Console.Clear();
        int[] arr = new int[UB];
        for(int i=0;i<UB;i++)
        {
            Console.Write($"Enter Item{i + 1}: ");
            arr[i] = int.Parse(Console.ReadLine());
        }
        int LargeNumber = arr[0];

```

```
for(int i=1;i<UB;i++)
{
    if(arr[i] > LargeNumber)
    {
        LargeNumber = arr[i];
    }
}
Console.WriteLine("Larger number in the array is: " + LargeNumber);
Console.ReadLine();
}
```

Write a program to print the given string in reverse?

```
class StringReverse
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        string reverse = "";
        foreach(char ch in input)
            reverse = ch + reverse;
        Console.WriteLine($"Reverse of given string '{input}' is: '{reverse}'");
        Console.ReadLine();
    }
}
```

Write a program to print the no. of words in each string?

```
class WordCount
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        int Count = 0, CharCount = 0;
        bool Flag = true, EndSpace = false;
        bool StartSpace = false;
        foreach (char ch in input)
        {
            CharCount += 1;
            if (CharCount == 1 && ch == 32)
                StartSpace = true;
            if (ch == 32 && Flag == false)
                continue;
            else {
                Flag = true;
                EndSpace = false;
            }
        }
    }
}
```

```

if (Count == 0)
    Count = 1;
if (ch == 32)
{
    Count += 1;
    Flag = false;
    EndSpace = true;
}
}

if (StartSpace == true)
    Count -= 1;
if (EndSpace == true)
    Count -= 1;
Console.WriteLine("No of words in the given string are: " + Count);
Console.ReadLine();
}
}

```

Write a program to print the length of a given string?

```

class StringLength
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        int Length = 0;
        foreach (char ch in input)
            Length += 1;
        Console.WriteLine("Length of given string is: " + Length);
        Console.ReadLine();
    }
}

```

Write a program to print the no. of characters in each string?

```

class CharCount
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        int Length = 0;
        foreach (char ch in input)
        {
            if(ch != 32)
                Length += 1;
        }
        Console.WriteLine("No. of char's in given string are: " + Length);
        Console.ReadLine();
    }
}

```

```
}
```

Write a program to print the words in reverse order of a given string?

```
class ReverseWords
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        string word = "", reverseWords = "";
        foreach(char ch in input)
        {
            if (ch != 32)
                word = word + ch;
            else
            {
                reverseWords = " " + word + reverseWords;
                word = "";
            }
        }
        if (word != "")
            reverseWords = word + reverseWords;
        Console.WriteLine(reverseWords);
        Console.ReadLine();
    }
}
```

Write a program to convert the given string into lower case?

```
class StringToLower
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        string output = "";
        foreach(char ch in input)
        {
            if (ch >= 65 && ch <= 90)
                output += (char)(ch + 32);
            else
                output += ch;
        }
        Console.WriteLine(output);
        Console.ReadLine();
    }
}
```

Write a program to convert the given string into upper case?

```
class StringToUpper
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        string output = "";
        foreach (char ch in input)
        {
            if (ch >= 97 && ch <= 122)
                output += (char)(ch - 32);
            else
                output += ch;
        }
        Console.WriteLine(output);
        Console.ReadLine();
    }
}
```

Write a program to convert the given string into pascal case?

```
class StringToPascal
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        string lower = "";
        foreach (char ch in input)
        {
            if (ch >= 65 && ch <= 90)
                lower += (char)(ch + 32);
            else
                lower += ch;
        }
        string pascal = "";
        bool firstChar = true, flag = false;
        foreach(char ch in lower)
        {
            if (firstChar == true)
            {
                if (ch >= 97 && ch <= 122)
                    pascal += (char)(ch - 32);
                firstChar = false;
                continue;
            }
            if (flag == true)
            {
```

```

        if (ch >= 97 && ch <= 122)
            pascal += (char)(ch - 32);
        flag = false;
    }
    else
        pascal += ch;
    if (ch == 32)
        flag = true;
}
Console.WriteLine(pascal);
Console.ReadLine();
}
}

```

Write a program to find out the unique characters in each string?

```

class UniqueChars
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        bool Exists = false;
        int Count1 = 0, Count2 = 0;
        foreach(char ch1 in input)
        {
            Count1 += 1;
            foreach(char ch2 in input)
            {
                Count2 += 1;
                if(Count1 != Count2)
                {
                    if (ch1 != ch2 && ch1 != 32)
                        Exists = false;
                    else
                    {
                        Exists = true;
                        break;
                    }
                }
            }
            if (Exists == false)
                Console.Write(ch1);
            Count2 = 0; Exists = false;
        }
        Console.ReadLine();
    }
}

```

Write a program to find out the duplicate characters in each string?

```
class DuplicateChars
{
    static void Main()
    {
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();
        int Length = 0;
        foreach (char ch in input)
            Length += 1;
        char[] arr = new char[Length];
        int Index = 0;
        foreach(char ch in input)
        {
            arr[Index] = ch;
            Index += 1;
        }
        int Count1 = 0, Count2 = 0;
        foreach(char ch1 in arr)
        {
            Count1 += 1;
            foreach(char ch2 in arr)
            {
                Count2 += 1;
                if(Count1 != Count2)
                {
                    if(ch1 == ch2 && ch1 != 32)
                    {
                        Console.WriteLine(ch1);
                        arr[Count1 - 1] = ' ';
                        arr[Count2 - 1] = ' ';
                        break;
                    }
                }
            }
            Count2 = 0;
        }
        Console.ReadLine();
    }
}
```

Write a program to print the roman number of a given number?

```
class NumberToRoman
{
    static void Main()
    {
        Console.Write("Enter a string: ");
```

```

int num = int.Parse(Console.ReadLine());
string roman = ToRoman(num);
Console.WriteLine(roman);
Console.ReadLine();
}
public static string ToRoman(int num)
{
    if (num < 0 || num > 3999)
        return "Enter a number between 1 and 3999";
    else if (num >= 1000)
        return "M" + ToRoman(num - 1000);
    else if (num >= 900)
        return "CM" + ToRoman(num - 900);
    else if (num >= 500)
        return "D" + ToRoman(num - 500);
    else if (num >= 400)
        return "CD" + ToRoman(num - 400);
    else if (num >= 100)
        return "C" + ToRoman(num - 100);
    else if (num >= 90)
        return "XC" + ToRoman(num - 90);
    else if (num >= 50)
        return "L" + ToRoman(num - 50);
    else if (num >= 40)
        return "XL" + ToRoman(num - 40);
    else if (num >= 10)
        return "X" + ToRoman(num - 10);
    else if (num >= 9)
        return "IX" + ToRoman(num - 9);
    else if (num >= 5)
        return "V" + ToRoman(num - 5);
    else if (num >= 4)
        return "IV" + ToRoman(num - 4);
    else if (num >= 1)
        return "I" + ToRoman(num - 1);
    else
        return "";
}
}

```

Write a program to print the below output:

```
1
12
123
1234
12345
```

```
class Pattern1
{
    static void Main()
    {
        Console.Write("Enter a number: ");
        int num = int.Parse(Console.ReadLine());
        Console.Clear();
        for(int i=1;i<=num;i++)
        {
            for(int j=1;j<=i;j++)
                Console.Write(j);
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}
```

Write a program to print the below output:

```
5
54
543
5432
54321
```

```
class Pattern2
{
    static void Main()
    {
        Console.Write("Enter a number: ");
        int Number = int.Parse(Console.ReadLine());
        Console.Clear();
        for (int i = Number; i >= 1; i--)
        {
            for (int j = Number; j >= i; j--)
                Console.Write(j);
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}
```

}

Write a program to print the below output:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
```

```
class Pattern3
{
    static void Main()
    {
        Console.Write("Enter number of rows: ");
        int Rows = int.Parse(Console.ReadLine());
        Console.Clear();
        int x = 1;
        for(int i=1;i<=Rows;i++)
        {
            for (int j = 1; j <= i; j++)
                Console.Write($"{x++} ");
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}
```

Write a program to print the below output:

```
1
01
101
0101
10101
010101
```

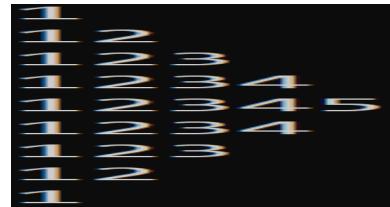
```
class Pattern4
{
    static void Main()
    {
        Console.Write("Enter number of rows: ");
        int Rows = int.Parse(Console.ReadLine());
        Console.Clear();
        int x = 0, y = 0;
        for (int i = 1; i <= Rows; i++)
        {
            if(i % 2 == 0)
            {
                x = 1;
```

```

        y = 0;
    }
    else
    {
        x = 0;
        y = 1;
    }
    for (int j = 1; j <= i; j++)
    {
        if (j % 2 == 0)
            Console.WriteLine(x);
        else
            Console.WriteLine(y);
    }
    Console.ReadLine();
}
Console.ReadLine();
}

```

Write a program to print the below output:



```

class Pattern5
{
    static void Main()
    {
        Console.Write("Enter number of rows: ");
        int num = int.Parse(Console.ReadLine());
        Console.Clear();
        for (int i = 1; i < num; i++)
        {
            for (int j = 1; j <= i; j++)
                Console.Write(j);
            Console.WriteLine();
        }
        for (int i = num; i >= 0; i--)
        {
            for (int j = 1; j <= i; j++)
                Console.Write(j);
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}

```

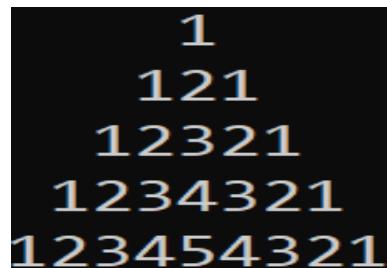
```
}
```

Write a program to print the below output:



```
class Pattern6
{
    static void Main()
    {
        Console.Write("Enter number of rows: ");
        int num = int.Parse(Console.ReadLine());
        Console.Clear();
        for (int i = num; i >= 0; i--)
        {
            for (int j = 1; j <= i; j++)
                Console.Write(j);
            if(i > 0)
                Console.WriteLine();
        }
        for (int i = 1; i <= num; i++)
        {
            for (int j = 1; j <= i; j++)
                Console.Write(j);
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}
```

Write a program to print the below output:



```
class Pattern7
{
    static void Main()
    {
```

```

Console.WriteLine("Enter number of rows: ");
int num = int.Parse(Console.ReadLine());
Console.Clear();
for(int i=1;i<= num;i++)
{
    for (int space = 1; space <= (num - i); space++)
        Console.Write(" ");
    for (int j = 1; j <= i; j++)
        Console.Write(j);
    for (int k = (i - 1); k >= 1; k--)
        Console.Write(k);
    Console.WriteLine();
}
Console.ReadLine();
}
}

```

Write a program to print the below output:

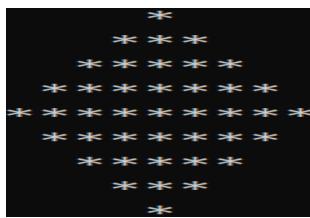
			1			
		1	1			
	1	2	1			
	1	3	3	1		
	1	4	6	4	1	
1	5	10	10	5	1	

```

class Pattern8
{
    static void Main()
    {
        Console.WriteLine("Enter number of rows: ");
        int num = int.Parse(Console.ReadLine());
        Console.Clear();
        int result;
        for(int i=0;i<=num;i++)
        {
            result = 1;
            for(int j=i;j <= num - 1;j++)
                Console.Write(" ");
            for(int k=0;k<=i;k++)
            {
                Console.Write(result + " ");
                result = (result * (i - k) / (k + 1));
            }
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}

```

Write a program to print the below output:



```
class Pattern9
{
    static void Main()
    {
        Console.Write("Enter number of rows: ");
        int num = int.Parse(Console.ReadLine());
        Console.Clear();
        int count = num - 1;
        for(int i=1;i<num+1;i++)
        {
            for (int j = 1; j <= count; j++)
                Console.Write(" ");
            count--;
            for (int k = 1; k <= 2 * i - 1; k++)
                Console.Write("*");
            Console.WriteLine();
        }
        count = 1;
        for(int i=1;i<=num-1;i++)
        {
            for (int j = 1; j <= count; j++)
                Console.Write(" ");
            count++;
            for (int k = 1; k <= 2 * (num - i) - 1; k++)
                Console.Write("*");
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}
```

Write a program to print the below output:



```
class Pattern10
```

```

{
static void Main()
{
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    for(int i=0;i<num;i++)
    {
        for (int j = 0; j <= i; j++)
            Console.Write("*");
        Console.WriteLine();
    }
    Console.ReadLine();
}
}

```

Write a program to print the below output:



```

class Pattern11
{
static void Main()
{
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    for(int i=0;i < num;i++)
    {
        if(i == 0 || i == num - 1)
        {
            for (int j = 0; j < num; j++)
                Console.Write('*');
            Console.WriteLine();
        }
        else
        {
            for(int j=0;j<num;j++)
            {

```

```

        if (j == 0 || j == num - 1)
            Console.Write("*");
        else
            Console.Write(" ");
    }
    Console.WriteLine();
}
}
Console.ReadLine();
}
}

```

Bubble Sort: how does Bubble Sort work is starting at index zero, we take an item and the item next in the array and compare them. If they are in the right order, then we do nothing, if they are in the wrong order (e.g. the item lower in the array is actually a higher value than the next element), then we swap these items. Then we continue through each item in the array doing the same thing (Swapping with the next element if it's higher).

```

class BubbleSort
{
    static void Main(string[] args)
    {
        int[] arr = { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14, 28 };
        bool itemMoved = false;
        do
        {
            itemMoved = false;
            for (int i = 0; i < arr.Length - 1; i++)
            {
                if (arr[i] > arr[i + 1])
                {
                    int lowerValue = arr[i + 1];
                    arr[i + 1] = arr[i];
                    arr[i] = lowerValue;
                    itemMoved = true;
                }
            }
        } while (itemMoved);

        foreach (int i in arr)
            Console.Write(i + " ");
        Console.ReadLine();
    }
}

```

Now since we are only comparing each item with its neighbor, each item may only move a single place when it needs to move several places. So how does Bubble Sort solve this? Well, it just runs the entire process all over again. Notice how we have the variable called “itemMoved”. We simply set this to true if we did swap an item

and start the scan all over again. Because we are moving things one at a time, not directly to the right position, and having to multiple passes to get things right, Bubble Sort is seen as extremely inefficient.

Selection Sort: It's remarkably a simple algorithm to explain and the way Selection Sort works is an outer loop visits each item in the array to find out whether it is the minimum of all the elements after it. If it is not the minimum, it is going to be swapped with whatever item in the rest of the array is the minimum. For example, if you have an array of 10 elements, this means that "i" goes from 0 to 9. When we are looking at position 0, we check to find the position of the minimum element in positions 1 ... 9. If the minimum is not already at position "i", we swap the minimum into place. Then we consider "i = 1" and look at positions 2 .. 9. And so on.

```
class SelectionSort
{
    static void Main()
    {
        int[] arr = { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14, 28 };
        for(int i=0;i<arr.Length;i++)
        {
            int min = i;
            for(int j=i+1;j<arr.Length;j++)
            {
                if(arr[min] > arr[j])
                {
                    min = j;
                }
            }
            if(min != i)
            {
                int lowerValue = arr[min];
                arr[min] = arr[i]; arr[i] = lowerValue;
            }
        }
        foreach (int i in arr)
        {
            Console.Write(i + " ");
        }
        Console.ReadLine();
    }
}
```

Insertion Sort: In the Insertion Sort algorithm, we build a sorted list from the bottom of the array. We repeatedly insert the next element into the sorted part of the array by sliding it down to its proper position. This will require as many exchanges as Bubble Sort, since only one inversion is removed per exchange.

```
class InsertionSort
{
    static void Main()
    {
        int[] arr = { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14, 28 };
        for(int i=0;i<arr.Length;i++)
        {
```

```

int item = arr[i];
int currentIndex = i;
while(currentIndex > 0 && arr[currentIndex - 1] > item)
{
    arr[currentIndex] = arr[currentIndex - 1];
    currentIndex--;
}
arr[currentIndex] = item;
}

foreach (int i in arr)
    Console.Write(i + " ");
Console.ReadLine();
}
}

```

Shell Sort: Donald Shell published the first version of this sort; hence this is known as Shell sort. This sorting is a generalization of insertion sort that allows the exchange of items that are far apart. It starts by comparing elements that are far apart and gradually reduces the gap between elements being compared. The running time of Shell sort varies depending on the gap sequence it uses to sort the elements.

```

class ShellSort
{
    static void Main()
    {
        int[] arr = { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14, 28 };
        int n = arr.Length;
        int gap = n / 2;
        int temp;
        while (gap > 0)
        {
            for(int i=0;i + gap < n;i++)
            {
                int j = i + gap;
                temp = arr[j];
                while(j - gap >= 0 && temp < arr[j - gap])
                {
                    arr[j] = arr[j - gap];
                    j = j - gap;
                }
                arr[j] = temp;
            }
            gap = gap / 2;
        }

        foreach (int i in arr)
        {
            Console.Write(i + " ");
        }
    }
}

```

```
        Console.ReadLine();
    }
}
```

Quick Sort: Like Merge Sort, Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of Quick Sort that pick pivot in different ways.

1. Always pick first element as pivot (implemented below).
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

The main idea for finding pivot is - the pivot or pivot element is the element of an array, which is selected first to do certain calculations. The key process in Quick Sort is partition. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

```
class QuickSort
{
    static int[] arr;
    public static void Sort(int left, int right)
    {
        int pivot, leftEnd, rightEnd;
        leftEnd = left;
        rightEnd = right;
        pivot = arr[left];
        while (left < right)
        {
            while ((arr[right] >= pivot) && (left < right))
            {
                right--;
            }
            if (left != right)
            {
                arr[left] = arr[right]; left++;
            }
            while ((arr[left] <= pivot) && (left < right))
            {
                left++;
            }
            if (left != right)
            {
                arr[right] = arr[left];
                right--;
            }
        }
        arr[left] = pivot;
```

```

pivot = left;
left = leftEnd;
right = rightEnd;
if(left < pivot)
{
    Sort(left, pivot - 1);
}
if(right > pivot)
{
    Sort(pivot + 1, right);
}
static void Main()
{
    arr = new int[] { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14 };
    Sort(0, arr.Length - 1);
    foreach (int i in arr)
    {
        Console.Write(i + " ");
    }
    Console.ReadLine();
}
}

```

ASP.NET Web Forms

ASP.NET is an Open-Source Web Technology from Microsoft for building **Web Applications** using .NET Languages.

Applications are divided into different categories like:

1. Desktop Applications
2. Web Applications
3. Mobile Applications

Desktop Applications means, these applications must be installed on our computer first to use them. Example: MS Office, Skype Messenger, Zoom, Browsers, Visual Studio, Management Studio etc.

Web Applications means, we install these applications first on a centralized machine known as Web Server and then provide access to clients, so that clients can connect to the Web Server using a browser through internet and then access the application. Example: Facebook, Gmail, Amazon, Flipkart, etc.

Mobile Applications are also like Desktop Applications only i.e., we need to install them on our Mobile to consume but work with the help of Internet like a Web Application. Example: WhatsApp, Swiggy, Zomato, Uber, Ola, etc.

.NET is Platform Independent i.e., applications that are developed by using .NET can run on multiple Platforms and to run .NET Applications on a machine that machine should be 1st installed with a software i.e., .NET Runtime. Microsoft provided us 3 different .NET Runtimes which are evolved over a period, like:

- ❖ .NET Framework Runtime
- ❖ .NET Core Runtime
- ❖ .NET Runtime

.NET Framework Runtime was launched into the market by Microsoft in the year 2002 and this is available only for Windows Platforms. The first version of this Runtime is 1.0 and the last version is 4.8.

.NET Core Runtime was launched into the market by Microsoft in the year 2016 and this is available for multiple platforms like Windows, Linux, and Mac. The first version of this Runtime is 1.0 and the last version is 3.1.

.NET Runtime is also same as .NET Core Runtime which was launched into the market by Microsoft in the year 2020 and this is also available for multiple platforms like Windows, Linux, and Mac. This is a combination of .NET Framework and .NET Core and evolved as “**One .NET**”. The first version of .NET Runtime started with 5.0 and we call this as .NET 5 and the latest is .NET 8.

In .NET Framework we have been provided with ASP.NET Technology for building Web Applications and under this we have different options again, those are:

- ❖ ASP.NET Web Forms
- ❖ ASP.NET MVC
- ❖ ASP.NET Web API

From .NET CORE we have been provided with ASP.NET Core Technology for building Web Applications and under this also we have different options again, those are:

- ❖ ASP.NET Core Web App (Razor Pages)
- ❖ ASP.NET Core Web App (Model-View-Controller)
- ❖ ASP.NET Core Web API

Till now you have created some Web Pages by using HTML, Java Script, and CSS, and are able to access those pages from your local machines by using their physical path or address in the browser, but how can we access those pages from remote machines within a network?

Ans: If we want to provide access to Web Pages, we have developed to remote machines we need to take the help of a **Server** known as “**Web Server**”.

What is a Server?

Ans: It is software which works on 2 principles like request and response. There are so many **Servers** software's in the industry, like **DB Servers** (**Oracle**, **SQL Server**, and **My SQL** etc.), **Application Servers**, **Web Servers**, etc.

What is the need for a Web Server?

Ans: this software is used for taking request (**HTTP Request**) from clients in the form of a “**URL (Uniform Resource Locator)**” and then sends **Web Pages** as response (**HTTP Response**).

What Web Server software is available for us to consume?

Ans: There are many **webserver** software's that are available in the market like **Apache Web Server** from Apache, **Nginx Web Server** from NGINX, **IIS Web Server** from Microsoft, **GWS Web Server** from Google, **LiteSpeed Web Server** from LiteSpeed Technologies.

Working with IIS Web Server:

- IIS stands for Internet Information Services which is formerly known as Internet Information Server.
- IIS Web Server is a product of Microsoft and more compatible for our ASP.NET Applications.
- IIS is a part of Windows OS which is generally installed on our machines along with OS, and to verify whether it is installed on your machine or not, open any Browser and type in the URL => <http://localhost>.

Note: If IIS is not installed on our machine, we get the below error: “**HTTP Error 404. The requested resource is not found.**”

Installing IIS on our machine if not installed: Go to **Control Panel** => Click on **Programs and Features** => In the window opened click on “**Turn Windows features on or off**”, which opens another window called “**Windows Features**” => In the window opened select the CheckBox “**Internet Information Services**” and also select each and every Sub-Item (Checkbox's) inside of it and click on the **Ok** button which will install **IIS** on your machine.

How to access a Web Page using IIS?

Ans: When we install **IIS Web Server** on our Machine it will provide us an **admin console** for managing **IIS** and we call it as “**IIS Manager**”, which can be launched by searching for “**inetmgr**” in the window search. Once **IIS Manager** is opened, in LHS of the window we find “**Connections Panel**” and in that **Panel** we find our **Computer Name** as **Server Name** and when we expand it, we find a node called as “**Sites**” and under that we find a **website** with the name “**Default Web Site**”.

What is a Web Site?

Ans: **Web Site** is a collection of **Web Pages**, and a **Web Server** is a collection of **Web Sites**, i.e., a **Web Server** can contain 1 or more **Web Sites** under it and by default when **IIS** is installed on a machine there will be 1 **Website** already created, with the name as “**Default Web Site**”.

To access Web Pages thru IIS Web Server, do the following:

Step 1: Create a folder on your PC in any drive naming it as “**ASP**”.

Step 2: Create an **HTML Page** with the name “**Login.html**” with the below code and save it into “**ASP**” folder.

```
<!DOCTYPE html>
<html>
<head>
<title>Login Form</title>
</head>
<body>
<table align="center">
<caption>Login Form</caption>
<tr>
```

```

<td>User Name:</td>
<td><input type="text" id="txtName" name="txtName" /></td>
</tr>
<tr>
<td>Password:</td>
<td><input type="password" id="txtPwd" name="txtPwd" /></td>
</tr>
<tr>
<td align="center" colspan=2>
<input type="submit" value="Login" />
<input type="reset" value="Reset" />
</td>
</tr>
</table>
</body>
</html>

```

If we want to access Web Pages thru Web Server, we are provided with different options:

Option 1: Accessing thru “Default Web Site”: because this **Website** is already created under **IIS Web Server** we can access our **Web Pages** thru the **Site** and to do that we need to copy our **Web Pages** to a folder that is linked with this **Website** i.e., “**<OS Drive>:\inetpub\wwwroot**” and if we copy our **Web Pages** into this folder we can access them thru **IIS** using their “**Virtual Path**” either from a local or a remote machine also. To test this let’s copy our “**Login.html**” file into this folder and then access it thru the **Virtual Path** of the file as following:

Local Machine => <http://localhost/Login.html> **Or** <http://Server/Login.html>
Remote Machine => <http://Server/Login.html>

Note: for every **Web Site** that is created on **IIS**, there will be 1 associated folder on the **Hard Disk** and all the **Web Pages** of that **Site** should be placed into that folder, right now “**Default Web Site**” is mapped with the “**wwwroot**” folder and that is the reason why we placed our “**Login.html**” Page into that folder. To view the mapping folder of “**Default Web Site**”, right click on the “**Default Web Site**” in “**IIS Manager**” Window => select “**Manage Website**” and under that select “**Advanced Settings**” which opens a window and in that we find “**Physical Path**” property and beside that we find the folder that is mapped to this Web Site.

Option 2: Accessing them thru an **Application** or **Virtual Directory** created under **Default Web Site**: in this case without copying all the **Web Pages** into “**inetpub\wwwroot**” folder as we did in the previous case, we can create an “**Application or Virtual Directory**” under “**Default Web Site**” and map them to our physical folder where we have saved our **Web Pages** i.e., “**ASP**” Folder.

Note: An **Application** or **Virtual Directory** is just a **Sub-Item** under the **Default Web Site** mapped with a **physical folder**.

To create an **Application or Virtual Directory**, Right Click on “**Default Web Site**” in “**IIS Manager**”, select the option “**Add Application**” or “**Add Virtual Directory**” which opens a Window and in that, in “**Alias**” **TextBox** enter a name of your choice and under “**Physical Path**” **TextBox** enter the **physical path** of your folder i.e., “**<drive>:\ASP**”.

Now following the above process, create 1 Application with the name "Site1" and create 1 Virtual Directory with the name "Site2" and map both to our physical folder i.e., "ASP". From now we can access the page of this folder in any of the following ways:

Local Machine: <http://localhost/Site1/Login.html> Or <http://Server/Site1/Login.html>

Local Machine: <http://localhost/Site2/Login.html> Or <http://Server/Site2/Login.html>

Remote Machine: <http://Server/Site1/Login.html>

Remote Machine: <http://Server/Site2/Login.html>

Option 3: Accessing the Web Pages by creating a Site. In this case we create a new Site under IIS just like the existing site i.e., Default Web Site and map it to a physical folder, but the difference is; in this case we can give our own Host Name or Domain Name just like "localhost" which is the Host Name or Domain Name for "Default Web Site" or we can use the same Host Name i.e., "localhost" and change the Port No.

Creating Dynamic Web Pages: These are pages that are created based on a client's request, and the content of these pages will change based on the request i.e., the output of the page will change time to time. For example, a transaction report in a Bank's Web Site.

Starting Date: 01/04/2021

Ending Date: 31/03/2022

Transaction Value: * >= 15000

Transaction Value: * <= 30000

*Optional

How to create a dynamic Web Page?

Ans: To create a dynamic Web Page along with HTML, Client-Side Scripting Languages (Java Script) and CSS we also require taking the help of some Server-Side Technologies. There are so many Server-Side Technologies that are available in the industry for creating dynamic Web Pages like: ColdFusion, Perl, PHP, Classical ASP, JSP, Servlets, ASP.NET, ASP.NET Core, Django, etc.

- | | |
|--|-----------------------------------|
| 1. HTML | => Static Web Pages |
| 2. HTML + CSS | => Static Web Pages (Beauty) |
| 3. HTML + CSS + Java Script | => Static Web Pages (Interactive) |
| 4. HTML + CSS + Java Script + Server Side Technologies | => Dynamic Web Pages |

Every Server-Side Technology uses some programming Language for implementing the logic, for example:

- ColdFusion => CFML
- Perl => Perl Script
- PHP => PHP Script
- Classical ASP => VB Script or JScript
- JSP and Servlets => Java
- ASP.NET & ASP.NET Core => .NET Languages
- Django => Python

ASP.NET is again divided into 2 parts like:

1. ASP.Net Web Forms
2. ASP.Net MVC

ASP.NET Web Forms:

- It is an **Open-Source Server-Side Web Application Framework** designed by **Microsoft** for web development to produce dynamic **Web Pages**.
- It's designed to allow programmers to build **Web Sites/Web Applications** and **Web Services**.
- It was 1st released in **Jan 5, 2002**, with 1.0 version of .NET Framework and this is a successor to **Microsoft's Classical ASP** technology.
- The current version of **ASP.NET** is **4.8** released on **October 2019** which will be the last version and it is succeeded by **ASP.NET Core**.
- **ASP.NET** is built on **Common Language Runtime (CLR)**, allowing programmers to write **ASP.NET** code by using any of the supported **.NET Languages** and while coding an **ASP.NET Application**, we will have access to all **Libraries of ".NET Framework"** which enable us to develop **Web Applications** and then benefit from **CLR, Type Safety, Inheritance, and all Object-Oriented Programming** features.
- **ASP.NET** is a unified Web Development Framework which includes all the services that are necessary for us to build an Enterprise Class Web Application's with minimum volume of coding.

How to create Dynamic Web Pages by using ASP.NET Web Forms?

Ans: To create a **Dynamic Web Page** by using **ASP.NET Web Forms** we need to follow the below process:

1. Save the Page with **".aspx"** extension and that Page can contain **HTML, Java Script, CSS and ASPX Code**.
2. Implement the logic (**ASPX Code**) that must be executed on the Server in any of the following ways:

```
<script language="language" runat="server">
    -Server-Side Logic (ASPx Code)
</script>
Or
<%
    -Server-Side Logic (ASPx Code)
%>
```

Language attribute is to specify in which language we want to implement the logic and it can either be **VB** or **CS**, but the default is **VB**. **Runat** attribute is to specify that this logic should be executed by the **Web Server**.

Note: in-case we are implementing the logic by using second approach we don't require to specify **runat** attribute and the default language will be **VB**.

The extension of a **Web Page** plays a very crucial role while creating dynamic web pages i.e., if the extension of Web Page is **".html"**, Web Server will not process the logic that we have implemented by **ASP.NET** and to test that, write the below code in a notepad, save it as "**Test.html**" and access it thru **Web Server**.

```
<!DOCTYPE html>
<html>
    <head>
        <title>First Dynamic Web Page</title>
    </head>
    <body>
```

```
<h1 style="text-align:center;color:red;background-color:yellow">Naresh I Technologies</h1>
Server Date: <% Response.Write(DateTime.Now.ToShortDateString()) %> <br />
Server Time: <% Response.Write(DateTime.Now.ToString()) %>
</body>
</html>
```

Note: save the above page in our “ASP” folder and access it from browser in any of the following ways:

http://localhost/Site1/Test.html	//Accessing thru Application created.
http://localhost/Site2/Test.html	//Accessing thru Virtual Directory created.
http://localhost:90/Test.html	//Accessing thru Site created with different Port No. 90
http://nitsite.com/Test.html	//Accessing thru Site created.

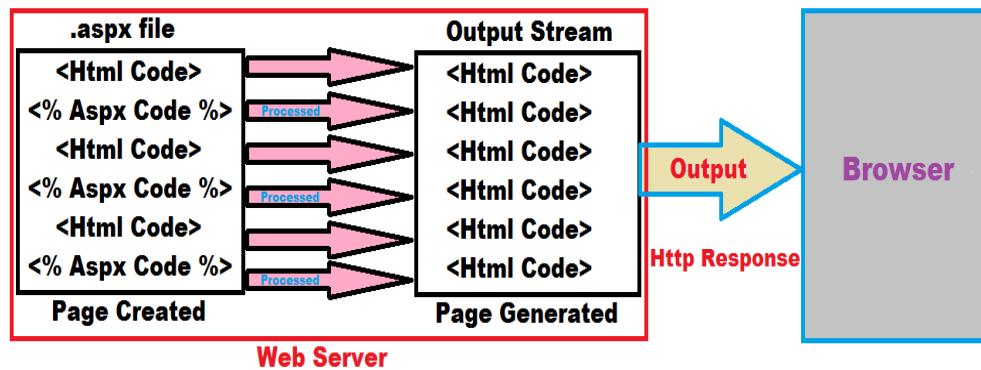
In the above case even if the **Web Page** contains **ASPX Code** in it, **Web Server** did not process that code because of the page **extension**, so it will send the content of page “as is” to the **Browser** for **publishing**. When we send a request from any **Browser** to **Web Server** for any “.html” page, **Web Server** will immediately send content of that page as **Response** to the **Browser** without checking what is present in the file, so the **ASPX Code** we implemented in the file is not processed by the server but displayed directly on the **Browser**.

Now use “Save As” option in notepad and save “Test.html” as “Test.aspx” in the same folder i.e., “ASP”, and when we access the **Page** from **Browser**, we don’t see the logic, but what we see is the result when accessed:

http://localhost/Site1/Test.aspx
http://localhost/Site2/Test.aspx
http://localhost:90/Test.aspx
http://nitsite.com/Test.aspx

When we send request from a **Web Browser** to **Web Server** for any “.aspx” page, **Web Server** will open the file, reads line by line from the file and writes the HTML Code “as is” into the **Output Stream**, whereas if it finds any **ASPX Code**, it will process that logic and writes the **results** that are obtained by processing, into the **Output Stream** as **Text (HTML)**, so that after processing the whole page, content in the **Output Stream** will be sent to **Browser** as **Response**.

ASP.NET Renders HTML: Unfortunately, **internet** still has **bandwidth limitations** and not every person is running on the same **OS**, same **Web Browser** or same **Device**, and these issues make it necessary to stick with **HTML** as our **mark-up** language of choice. So, in all the **Server-Side** technologies including **ASP.NET**; **Web Server** will process all the logic implemented by us using any technology and converts the result into **Text (HTML)** which we call it as **Rendering** and then that **HTML** will be sent to **Clients** as **Response**.



What is the default language used in the creation of ASPX Pages?

Ans: The default language that is used in the creation of ASPX Pages is VB (VB.NET).

How to use C# as the language for creation of ASPX Pages?

Ans: If we want to use C# language for creation of ASPX Pages we need specify that by using “Page Directive” on top of the page and by using this we can specify the language we want to use for development of the page with the help of its language attribute as following:

```
<%@ Page Language="C#" %>
```

To test this, write the above statement on the top of our “`Test.aspx`” file we have created earlier and immediately we get an error when we run the page stating that “`;`” is expected and to resolve the problem end the below 2 statements with “`;`”:

```
<% Response.Write(DateTime.Now.ToShortDateString()); %>  
<% Response.Write(DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss")); %>
```

Note: a directive is an instruction that we give to the Web Server. By using Page directive, we are telling the Web Server that the content in this page is implemented by using the specified language i.e., “C#” so that Web Server will use the appropriate language compiler to compile the page.

What is “Response” in our above code?

Ans: Response is an instance of a pre-defined class “`HttpResponse`”, and we call it as an **intrinsic object** and apart from this Response we also have 7 other **intrinsic objects** like: Request, Server, Session, Application, Trace, User and Cache, which can be directly consumed in our Web Pages. Each **intrinsic** object is internally an **instance** of some class like “`Request`” is an instance of class “`HttpRequest`”, “`Response`” is an instance of class “`HttpResponse`”, and “`Server`” is an instance of class “`HttpServerUtility`” and so on.

How intrinsic objects are accessible to our ASPX Pages?

Ans: Intrinsic objects are accessible to our ASPX Pages from its parent class i.e., every ASPX Page after its compilation gets converted into a class and that class will internally inherit from a pre-defined class known as “Page”, which is defined in “System.Web.UI” namespace of BCL (Base Class Libraries) and all the above 8 intrinsic objects are inherited from that “Page” class to our ASPX Pages.

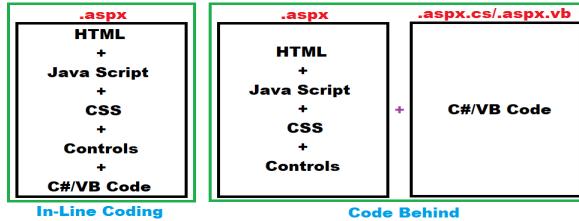
Coding Techniques in ASP.NET: ASP.NET supports 2 different coding techniques for creating Web Pages, those are:

- ## 1. Inline coding technique

2. Code behind technique

Inline Coding: in this case **HTML**, **Java Script**, **CSS**, **Controls** and **Business Logic** (implemented by **C#** or **VB**) will be present in 1 file which is saved with “**.aspx**” extension. The “**Test.aspx**” page we have created above is an example for “**Inline Coding**”.

Code Behind: in this case **HTML**, **Java Script**, **CSS**, and **Controls** will be present under 1 file which is saved with “**.aspx**” extension and **Business Logic** (implemented by **C#** or **VB**) will be present in a separate file which is saved with “**.aspx.cs**” or “**.aspx.vb**” based on the **language** in which we are implementing the **Business Logic**.



Developing Web Pages using Visual Studio: Open Visual Studio => select “Create a new project” option => now under the Window opened, choose “C#” under “All _Languages DropDownList”, choose “Windows” under “All _Platforms DropDownList”, choose “Web” under “All Project _Types” DropDownList and now in the below choose “ASP.Net Web Application (.NET Framework)” and click “Next”, which launches a new window, enter **Project name** as “FirstWebApp”, Location as “<drive>:\ASP”, under Framework choose the latest Framework Version and click on the “Create” button which launches a new Window and in that choose “Empty” Project Template and in the RHS select “Web Forms” CheckBox and make sure all the other Checkbox’s are un-checked and click on “Create” Button which will create and opens the new project.

Right now, the project which is opened is empty which doesn’t contain any **Web Pages** in it and if we open the **Solution Explorer** and watch, we will find 2 empty folders here with the names “**App_Data**” & “**Models**” which can either be deleted or leave them as is, and under the project we will also find 3 files with the names “**Global.asax**”, “**Packages.config**” & “**Web.config**” which are required under every **ASP.NET Web Application**.

Note: Every “**ASP.Net Web Application**” project is a collection of “**Web Pages**” and call them as “**Web Forms**”.

Adding a Web Form in our project: right now our project doesn’t contain any **Web Form’s** in it, and if we want to add a new **WebForm**, open **Solution Explorer** right click on the project and Select Add => “**New Item**” which opens the “**Add New Item**” window, choose **Web Form** in it, which will ask for a name, default will be “**WebForm1.aspx**”, change it as “**First.aspx**” and click on the **Add** button.

By default, the page “**First.aspx**” contains “**Page Directive**” on top of the page with few attributes like: “**Language**”, “**AutoEventWireup**”, “**Code Behind**” and “**Inherits**”, below that we find some **HTML Code** containing **Head** and **Body** sections.

To view the **Code Behind** file associated with this **WebForm** right click on the **WebForm** in document window and select “**View Code**” option which will launch the file “**First.aspx.cs**” that contain a class in it with the name “**First**” inheriting from the “**Page**” class and notice, that class First is a partial class i.e., it is defined on more than one source files.

1. First.aspx.cs

2. First.aspx.designer.cs

Note: by default, “First.aspx.cs” file is only visible to us and if we want to view “First.aspx.designer.cs” file, open Solution Explorer and expand “First.aspx” node and under that we find both the files. “Designer.cs” files are used by **Visual Studio** to auto-generate code i.e., the code in these files will be generated by **Visual Studio**, so never edit the content of this files.

Writing code in a WebForm: because our page is created in **Code Behind** model, we can implement our logic either in “.aspx” file or “.aspx.cs” file also. For example, if we want to implement logic in “First.aspx”, then write the below code under “<div>” tag which we find under the “<body>” tag:

```
Server Date: <% Response.Write(DateTime.Now.ToShortDateString()); %> <br />
Server Time: <% Response.Write(DateTime.Now.ToString()); %>
```

Note: to run the **Web Page** hit F5 and it executes the page and displays the output on browser.

If we want to implement logic in “.aspx.cs” file, then add another new WebForm in the project naming it as “Second.aspx”, go to its Code View and write the below code under the method “**Page_Load**” which we find in the file “Second.aspx.cs”:

```
Response.Write($"Server Date: {DateTime.Now.ToShortDateString()}");
Response.Write("<br />");
Response.Write($"Server Time: {DateTime.Now.ToString()}");
```

How does a Web Application run under Visual Studio?

Ans: To run a **Web Application** we require a **Web Server** and without that **Web Server** we can't run any **Web Application**, so **Visual Studio** by default provides a built-in **Web Server** to run the **Web Application's** that we develop under **Visual Studio** i.e., “**IIS Express**” and right now our project “FirstWebApp” is running thru that “**IIS Express Web Server**” only.

Visual Studio will add each **Web Application** to **IIS Express** as 1 **Site** by allocating 1 random **Port** to each **Application**, which will be different from **Project** to **Project** and **Machine** to **Machine**. To view the **Port** that is allocated for our **Web Application** run any **Web Form** in our project and watch the URL in browsers address bar which will be as following:

<http://localhost:YourPort/First.aspx>

<http://localhost:YourPort/Second.aspx>

Note: IIS Express is provided for testing our **Web Applications** at the time of development and we call this as a “**Development Web Server**” also and once the **Application** development is completed, we can host our **Web Application** either on the “**Local IIS**” of our machine or “**IIS**” on remote machines or on some **Public Server** also.

How to host our Web Application into Local IIS?

Ans: as discussed above our **Web Application** is right now running under **IIS Express** of **Visual Studio**, and it's possible to host the **Application** on **Local IIS** of our machine and to do that first close your **Visual Studio** and re-open it in **Administrator** mode and to do that, right click on the **Visual Studio** and select the option “**Run as Administrator**”. Now open **Solution Explorer**, right click on the project and select “**Properties**” option which will launch **Project Property Window**, in that window on the **LHS** we find “**Web**” tab, click on it and now on the right

scroll down to “Servers” section, there we find a “Drop Down List” control and in that by default “IIS Express” is selected and in the Project URL TextBox it will be showing the current Project’s URL i.e.,: <http://localhost:Port/>.

Now in the “DropDownList” select “Local IIS” option which will change the Project URL as following: <http://localhost/FirstWebApp>, and beside the Project URL Textbox we find a button “Create Virtual Directory” click on it, which will create an “Application” under the “Default Web Site” of “Local IIS”, now click on the Save button in “Visual Studio Standard Toolbar” and from now our Web Application will be running thru Local IIS and the URL when we run our Web Forms will be as following:

<http://localhost/FirstWebApp/First.aspx>
<http://localhost/FirstWebApp/Second.aspx>

Working with Web Forms: every WebForm in an ASP.NET Web Application project will be having 3 places to work with, those are:

1. Source View
2. Design View
3. Code View

Source View: when a new Web Form is added by default its display’s the Source View only, which is nothing but the “.aspx” file and we use this for writing HTML, CSS, Java Script as well as all the logic for creation of Controls.

Design View: This is a visual representation of Source View i.e., whatever we write in Source View will be displayed as output in Design View just like how output is displayed on Browser and because of this feature Visual Studio IDE is known as WYSIWYG (What You See Is What You Get) IDE. We can also use this for designing the UI’s, by a feature called “Drag and Drop”.

Note: To navigate from “Source View” to “Design View”, in the bottom of “.aspx” file we find 3 options: Design, Split & Source. By default, it will be pointing to Source, click on Design which will take us to the Design View or click on Split to see both at a time in top and bottom.

Code View: this is nothing but “.aspx.cs” file of the Web Form and we use this for implementing all business logic using C# Language.

ASP.NET Server Controls: in ASP.NET to design the UI’s we don’t use HTML Controls at all, and in that place, we are provided with “ASP.NET Server Controls” for designing UI’s. The advantages of using ASP.NET Server Controls over HTML Controls are:

1. HTML Controls will lose the state of their values once the Web Page is submitted to Server, whereas ASP.NET Server Controls will maintain the state of their values even after page submission.
2. HTML Controls can’t be accessed directly in C# Code whereas ASP.NET Server Controls are directly accessible in the C# Code and, we have the advantage of intellisense listing members of that control.

We are provided with various Controls in ASP.NET for designing UI’s and all those Controls are pre-defined Classes in our Libraries, which are defined under the namespace “System.Web.UI.WebControls”. ASP.NET Server Controls are divided into different categories like:

1. Standard Controls
2. Data Controls
3. Validation Controls
4. Navigation Controls

5. Login Controls
6. Web Parts Controls
7. Ajax Extension Controls
8. Dynamic Data Controls

Note: we have been provided with number of **Controls** that are developed by using **C# Language**, but all those controls at the time of page processing will be rendering required **HTML** and the logic for rendering is implemented in a method of that **Control** class i.e., “**RenderControl**” and this method will be called by **ASP.NET Framework** while processing the page.

To check **ASP.NET Server Controls** renders **HTML**, add a new **Web Form** in the existing project i.e., “**FirstWebApp**”, naming it as “**Fourth.aspx**” and write the below code under the “**<div>**” tag we find in **Source View**:

```
Enter Name: <input name="txtName1" type="text" id="txtName1" />
<input type="submit" name="btnSubmit1" value="Save" id="btnSubmit1" />
<br />
Enter Name: <asp:TextBox ID="txtName2" runat="server" />
<asp:Button ID="btnSubmit2" runat="server" Text="Save" />
```

Run the above page, we see 2 **TextBox’s** and 2 **Button’s** on the page, now right click on the **Browser** and select the option “**View Page Source**” which will display the **Source Code** of this page and notice in that code; **ASP.Net Server Controls** also will be showing code in **HTML format** only because they are rendered and this is how we see the code there, under the last **<div>** tag.

```
Enter Name: <input name="txtName1" type="text" id="txtName1" />
<input type="submit" name="btnSubmit1" value="Save" id="btnSubmit1" />
<br />
Enter Name: <input name="txtName2" type="text" id="txtName2" />
<input type="submit" name="btnSubmit2" value="Save" id="btnSubmit2" />
```

Working with ASP.NET Server Controls: every **ASP.NET Server Control** is associated with 3 things in it:

1. Properties
2. Methods
3. Events

- **Properties** are nothing but attributes of a control which defines the look of a control. E.g.: **Width**, **Height**, **BackColor**, **ForeColor**, **BorderColor**, **BorderStyle**, **Font**, etc.
- **Methods** are actions performed by the control when we call them. E.g.: **Focus()**, **Clear()**, etc.
- **Event** is a time period which tells when an **action** has to be performed. E.g.: **Click of a Button**, **Load of a Page**, **TextChanged of a TextBox**, etc.

Note: The parent class for all **ASP.NET Server Controls** is class **Control** of “**System.Web.UI**” namespace, which contains all the common **Properties**, **Methods**, and **Events** of all **ASP.NET Server Controls**.

Placing an ASP.NET Server Control on a Web Form: to work with **ASP.NET Server Controls** create a new empty “**ASP.NET Web Application (.NET Framework)**” project, naming it as “**ControlsDemo**”, choose **Empty Project**

Template and select the “Web Forms - Check Box”, uncheck all the other **Checkbox**’s and click on Create button. In the project first delete the existing folders, and then add a Web Form, naming it as “**ControlCreation.aspx**” and delete the “**<div>**” tag that is present inside the “**<form>**” tag. We can place a control on our **Web Form** either thru “**Design View**” or “**Source View**” or “**Code View**” also.

Design View: go to **Design View** of our **Web Form** and in the **LHS** we find a window called “**Toolbox**”, open it and in that window under the “**Standard Tab**” we find a list of controls, either double click on the **Button** control or **Drag and Drag** the **Button** control on to the **Design View**.

Note: this action will generate all the required “**ASPX Code**” for creation of the **Button** and to view that code go to **Source View** of the page and there we find the below code:

```
<asp:Button ID="Button1" runat="server" Text="Button" />
```

Source View: same as the above we can also write code for creation of a control just like what VS has generated and to test that write the below code under existing “**Button1**” in **Source View**:

```
<asp:Button ID="Button2" runat="server" Text="Button" />
```

Note: Every control we place on a **WebForm** is an instance of an appropriate control class, so in the above case “**Button1**” and “**Button2**” are 2 instances of **Button Class**.

Code View: we can also explicitly create the instance of any control class thru **C# Code** and add it on the **Web Form** and to test this process go to “**.aspx.cs**” file of our **Web Form** and write the below code in “**Page_Load**” method which is present there:

```
Button Button3 = new Button();
Button3.ID = "Button3";
Button3.Text = "Button";
form1.Controls.Add(Button3);
```

Note: now run the **Web Form** and you will see all the 3 buttons on the **Browser Window**.

Working with properties of Controls: every ASP.Net Server control is associated with “n” no. of properties and all those properties will define the look of the control. We can set properties to a control also in 3 different ways.

Using Design View: go to **Design View** of the “**ControlCreation.aspx**” **Web Form**, select **Button1** and hit **F4** which opens the **Properties Window** on RHS, listing all the properties of **Button1** and in that window we find Control properties like **BackColor**, **BorderColor**, **BorderStyle**, **BorderWidth**, **FontSize**, **ForeColor**, etc, change any of the required property values and view the output directly in **Design View**.

Note: above action will generate all the required code in **Source View** as per our settings, to verify that go to **Source View** and watch the code added to **Button1** as following:

```
<asp:Button ID="Button1" runat="server" Text="Button" BackColor="Yellow" BorderColor="Blue"
BorderStyle="Dotted" BorderWidth="5px" Font-Size="XX-Large" ForeColor="Red" />
```

Using Source View: same as **Visual Studio** generated the code for settings of **Button1** in **Source View**, we can also write manual code to perform property settings and to test that add the following code for **Button2** tag, which should now look as below:

```
<asp:Button ID="Button2" runat="server" Text="Button" BackColor="Yellow" BorderColor="Blue" BorderStyle="Dashed" BorderWidth="5px" Font-Size="XX-Large" ForeColor="Red" />
```

Using Code View: we can also set properties to controls by writing “C# Code”, to test that go to “ButtonCreation.aspx.cs” file and write the below code in “Page_Load” method above the statement “form1.Controls.Add(Button3);”:

```
Button3.BackColor = System.Drawing.Color.Yellow;
Button3.BorderColor = System.Drawing.Color.Blue;
Button3BorderStyle = BorderStyle.Double;
Button3.BorderWidth = Unit.Pixel(5);
Button3.Font.Size = FontUnit.XXLarge;
Button3.ForeColor = System.Drawing.Color.Red;
```

Note: run the web page again to watch the output.

Working with Events of Control: An **Event** is a **time-period** which tells when an action must be performed i.e., when a method must be executed. Every **Control** is associated with a set of **events** and each **Event** will fire at some point of time which is based on **user interactions**, so we can bind **methods** with these **events** and those methods will execute whenever the event fires.

Note: in **GUI (Graphical User Interface)** programming we don’t call methods directly, but we bind the methods with events and those methods gets executed whenever the event fires and every event will fire at some point of time. **Events** are already defined under the **Control** classes, so we need to define methods in our page class and then bind those methods with events.

We can define methods for events in 3 different ways:

Using Design View: go to Design View of “ControlCreation.aspx”, select **Button1**, hit **F4**, in the property window opened we find “**Events**” option on the top, select it which will show events of **Button** and every **Event** will fire at some point of time which can be identified by watching the description below. Now double click on **Click Event** which will generate a method in “**aspx.cs**” file for implementing the logic that should be executed when the End user clicks on the Button and the name of that method will be “**Button1_Click**” which follows a pattern i.e., “**<Control Name>_<Event Name>**”, now write the below code in that method:

```
Response.Write("<script>alert('Button1 is clicked.')</script>");
```

Note: now if we go to Source View and watch we find “**Button1_Click**” method bound to the Button’s click event as following:

```
<asp:Button ID="Button1" runat="server" ..... OnClick="Button1_Click" />
```

Using Source View: we can generate methods for events thru **source view** also and to do that go to **source view** and add the following code to **Button2** tag in the end:

```
OnClick="Button2_Click"
```

Note: when we type “**OnClick=**”, Intellisense will display the option “**<Create New Event>**” select it, which will automatically generate the above code. Now in **code view** we find our method “**Button2_Click**” for implementing the logic, write the below code in it:

```
Response.Write("<script>alert('Button2 is clicked.')</script>");
```

Using Code View: we can generate methods for events in **code view** also, and to do that write the following statement in “**Page_Load**” method above the statement “**form1.Controls.Add("Button1");**”:

Type **Button3.Click +=** and hit the tab key => which will change as **Button3.Click += Button3_Click;**

This above action will generate a method with the name “**Button3_Click**”, write the below code under that method:

```
Response.Write("<script>alert('Button3 is clicked.')</script>");
```

Note: the methods under which we are implementing the logic for an **Event** are known as “**Event Handlers**” and all these methods are **non-value returning** and every method takes **2 parameters** in common, those are:

1. Object
2. EventArgs or a child class of EventArgs

Default Events: every control is associated with multiple events with it and to generate a “**Event Handler**” for any of those events we need to follow any of the 3 processes that are described above, whereas for every control there will be **1 default event** and in case we want to generate “**Event Handler**” to that default event without following any of the options that are described above, we can directly **double click** on the **Control** in design view which will generate the required **Event Handler**.

Control	Default Event
Button, LinkButton, ImageButton	Click
TextBox	TextChanged
CheckBox and RadioButton	CheckedChanged
DropDownList, ListBox, CheckBoxList and RadioButtonList	SelectedIndexChanged
Calendar	SelectionChanged

Label Control: this control is used for displaying **static text** on the **UI** and we set the **text** value by using the “**Text**” property of the control.

Button Control: this control is designed for **submitting** a page to the **Server** and under this family we have 3 Classes (**Controls**): **Button**, **LinkButton** and **ImageButton**, and when we place any of the above 3 button on a **Web Form** they will render all the required **HTML Code** and converts as following:

Button	=>	Html Input-Type Submit
LinkButton	=>	Html Hyperlink
ImageButton	=>	Html Input-Type Image

Note: to use **ImageButton** we need to set the **ImageUrl** property and to do that open **Solution Explorer**, right click on the **Project**, select “**Add**” => “**New Folder**”, which will add a new **Folder** in the project, name it as “**Images**”. Now right click on the **Images** folder and select “**Add**” => “**Existing Item**” which will open a dialog box, select an image from your hard disk and it will add into the folder.

To work with “**Button**” controls add a new **Web Form** in the project naming it as “**ButtonDemo1.aspx**” and write the below code under “**<div>**” tag:

```
<asp:Button ID="btnClick1" runat="server" Text="Click Me" />
<asp:LinkButton ID="btnClick2" runat="server" Text="Click Me" />
```

```
<asp:ImageButton ID="btnClick3" runat="server" ImageUrl="~/Images/Nike.jpg" Width="50" Height="50" />
```

When we run the page, it will render the below code and we can view that by using “View Page Source” option of browser:

```
<input type="submit" name="btnClick1" value="Click Me" id="btnClick1" />
<a id="btnClick2" href="javascript:_doPostBack(&#39;btnClick2&#39;,&#39;&#39;)">Click Me</a>
<input type="image" name="btnClick3" id="btnClick3" src="Images/Nike.jpg" style="height:50px;width:50px;" />
```

What is meant by submitting a page to Server?

Ans: submitting a page to server means transferring all the values of a page that are entered into the controls from Client’s Browser to the Web Server.

Page Submission is of 2 types:

1. Postback Submission
2. Cross-Page Submission

Note: Postback submission is a process of submitting a page back to itself whereas cross page submission means it is a process of submitting a page to another page.

```
Page1 => Submitting to Page1           //Postback Submission
Page1 => Submitting to Page2           //Crosspage Submission
```

By default, **Button** control submits a page back to itself (**Postback**) and it is also capable of submitting the page to other pages also (**Cross Page**).

Understanding “Post Back Submission” and “Cross Page Submission”: add a new **Web Form** in our project naming it as “**ButtonDemo2.aspx**” and write the below code under its “**<div>**” tag:

```
<asp:Button ID="Button1" runat="server" Text="Postback Submission" />
<asp:Button ID="Button2" runat="server" Text="Crosspage Submission" />
```

Right now, both the **Buttons** will perform **Postback Submission** only, and to check that, go to “**ButtonDemo2.aspx.cs**” file and write the below code under “**Page_Load**” Event Handler:

```
if (IsPostBack)
    Response.Write("This is a post or post back request.");
else
    Response.Write("This is a first or get request.");
```

Note: “**IsPostBack**” is a property of our parent class (**Page**) which returns **true** if the current request is a **Post** or **Postback** request and **false** if it is a **First** or **Get** request. First request to a page is called as “**Get Request**” and the next requests to a page are called as “**Post Request**”.

Now run the **Web Form** and watch the output which will initially display the value as “**This is a first or get Request.**” which indicates it’s a first request and when we click on any of the **2 buttons** it will display the value as “**This is a post or post back Request.**” which indicates it’s a post back request.

Right now, both buttons are performing post back submission only whereas if we want the 2nd Button to perform a cross page submission then add a new **WebForm** in the project naming it as “**ButtonDemo3.aspx**” and write the following code under its “**Page_Load**” Event Handler in **Code View**:

```
Response.Write("<font color='red'>Cross Page Submission.</font>");
```

Now come to Design View of “**ButtonDemo2.aspx**”, select 2nd Button, go to its properties, identify the “**PostBackUrl**” property, select it, which will display a Button beside, click on it which will launch a new screen and in that select “**ButtonDemo3.aspx**” and then run “**ButtonDemo2.aspx**” form, and when we watch the output i.e. when we click on “**Button1**” it will perform Post Back Submission and when we click on “**Button2**” it will perform Cross Page Submission and launches “**ButtonDemo3.aspx**”.

Important members of Button Control:

1. **Text:** It is a property using which we can associate some data with the control that is visible to end users and this works as a caption of the button.
2. **PostBackUrl:** It is a property using which we can specify the Address of target page to whom the current page has to be submitted when the button is clicked.
3. **OnClientClick:** It is a property using which we can specify the name of a **JavaScript** function which has to be executed when the button is clicked, i.e., before submitting the page to server.
4. **CommandName & CommandArgument:** These 2 properties are used for associating some additional data with the button apart from the Text Property but these are not visible to end users and these values can be used in code under **Command** Event.
5. **Click:** It is an **event** which **fires** when the button is **clicked**.
6. **Command:** This is also an **event** which fires when the button is clicked and there are associated **CommandName** or **CommandArgument** values.

To understand about the “**OnClientClick**” property, add a new **Web Form** in the project naming it as “**ButtonDemo4.aspx**” and write the below code under its “**<div>**” tag:

```
<asp:Button ID="Button1" runat="server" Text="Click Me" />
<asp:Label ID="Label1" runat="server" ForeColor="Red" />
```

In this **Web Form** when we click on **Button** it has to display a message in **Label Control** saying that, the page has been submitted to server. To achieve it generate a “**Click Event Handler**” for **Button Control** and write the below code in it:

```
Label1.Text = "Page is submitted to server.;"
```

Now when we run the **Form** and click on the **Button** control, **Label** control will display the message. But when the user clicks on the **Button** first it must ask for a confirmation about submitting the page to server and if the user accepts it, then only page should be submitted to server. This can be done by using **Java Script** code and to do that, go to **Source View** of “**ButtonDemo4.aspx**” and write the below code under “**<head>**” section of page:

```
<script>
function Confirmation() {
    var Result = confirm("Are you sure of submitting the page to server?");
    return Result;
}
</script>
```

Now to execute this code when the **Button** is clicked, go to **Design View** of the page, select the **Button**, go to its **Properties**, select “**OnClientClick**” property and type the name of **Java Script** function we defined, as following: “**return Confirmation();**”. Now run the **Web Form** again and check the output.

Understanding about Object and EventArgs parameters of Event Handlers: as discussed earlier every Event Handler will be taking 2 parameters i.e., “sender” of type “object” and “e” of type “EventArgs” or child class of “EventArgs”. The use of these parameters is we can implement logic for multiple controls in a single Event Handler.

1. **“object sender”:** to understand about this parameter, add a new WebForm in the project naming it as “Calculator1.aspx” and write the below code in its “`<div>`” tag:

```
<table align="center">
    <caption>Calculator</caption>
    <tr>
        <td>Enter 1st number:</td>
        <td><asp:TextBox ID="txtNum1" runat="server" /></td>
    </tr>
    <tr>
        <td>Enter 2nd number:</td>
        <td><asp:TextBox ID="txtNum2" runat="server" /></td>
    </tr>
    <tr>
        <td>Result Generated:</td>
        <td><asp:TextBox ID="txtResult" runat="server" /></td>
    </tr>
    <tr>
        <td colspan="2" align="center">
            <asp:Button ID="Button1" runat="server" Text="Add" />
            <asp:Button ID="Button2" runat="server" Text="Sub" />
            <asp:Button ID="Button3" runat="server" Text="Mul" />
            <asp:Button ID="Button4" runat="server" Text="Div" />
            <asp:Button ID="Button5" runat="server" Text="Mod" />
        </td>
    </tr>
</table>
```

Now let's write 1 **Event Handler** method for all the 5 buttons and to do that go to design view of the form, select “Add” button, open property window, select **Events**, select **Click Event** and beside that specify a name to the Event Handler - “Buttons_Click” and click on it which will generate the **Event Handler** with the given name.

Now to bind the **Event Handler** with remaining 4 buttons, go to design view again, select “Sub”, “Mul”, “Div” and “Mod” buttons at a time, open property window, select **Events**, select **Click Event** and now we will find a drop down besides, click on it which will list the Event Handler - “Buttons_Click” select it which will bind the “Event Handler” with all 4 buttons.

Now go to “Calculator1.aspx.cs” file and write the below code over there:

Code under Page_Load:

```
if (!IsPostBack)
    txtNum1.Focus();
```

Code under Buttons_Click:

```
int Num1 = int.Parse(txtNum1.Text);
int Num2 = int.Parse(txtNum2.Text);
int result = 0;
Button b = sender as Button;
if (b.ID == "btnAdd")
    result = Num1 + Num2;
else if (b.ID == "btnSub")
    result = Num1 - Num2;
else if (b.ID == "btnMul")
    result = Num1 * Num2;
else if (b.ID == "btnDiv")
    result = Num1 / Num2;
else if(b.ID == "btnMod")
    result = Num1 % Num2;
txtResult.Text = result.ToString();
```

When an **Event Handler** is bound with multiple **controls** we can identify the **control** which is raising the Event (**Click in our case**) by using the “**sender**” parameter of **Event Handler**, because we are already aware that every **Control** is a class and a control that is placed on the **Form** is an **Instance** of that **class**, so whenever a **Control Instance** raises an **Event** immediately that **instance** is captured in the “**sender**” parameter which of type “**object**” (default parent class of all classes).

Because “**sender**” is of type “**object**” it is not capable of accessing any of the child class members, for example in the above case “**Button**” is the child class, and to overcome this problem we need to convert “**sender**” back into **Button** as we have performed in our above code and then we can read its **ID** or **Text** or any other property and implement the logic specific to the control.

2. “EventArgs (or child class of EventArgs) e”: to understand about this **parameter**, add a new **Web Form** in the project naming it as “**Calculator2.aspx**” and write the below code in its “**<div>**” tag:

```
<table align="center">
<caption>Calculator</caption>
<tr>
<td>Enter 1st number:</td>
<td><asp:TextBox ID="txtNum1" runat="server" /></td>
</tr>
<tr>
<td>Enter 2nd number:</td>
<td><asp:TextBox ID="txtNum2" runat="server" /></td>
</tr>
<tr>
<td>Result Generated:</td>
<td><asp:TextBox ID="txtResult" runat="server" /></td>
</tr>
<tr>
```

```

<td colspan="2" align="center">
<asp:Button ID="Button1" runat="server" Text="Add" CommandName="+" />
<asp:Button ID="Button2" runat="server" Text="Sub" CommandName="-" />
<asp:Button ID="Button3" runat="server" Text="Mul" CommandName="*" />
<asp:Button ID="Button4" runat="server" Text="Div" CommandName="/" />
<asp:Button ID="Button5" runat="server" Text="Mod" CommandName "%" />
</td>
</tr>
</table>

```

In the above case we have set a **CommandName** property for each button and this value can be accessed and used in the **Command Event** of button. Now let's write 1 **Event Handler** method for all the 5 buttons and to do that go to design view of the form, select "Add" button, open property window, select **Events**, select **Command Event** and beside it specifies a name to the Event Handler i.e., "**Buttons_Command**" and click on it which will generate an **Event Handler** with the given name.

Now to bind the Event Handler with remaining 4 buttons, go to design view again, select "Sub", "Mul", "Div" and "Mod" buttons at a time, open property window, select **Events**, select **Command Event** and now we will find a drop down besides, click on it which will list the **Event Handler** i.e., "**Buttons_Command**" select it which will bind the "**Event Handler**" with all 4 buttons. Now go to "**Calculator2.aspx.cs**" file and write the below code there:

Code under Page_Load:

```

if (!IsPostBack)
{
    txtNum1.Focus();
}

```

Code under Buttons_Command:

```

int Num1 = int.Parse(txtNum1.Text);
int Num2 = int.Parse(txtNum2.Text);
int Num3 = 0;
switch(e.CommandName)
{
    case "+":
        Num3 = Num1 + Num2;
        break;
    case "-":
        Num3 = Num1 - Num2;
        break;
    case "*":
        Num3 = Num1 * Num2;
        break;
    case "/":
        Num3 = Num1 / Num2;
        break;
    case "%":
        Num3 = Num1 % Num2;
        break;
}

```

```

        Num3 = Num1 % Num2;
        break;
    }
    txtResult.Text = Num3.ToString();

```

In the above case we have specified a **CommandName** to each Button and we can access that **CommandName** value under the **Event Handler** by using “e” of type **CommandEventArgs** which is nothing but the child class of the **EventArgs** class, so once we can access the **CommandName** we can easily identify the **Control** which is raising the **Event** and implement the logic as we have done in the above case.

Note: Same as **CommandName** we can also use **CommandArguments** property also but **CommandName** is of type **string** and **CommandArguments** is of type **object**. But these 2 can be accessed only under **Command Event**.

In the above **pages** whenever we click on a **button** the whole page is **submitted** to the **server** and this happens every time we **click** on the **button** and **re-loads** the whole page again. To avoid this problem of **reloading** the whole page again, we need to use “**AJAX**”.

AJAX: ASynchronous JavaScript and XML

- AJAX is a technique for creating fast and dynamic web pages. It is a cross platform technology which speeds up the response time.
- AJAX allows Web Pages to be updated asynchronously by exchanging small amount of data with the server behind the screen. This means it is possible to update parts of a Web Page with-out the whole page being updated.
- Ajax is not a single technology, but rather a group of technologies combined together, like:
 1. HTML (or XHTML) and CSS for presentation.
 2. The Document Object Model (DOM) for dynamic display of data and interaction with data.
 3. JSON or XML for the interchange of data, and XSLT for its manipulation.
 4. The XMLHttpRequest object for asynchronous communication.
 5. Java Script to bring all these technologies together.

Note: a variety of popular **Java Script** libraries like **JQuery** provides built-in support and assistance in executing **AJAX** requests. Classical **Web Pages** which do not use **AJAX** used to reload the entire page if any content has to change. **ASP.NET** provides us various controls for using **AJAX** in **ASP.NET** App’s and those controls will add script to the page which is executed and processed by the browser without us implementing all the required logic. In the **ToolBox** we find a “**Tab**” called “**AJAX Entensions**” and under that we find all the **AJAX** controls provided by **ASP.NET** for us to consume.

1. ScriptManager Control: this is the most important control and must be present on the **Web Page** for other **AJAX** Controls to work which takes care of the client-side script for all the Server Side **AJAX** Controls.

2. ScriptManagerProxy Control: this control lets you add scripts and services to **Content Pages** and **User Controls** if the **Master Page** or **Host Page** already contains a **ScriptManager** control and by using this control, you can add to the script and services collections which is defined by the **ScriptManager** control in **Master Page**.

3. UpdatePanel Control: this control is a container control. It just act’s as an **interface** for holding other controls on it and doesn’t have any UI. When a control inside of it triggers a “**Callback**” then the **UpdatePanel** intervenes to initiate the post **asynchronously** and updates just that portion of the page.

4. Timer: this control allows you to do **Callbacks** at certain intervals, if used together with **UpdatePanels** (which is the most common approach) performs timed partial updates of your page, but it can also be used for posting back

the entire page as well. This control uses the interval attribute to define the number of **milliseconds** for firing the **Tick** event.

5. UpdateProgress: this control provides status information about partial-page updates in **UpdatePanel** controls because one of the big problems with **AJAX** is since it's asynchronous and performs actions in background, the browser will not show us any status whereas with fast **servers** and fast methods this is not a big problem, but whenever we have a method which takes up a bit of time, the user is very likely to get impatient which can be resolved with **UpdateProgress**.

To resolve the problem in our previous pages use **ASP.NET AJAX Controls** and to do that rewrite the code that is present in the “**<div>**” tag of our 2 pages “**Calculator1**” & “**Calculator2**” as below:

```
<asp:ScriptManager ID="ScriptManager1" runat="server"></asp:ScriptManager>
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
<ContentTemplate>
    -Put the above table creation code here
</ContentTemplate>
</asp:UpdatePanel>
```

TextBox Control: we use this control for taking **Text** input from the users and we can use this **Control** in 16 different ways like **Single Line Text**, **Multi Line Text**, **Password**, etc.

Important members of TextBox Control are:

1. **TextMode:** By using this property we can set the behavior for the control and this property can be set with any of the following values:

SingleLine [d]	Date	Month	Number
MultiLine	DateTime	Time	Range
Password	DateTimeLocal	URL	Search
Color	Week	Email	Phone

2. **ReadOnly:** It's a **Boolean** property with default value as false and if set as true, control becomes non editable.
3. **Rows:** This property is to specify the no. of display rows we want, when the **TextMode** is set as **Multiline**.
4. **MaxLength:** This is to specify the maximum no. of char's that must be accepted into the Textbox control.
5. **AutoPostBack:** This is a **Boolean** property with default value as false, whereas if we set it as true this control gets the capability of submitting the page to server i.e., it can perform a **Postback**.
6. **TextChanged:** This is the default **Event** of the control which fires when the text property has been changed.

To work with **TextChanged** event of the **TextBox** add a new **WebForm** under the project naming it as “**ColorDialog.aspx**” and write the below code under “**<form>**” tag by deleting existing “**<div>**” tag over there:

```
<div id="div1" runat="server">
<br />
Change Color: <asp:TextBox ID="txtColor1" runat="server" TextMode="Color" />
<br /><br />
</div><asp:Button ID="Button1" runat="server" Text="Button" />
```

Now go to design view of **WebForm**, double click on **TextBox** to generate a “**TextChanged**” “**EventHandler**” for it and write the below code in corresponding “**Event Handlers**” defined in “**aspx.cs**” file:

Code under txtColor1_TextChanged:

```
div1.Attributes.Add("style", "background-color:" + txtColor1.Text);
```

Now run the **Web Form** and select a color under any “**ColorDialog**”, but the **background color** of corresponding “**<div>**” control will not change even if we implemented logic under the “**TextChanged**” event of **TextBox**, because **TextBox** control is not capable of **Submitting** the page to server and to overcome this problem, after selecting a **Color** in **ColorDialog**’s we need to click on the **Button** we placed below which will **Submit** the page to server and then the logic we implemented under “**TextChanged**” event of **TextBox**’s will execute.

How the logic is executing when the Button is clicked?

Ans: When a control raises an event but not capable of submitting the page to server, until the page is submitted to Server the event which is raised gets **cached** and executes when the page is submitted to the server next time and we call them as “**Cached Events**”.

Cached Event: this event will store page data, which gets processed when the page is submitted to the server by a Postback.

In our previous example, if we want the logic to be executed **immediately** after **changing** the **color**, without waiting for the **Button** to click, do the following:

Step 1: Delete the **Button** on the form.

Step 2: Set the **AutoPostBack** property of the **TextBox** as **true** so that this control will get the capability of **submitting** the **page to server** immediately when the **Text (Color)** is changed.

Understanding Page Submission: as learnt earlier the process of sending values that are entered by a user in controls of a page to the **Web Server** is known as **Page Submission** and it is of 2 types:

1. **Postback Submission**
2. **Cross Page Submission**

Postback Submission: To understand this in detail let’s create a **Login Form** and to do that add a new **WebForm** in the project naming it as “**LoginForm.aspx**” and write the below code under “**<div>**” tag:

```
<table align="center">
<caption>Login Form</caption>
<tr>
<td>Enter Name:</td>
<td><asp:TextBox ID="txtName" runat="server" /></td>
</tr>
<tr>
<td>Enter Password:</td>
<td><asp:TextBox ID="txtPwd" runat="server" TextMode="Password" MaxLength="16" /></td>
</tr>
<tr>
<td>Enter Email:</td>
```

```

<td><asp:TextBox ID="txtEmail" runat="server" TextMode="Email" /></td>
</tr>
<tr>
  <td colspan="2" align="center">
    <asp:Button ID="bntSubmit" runat="server" Text="Submit" />
    <asp:Button ID="btnReset" runat="server" Text="Reset" />
  </td>
</tr>
<tr>
  <td colspan="2">
    <asp:Label ID="lblStatus" runat="server" ForeColor="Red" />
  </td>
</tr>
</table>

```

Now go to **design view**, double click on the “Login Button” & “Reset Button” to generate required event handlers and write the below code in “**LoginPage.aspx.cs**” file:

Code under Page_Load:

```

if(!IsPostBack) {
  txtName.Focus();
}

```

Code under Login Button Click:

```

if (txtName.Text == "admin" && txtPwd.Text == "admin#123" && txtEmail.Text == "admin@nareshit.com") {
  lblStatus.Text = "Valid User";
}
else {
  lblStatus.Text = "Invalid User";
}

```

Code Under Reset Button Click:

```

txtName.Text = txtPwd.Text = txtEmail.Text = "";
txtName.Focus();

```

Note: in the above **Page** we have performed a **Postback Submission** i.e., the **Page** has submitted the data back to **itself** and was **validated**. In a **Postback Submission** we can read values of **controls** directly on server by using their associated **properties**, for example we have read the value of **TextBox** by using its **Text** property is our above code.

Cross Page Submission: this is a process of **submitting a page to another page** and to **understand Cross Page Submission** add 2 new **Web Forms** in the **project** naming them as “**Contact.aspx**” and “**Respond.aspx**”. Write the below code under “**<div>**” tag of “**Contact.aspx**”:

```

<table align="center">
  <caption>Contact Details</caption>
  <tr>

```

```

<td>Name:</td>
<td><asp:TextBox ID="txtName" runat="server" /></td>
</tr>
<tr>
<td>Phone No:</td>
<td><asp:TextBox ID="txtPhone" runat="server" MaxLength="10" /></td>
</tr>
<tr>
<td>Email Id:</td>
<td><asp:TextBox ID="txtEmail" runat="server" TextMode="Email" /></td>
</tr>
<tr>
<td colspan="2" align="center">
<asp:Button ID="btnSubmit" runat="server" Text="Submit" PostBackUrl="~/Respond.aspx" />
<asp:Button ID="btnReset" runat="server" Text="Reset" />
</td>
</tr>
</table>

```

Now go to design view, double click on the “Reset Button” to generate the event handler and write the below code in “Contact.aspx.cs” file:

Code under Page_Load:

```

if(!IsPostBack) {
    txtName.Focus();
}

```

Code under Reset Button:

```

txtName.Text = txtPhone.Text = txtEmail.Text = "";
txtName.Focus();

```

Run “Contact.aspx”, fill in the details and click on **Login** button, which will send a request from the browser to “Respond.aspx” because for **Submit Button** we have set “PostBackUrl” Property with the value “Respond.aspx” and we call this as “Cross Page Submission”.

In the process of “Cross Page Submission” all values that are entered into the **Controls** of “Contact.aspx” page will be submitted to “Respond.aspx” page, and at this time, lot of information is transferred from the browser to server like: **Session State, Application State, Request Cookie Collection, Response Cookie Collection, Header Collection, Form Collection, Query String Collection, Server Variable Collection**, etc.

Form Collection will contain the information of **controls** and their values in a “name/value” pair combination, so in “Respond.aspx” page we can read **Form Collection** values with the help of “name” using the request object and consume their values.

To test that go to “Respond.aspx.cs” file and write the below code under “Page_Load” Method:

```

string Name = Request.Form["txtName"];

```

```

string Phone = Request.Form["txtPhone"];
string Email = Request.Form["txtEmail"];
Response.Write("<h3>Hello " + Name + ", we have received your contact details.</h3>");
Response.Write("Contact Phone: " + Phone + "<br />");
Response.Write("Contact Email: " + Email + "<br />");
```

Note: in the above, “Request.Form” refers to “Form Collection” and the parameter we pass to it is the “name” corresponding to the value we want to read.

When data is submitted from 1 page to another page, data transfer from source page to target page will be done in any of the 2 methods: Get Method or Post Method.

For Html Form’s the default method is “get” and if we want to change it we require to use “method” attribute on the “<form>” element and specify it as “post” as following:

```
<form id="f1" name="f1" method="post">
```

For ASPX Form’s the default method is “post” and if we want to change it we require to use “method” attribute on the “<form>” element and specify it as “get” as following:

```
<form id="f1" runat="server" method="get">
```

Currently “Contact.aspx” is using an ASPX Form and so the submit method is “post” and if we want to change it as “get”, go to “Contact.aspx” file and add “method” attribute to “<form>” tag with value as “get” and it should look now as following:

```
<form id="form1" runat="server" method="get">
```

Run “Contact.aspx” Form again, enter values into the controls and click on “Submit” button which will Submit Control values to “Respond.aspx” but the difference is when the submit method is Post, values will go to target page as “Form Collection”, whereas when the Submit method is Get, values will go to target page as “Query String Collection”, so right now “Respond.aspx” page will not display any values that we entered in “Contact.aspx” page. To resolve the problem, go to “Respond.aspx.cs” file and change “Request.Form” as “Request.QueryString” in “Page_Load” method which should not look as below:

```

string Name = Request.QueryString["txtName"];
string Phone = Request.QueryString["txtPhone"];
string Email = Request.QueryString["txtEmail"];
```

Note: when we are not sure whether the submit method is “Get” or “Post” then we can directly read the values using Request object without specifying the Query String or Form as below:

```

string Name = Request["txtName"];
string Phone = Request["txtPhone"];
string Email = Request["txtEmail"];
```

Differences between Get and Post:

1. In case of Get the data that has to be transferred from browser to server will be concatenated to the URL of page we are requesting (target page) in the form of a “Query String” and we can view those values in the Address Bar of our browser, whereas in case of Post, data will be passed through “Http Headers” which is not at all visible to the end user.

Note: Http Headers let the client and server pass additional information with an Http Request or Http Response.

2. Get is faster in transfer when compared to post whereas post is secured compared to get.
3. Get is not recommended for transporting sensitive info like **password**, **credit card details**, etc. as the values are visible in address bar whereas post is **secured** method of transporting because, data is transferred through **HTTP Headers** by using secure **HTTP protocol**, so we can make sure that data is **secured**.
4. In case of Get we have limitations on the data being submitted i.e., the maximum length of a URL can be **2048** characters only whereas we don't have any size limitations for data in Post.
5. In case of get we access data from "**Query String Collection**" whereas in case of post we access data from "**Form Collection**" on the target pages.

How do we specify the URL of target page for a Cross Page Submission?

Ans: we specified the **URL** of target page for "**Cross Page Submission**" using the "**PostBackUrl**" property of **Button** whereas in other web technologies we specify that by using "**action**" attribute of form. We can use that in **ASP.NET** also and to test that delete the "**PostBackUrl**" property associated with "**Submit**" **Button**, then go to "**<form>**" tag and add **action** attribute to it, which should now look as following:

```
<form id="form1" runat="server" method="get" action="Respond.aspx">
```

Run the page again and watch, now also when we click on "**Submit**" button, page will be submitted to "**Respond.aspx**". The drawback in this approach is not only "**Submit**" button of our page, "**Reset**" button of our page also will do "**Page Submission**" and to avoid this problem, without using form's "**action**" attribute we use button's "**PostBackUrl**" attribute, so that "**Submit**" button performs "**Cross Page Submission**" and "**Reset**" button will perform "**Postback Submission**".

Transferring control from 1 page to another page: we can transfer control from 1 page to another page, in a **Web Application** and that can be performed in 2 different ways:

1. **Server.Transfer**
2. **Response.Redirect**

To check transferring of the control from 1 page to another page add 3 new **Web Forms** in the project naming them as: **LoginPage.aspx**, **SuccessPage.aspx** and **FailurePage.aspx**. Now go to **LoginPage.aspx**, define style for body tag and set a background-color to it: **<body style="background-color:lawngreen">** and then write the below code in "**<div>**" tag:

```
<table align="center">
<caption>Login Page</caption>
<tr>
<td>User Name:</td>
<td><asp:TextBox ID="txtName" runat="server" /></td>
</tr>
<tr>
<td>Password:</td>
<td><asp:TextBox ID="txtPwd" runat="server" TextMode="Password" /></td>
</tr>
<tr>
<td colspan="2" align="center">
```

```

<asp:Button ID="btnReset" runat="server" Text="Reset" />
<asp:Button ID="btnLogin" runat="server" Text="Login" />
</td>
</tr>
</table>

```

Now generate Click Event Handlers for Login and Reset Button, and write the below code in "LoginPage.aspx.cs" file:

Code under Page_Load:

```

if(!IsPostBack) {
    txtName.Focus();
}

```

Code under Reset Button:

```

txtName.Text = txtPwd.Text = "";
txtName.Focus();

```

Code under Login Button:

```

if(txtName.Text == "admin" && txtPwd.Text == "admin")
    Server.Transfer("SuccessPage.aspx");
else
    Response.Redirect("FailurePage.aspx");

```

Now go to "SuccessPage.aspx" and set "style" attribute for "<body>" tag which should look as below:

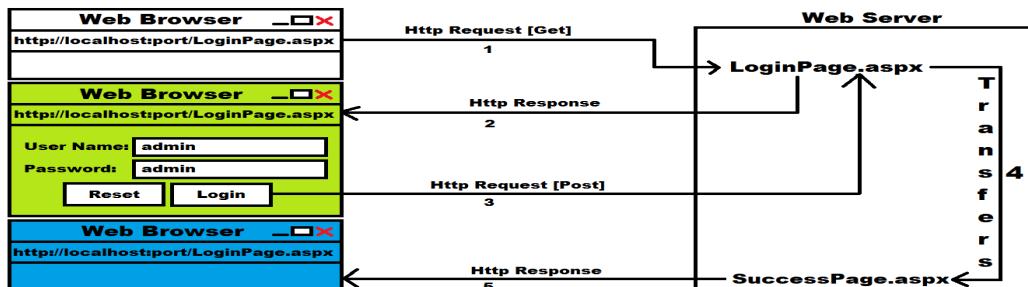
```
<body style="background-color:deepskyblue">
```

Now go to "FailurePage.aspx" and set "style" attribute for "<body>" tag which should look as below:

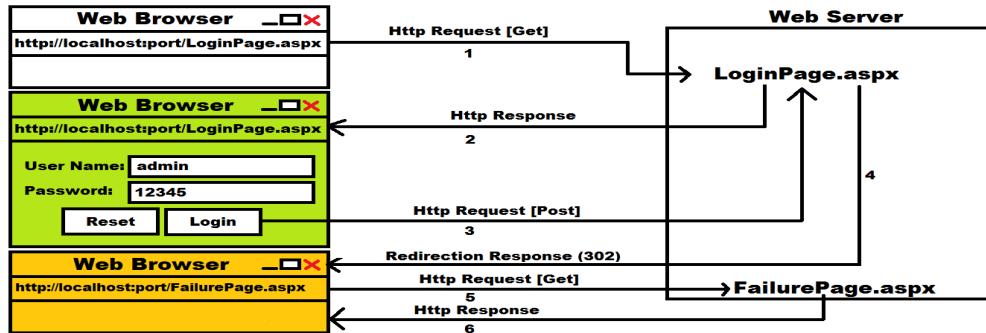
```
<body style="background-color:gold">
```

Now run "LoginPage.aspx" enter valid credentials and click on "Login" button which will take you to "SuccessPage.aspx" and when we enter any in-valid credentials and click on "Login" button it will take you to "FailurePage.aspx".

Server.Transfer: in this case the transfer between source and target pages will be performed within the server only i.e., when we enter valid credentials and click on the Login button first it goes to "LoginPage" because it is a "Postback" and from there "Server.Transfer" will transfer the control to SuccessPage as following:



Response.Redirect: in this case the transfer between source and target pages will be performed thru the browser i.e., when we enter **in-valid credentials** and click on the **Login** button first it goes to “**LoginPage**” because it is a “**PostBack**” and from there “**Response.Redirect**” will send a **re-direction (302) response** to the browser, asking the browser to make a new request to server for “**FailurePage**”, so browser will make a request to the server for “**FailurePage**” and receives the page as response, as following:



Differences between Server.Transfer and Response.Redirect methods:

1. In case of **Server.Transfer**, transfer between pages is performed directly on the **Web Server** it-self, whereas in case of **Response.Redirect**, transfer between pages is performed from the **browser**.
2. **Server.Transfer** is faster in execution because it will not have any extra round trips between the browser and server whereas in case of the second i.e., **Response.Redirect** there will be extra round trips between the browser and server, so execution is not faster.
3. In case of **Server.Transfer** the Url in browser’s Address bar will not be updated with new page Address, so we can’t **bookmark** the page whereas it happens in case of **Response.Redirect**, so we can **bookmark** the page.
4. In case of **Server.Transfer** we can transfer only to the pages of same site whereas in the case of **Response.Redirect** we can transfer to pages of same site as well as other sites on the same server as well as other servers also.

RadioButton and CheckBox Controls: We use these controls to provide the users with a list of values to choose from, so that users can select from those values. **Radio Buttons** are used in-case we want to provide an option for **single-selection** and **Checkbox’s** are used in-case we want to provide an option for **multi-selection**. Both **Radio Button** and **Check Box** Controls provides a **boolean** property called “**Checked**” using which we can recognize whether the control is selected or not, which returns **true** if **selected** and **false** if not **selected**.

Note: in-case of **Radio Button** control we need to explicitly tell from how many **Radio Button’s** 1 value should be selected and to do that we need to use “**Group Name**” attribute and group the **Radio Button’s** so that 1 **Radio Button** can be selected from the group.

To work with “**Check Box**” and “**Radio Button**” controls add a new **WebForm** in our project naming it as “**RadioAndCheck.aspx**” and write the below code under “**<div>**” tag:

Name:

```
<asp:TextBox ID="txtName" runat="server" /><br />
```

Gender:

```
<asp:RadioButton GroupName="Gender" ID="rbMale" runat="server" Text="Male" />
<asp:RadioButton GroupName="Gender" ID="rbFemale" runat="server" Text="Female" />
```

```

<asp:RadioButton GroupName="Gender" ID="rbTrans" runat="server" Text="Transgender" /><br />

Eating Habit:
<asp:RadioButton GroupName="Habit" ID="rbVeg" runat="server" Text="Vegetarian" />
<asp:RadioButton GroupName="Habit" ID="rbNonVeg" runat="server" Text="Non Vegetarian" />
<asp:RadioButton GroupName="Habit" ID="rbVegan" runat="server" Text="Vegan" /><br />
Hobbies:
<asp:CheckBox ID="cbReading" runat="server" Text="Reading Books" />
<asp:CheckBox ID="cbPlaying" runat="server" Text="Playing Games" />
<asp:CheckBox ID="cbWatching" runat="server" Text="Watching Movies" /><br />

<asp:Button ID="btnSubmit" runat="server" Text="Submit Values" /><br />
<asp:Label ID="lblMsg" runat="server" ForeColor="Red" />

```

Now generate a **Click Event Handler** form “Display Values” button, import the namespace “**System.Text**” and write the below code in “**RadioAndCheck.aspx.cs**” file:

```
using System.Text;
```

Code under Page Load Event:

```
if (!IsPostBack) {
    txtName.Focus();
}
```

Code under Display Button Click Event:

```
StringBuilder sb = new StringBuilder();

if (txtName.Text.Trim().Length > 0)
    sb.Append($"Name: {txtName.Text}<br />");
if (rbMale.Checked)
    sb.Append($"Gender: Male<br />");
else if (rbFemale.Checked)
    sb.Append($"Gender: Female<br />");
else if (rbTrans.Checked)
    sb.Append($"Gender: Transgender<br />");
if (rbVeg.Checked)
    sb.Append($"Eating Habit: Vegetarian<br />");
else if ((rbNonVeg.Checked))
    sb.Append($"Eating Habit: Non-Vegetarian<br />");
else if (rbVegan.Checked)
    sb.Append($"Eating Habit: Vegan<br />");

List<string> list = new List<string>();
if (cbPlaying.Checked)
    list.Add("Playing Games");
if (cbReading.Checked)
    list.Add("Reading Books");
if (cbWatching.Checked)
```

```

list.Add("Watching Movies");

if (list.Count > 0)
    sb.Append($"Hobbies: {String.Join(", ", list)}");

```

lblMsg.Text = sb.ToString();

List Controls: These controls are also used for providing the users with a list of values to choose from. We have 4 List Controls in ASP.NET Web Forms and those are:

1. DropDownList
2. ListBox
3. RadioButtonList
4. CheckBoxList

- **DropDownList:** default visible items are 1 and we can select only 1 item.
- **ListBox:** default visible items are multiple, and we can select 1 or more items.
- **RadioButtonList:** default visible items are multiple, but we can select only 1 item.
- **CheckBoxList:** default visible items are multiple, and we can select 1 or more items.

To work with these controls first we need to add values into them, and every value of a List Control is known as a **ListItem (class)** and each value is an instance of this class. Every ListItem is associated with 4 attributes:

- Text
- Value
- Enabled
- Selected

- **Text** refers to the “Display Member” of the control i.e., what user views.
- **Value** refers to the “Value Member” of the control i.e., this is not visible to end users but used for implementing the background logic.
- **Enabled** is a Boolean property with default value true and if set as false this ListItem is not visible to end users.
- **Selected** is a Boolean property with default value false and if set as true this ListItem is selected by default.

To work with List Controls, add a new Web Form in our project naming it as “ListControls1.aspx”, add 1 DropDownList, ListBox, RadioButtonList and CheckBoxList controls on the Form.

Adding List Item's into List Controls: to add List Item's into a List Controls we are provided with 5 different options, those are:

1. By using the **Items** property in properties window we can add List Item's into a List Control and to do that, in the design view of Web Form select “DropDownList”, go to its properties, select **Items** property and click on the button beside it which open “ListItem Collection Editor” window, click on the “Add Button” which will add a new ListItem in the LHS and now on the RHS it will display the 4 attributes where we can enter “Text & Value” for each ListItem. Follow this process and add 6 Countries into the Control with below Text & Value.

Text	Value
India	C1

Japan	C2
China	C3
France	C4
America	C5
Australia	C6

Note: This action will write all the necessary code in **Source** file and to view that code go to **Source View** and watch the “Inner Html” of “DropDownList” which will be as following:

```
<asp:ListItem Value="C1">India</asp:ListItem>
<asp:ListItem Value="C2">Japan</asp:ListItem>
<asp:ListItem Value="C3">China</asp:ListItem>
<asp:ListItem Value="C4">France</asp:ListItem>
<asp:ListItem Value="C5">America</asp:ListItem>
<asp:ListItem Value="C6">Australia</asp:ListItem>
```

2. Same as the above code what **VS** has generated for “**DropDownList**”, we can also manually write that code in Source View and to test that write the following code as “**Inner Html**” to “**ListBox**”:

```
<asp:ListItem Text="Kerala" Value="S1" />
<asp:ListItem Text="Tamilnadu" Value="S2" />
<asp:ListItem Text="Karnataka" Value="S3" />
<asp:ListItem Text="Telangana" Value="S4" />
<asp:ListItem Text="Maharastra" Value="S5" />
<asp:ListItem Text="Andhra Pradesh" Value="S6" />
```

3. We can also add Item’s to List Controls thru “**C# Code**” in Code View by calling “**Items.Add**” method on the “**ListControl**” and this method is an **overloaded** method which is provided with 2 overloads:

```
<ListControl>.Items.Add(string Item)
<ListControl>.Items.Add(ListItem Item)
```

Note: in case of the first overload, we are passing string as a parameter and internally it creates a “**ListItem**” and the “**Text & Value**” will be same i.e., the string what we passed over there only will be taken as “**Text & Value**” also, whereas in case of the second overload we need to explicitly create a “**ListItem**” with a “**Text & Value**”, and **add** it to the Control. To test the above go to “**ListControl1.aspx.cs**” file and write the below code in “**Page_Load**” Event Handler:

```
RadioButtonList1.Items.Add("Delhi");
RadioButtonList1.Items.Add("Kolkata");
RadioButtonList1.Items.Add("Mumbai");
RadioButtonList1.Items.Add(new ListItem("Chennai", "City4"));
RadioButtonList1.Items.Add(new ListItem("Bengaluru", "City5"));
RadioButtonList1.Items.Add(new ListItem("Hyderabad", "City6"));
```

4. Same as the above we have another method “**Items.AddRange**” and by using this we can add an **Array of List Item’s** at a time into the **control**.

```
<ListControl>.Items.AddRange(ListItem[] Items)
```

To test the above go to “ListControls1.aspx.cs” file and write the below code in “Page_Load” Event Handler:

```
ListItem color1 = new ListItem("Red", "Color1");
ListItem color2 = new ListItem("Blue", "Color2");
ListItem color3 = new ListItem("White", "Color3");
ListItem color4 = new ListItem("Green", "Color4");
ListItem color5 = new ListItem("Purple", "Color5");
ListItem color6 = new ListItem("Magenta", "Color6");
ListItem[] colors = new ListItem[] { color1, color2, color3, color4, color5, color6 };
CheckBoxList1.Items.AddRange(colors);
```

Or

```
ListItem[] colors = new ListItem[]
{
    new ListItem("Red", "Color1"),
    new ListItem("Blue", "Color2"),
    new ListItem("White", "Color3"),
    new ListItem("Green", "Color4"),
    new ListItem("Purple", "Color5"),
    new ListItem("Magenta", "Color6")
};
```

```
CheckBoxList1.Items.AddRange(colors);
```

Or

```
CheckBoxList1.Items.AddRange(new ListItem[] {
    new ListItem("Red", "Color1"),
    new ListItem("Blue", "Color2"),
    new ListItem("White", "Color3"),
    new ListItem("Green", "Color4"),
    new ListItem("Purple", "Color5"),
    new ListItem("Magenta", "Color6")
});
```

5. By using “DataSource” property of **List Control’s** we can assign values from any **DataSource** like a **File** or **Database** also.

```
<ListControl>.DataSource = <Table>;
<ListControl>.DataTextField = <Column_Name>;
<ListControl>.DataValueField = <Column_Name>;
<ListControl>..DataBind();
```

Behavior of each List Control:

1. **DropDownList:** This control is used in-case we want to provide the option for **single selection**.
2. **ListBox:** This control by default supports **single selection only** whereas if we want **multi-selection** then we need to set **“SelectionMode”** property of the control as **“Multiple”** because, default is **“Single”**.
3. **RadioButtonList:** This control is also like **DropDownList** control, providing support for **single selection only** whereas in case of **RadioButtonList** we find a **Radio Button** beside each item for selection.
4. **CheckBoxList:** this control works same as a **RadioButtonList**, but the only difference is it supports **Multi-Selection** with a **CheckBox** beside each item for selection.

Note: RadioButtonList and CheckBoxList provides a property “RepeatDirection” and by using it we can specify the direction in which “List Items” are laid out, default is “Vertical” direction and can be changed to “Horizontal”. By using the “RepeatLayout” property we can set layout for “List Items” which is by default “Table” but can be changed to “Flow”, “OrderedList” and “UnOrderedList” (OrderedList and UnOrderList layout will not work in case the “RepeatDirection” is set as Horizontal i.e., will work only in case of Vertical). By using “RepeatColumns” property we can specify the number of columns to layout the “List Items”.

Accessing values from ListControls: to access values from ListControls we are provided with a set of Members under them as following:

1. **Items:** by using this property we can access all the Items of a ListControl which returns them in the form of ListItem Array.

<ListControl>.Items => ListItem[]

2. **SelectedItem:** by using this property we can access the “SelectedItem” form the ListControl which will return the value as a ListItem.

<ListControl>.SelectedItem => ListItem (Text and Value)

3. **SelectedValue:** this property returns the “Value” that is associated with a selected “ListItem” in the form of a string.

<ListControl>.SelectedValue => String (Only Value)

4. **Text:** this property is also same as “SelectedValue” only which returns the “Value” that is associated with a selected ListItem in the form of a string.

<ListControl>.Text => String (Only Value)

5. **SelectedIndex:** this property returns the index position of the selected “ListItem” in the form of int.

<ListControl>.SelectedIndex => int

6. **GetSelectedIndices():** this is a method present under “ListBox” control that returns an array of indices corresponding to the SelectedItems.

<ListBox>.GetSelectedIndices => int[]

Note: the first 5 members are available under all the controls whereas the last member i.e., “GetSelectedIndices” is specific to “ListBox” control.

XML: Extensible Markup Language

- This is also a markup language like HTML but used for describing the content of a Text Document.
- This is used in the industry for transporting the data from 1 machine to another machine apart from Excel and CSV (Comma Separated Values).
- This can also be used as a Temporary Database in scenarios where we don’t have access to a Live Database.
- XML is Platform Independent which can be used in any O.S. and well as every application provides options to read data from XML.
- Just like HTML, XML data is also present under Tags, but the difference is HTML Tags are pre-defined whereas XML Tags are user-defined.
- An XML document should be saved with “.xml” extension.

- An XML document can have **1 & only 1** root element.
- A simple XML documents looks as following:

```
<Employees>
  <Employee SerialNo="1">
    <Id>1001</Id>
    <Name>Scott</Name>
    <Job>Manager</Job>
    <Salary>50000</Salary>
  </Employee>
  <Employee SerialNo="2">
    <Id>1002</Id>
    <Name>Smith</Name>
    <Job>Salesman</Job>
    <Salary>15000</Salary>
  </Employee>
  <!--Enter multiple employees data-->
</Employees>
```

- XML is case sensitive, so we need to strictly follow the same case for starting and ending tag.

<Id>101</Id> => Valid
<Id>101</ID> => Invalid

- To understand XML, we need a software known as **XML Parser** and this is built into every browser.
- XML tags also can have **attributes**, for example we have defined “SerialNo” as an attribute for “Employee” tag, but values to those attributes must be enclosed in double quotes (**Mandatory**).

Note: our “ASPx” code in ASP.NET pages, completely follows XML syntax’s only.

Loading Data into List Control’s from an XML File: to load data into List Controls from XML File do the below.

Step 1: Add a WebForm naming it is as “**ListControls2.aspx**” and write the following code under its “**<div>**” tag:

```
<table align="center" border="1">
  <caption>Product Catalog</caption>
  <tr>
    <td align="center"><asp:DropDownList ID="DropDownList1" runat="server" /></td>
    <td align="center"><asp:ListBox ID="ListBox1" runat="server" SelectionMode="Multiple" /></td>
    <td align="center"><asp:RadioButtonList ID="RadioButtonList1" runat="server" /></td>
    <td align="center"><asp:CheckBoxList ID="CheckBoxList1" runat="server" /></td>
  </tr>
  <tr>
    <td align="center"><asp:Button ID="Button1" runat="server" Text="Show Selected Product" /></td>
    <td align="center"><asp:Button ID="Button2" runat="server" Text="Show Selected Products" /></td>
    <td align="center"><asp:Button ID="Button3" runat="server" Text="Show Selected Product" /></td>
    <td align="center"><asp:Button ID="Button4" runat="server" Text="Show Selected Products" /></td>
```

```

</tr>
<tr>
    <td><asp:Label ID="Label1" runat="server" ForeColor="Red" /></td>
    <td><asp:Label ID="Label2" runat="server" ForeColor="Red" /></td>
    <td><asp:Label ID="Label3" runat="server" ForeColor="Red" /></td>
    <td><asp:Label ID="Label4" runat="server" ForeColor="Red" /></td>
</tr>
</table>

```

Step 2: In “Add New Item” window, choose XML File item, name it as “Products.xml” and write the below code.

```

<Products>
    <Product>
        <Id>101</Id>
        <Name>Television</Name>
    </Product>
    <Product>
        <Id>102</Id>
        <Name>Microwave</Name>
    </Product>
    <Product>
        <Id>103</Id>
        <Name>Washing Machine</Name>
    </Product>
    <Product>
        <Id>104</Id>
        <Name>Refrigerator</Name>
    </Product>
    <Product>
        <Id>105</Id>
        <Name>Home Theatre</Name>
    </Product>
    <Product>
        <Id>106</Id>
        <Name>Mixer Grinder</Name>
    </Product>
    <Product>
        <Id>107</Id>
        <Name>Blu-ray Player</Name>
    </Product>
    <Product>
        <Id>108</Id>
        <Name>Split A/C</Name>
    </Product>
    <Product>
        <Id>109</Id>

```

```

<Name>Air Cooler</Name>
</Product>
<Product>
<Id>110</Id>
<Name>Window A/C</Name>
</Product>
</Products>

```

Step 3: Now go to design view of the **Web Form**, generate **Event Handlers** for all the **4 Buttons** and write the below code in “**ListControls2.aspx.cs**” file:

```

using System.Data;
Code under Page Load Event:
if (!IsPostBack)
{
    string xmlFilePath = Server.MapPath("~/Products.xml");
    DataSet ds = new DataSet();
    ds.ReadXml(xmlFilePath);

    DropDownList1.DataSource = ds;
    DropDownList1.DataTextField = "Name";
    DropDownList1.DataValueField = "Id";
    DropDownList1.DataBind();
    DropDownList1.Items.Insert(0, "-Select Product-");

    ListBox1.DataSource = ds;
    ListBox1.DataTextField = "Name";
    ListBox1.DataValueField = "Id";
    ListBox1.DataBind();

    RadioButtonList1.DataSource = ds;
    RadioButtonList1.DataTextField = "Name";
    RadioButtonList1.DataValueField = "Id";
    RadioButtonList1.DataBind();

    CheckBoxList1.DataSource = ds;
    CheckBoxList1.DataTextField = "Name";
    CheckBoxList1.DataValueField = "Id";
    CheckBoxList1.DataBind();
}

```

Code under Button1 Click Event:

```

if (DropDownList1.SelectedIndex > 0) {
    ListItem li = DropDownList1.SelectedItem;
    Label1.Text = li.Value + ":" + li.Text;
}

```

```
else {
    Label1.Text = "";
}
```

Code under Button2 Click Event:

```
Label2.Text = "";
foreach(int index in ListBox1.GetSelectedIndices()) {
    ListItem li = ListBox1.Items[index];
    Label2.Text += li.Value + ":" + li.Text + "<br />";
}
```

Code under Button3 Click Event:

```
if (RadioButtonList1.SelectedItem != null) {
    ListItem li = RadioButtonList1.SelectedItem;
    Label3.Text = li.Value + ":" + li.Text;
}
```

Code under Button4 Click Event:

```
Label4.Text = "";
foreach(ListItem li in CheckBoxList1.Items) {
    if(li.Selected) {
        Label4.Text += li.Value + ":" + li.Text + "<br />";
    }
}
```

DataSet is a class, present under “**System.Data**” namespace which can read the data from an **XML File** or **Database** and hold it in a **Table** format.

MapPath is a method of “**HttpServerUtility**” class, and this method returns the **Physical Path** of a file when we pass **Virtual Path** of the file.

FileUpload Control

This control provides an interface to the end user for selecting a file to upload. Once the file is selected by the user, we can call members of “**HttpPostedFile**” class to upload the selected file to **Web Server**. To work with “**FileUpload**” control add a new **Web Form** in our project naming it as “**FileDemo1.aspx**” and write the below code under its “**<div>**” tag:

```
<asp:FileUpload ID="FileUpload1" runat="server" />
<br />
<asp:Button ID="btnUpload" runat="server" Text="Upload File" />
<br />
<asp:Label ID="lblMsg" runat="server" ForeColor="Red" />
```

Now go to “**FileDemo1.aspx.cs**” file and write the below code under “**Upload File**” button **Click Event Handler** by import “**System.IO**” namespace:

```
string folderPath = Server.MapPath("~/Uploads/");
//Cheching whether the Folder we want is existing or not, so that we can create a new folder if not exist's
```

```

if (!Directory.Exists(folderPath))
{
    //Code for creating a new folder under the project if it does not exist:
    Directory.CreateDirectory(folderPath);
}

//Code for uploading and saving the file selected by user in FileUpload control:
HttpPostedFile selectedFile = FileUpload1.PostedFile;
selectedFile.SaveAs(folderPath + selectedFile.FileName);
lblMsg.Text = selectedFile.FileName + " uploaded to the server.";

```

“**Directory**” is a class under “**System.IO**” namespace and “**Exists**” is a static method under the class which determines whether the given path refers to an existing folder or file on the hard disk and returns a **Boolean** value. “**CreateDirectory**” is also a static method of the “**Directory**” class which will create a directory at the specified path.

“**PostedFile**” is a property of “**FileUpload**” class which will return the “**HttpPostedFile**” class’s object for the file that is selected under “**FileUpload**” control. “**HttpPostedFile**” class provides the complete info of the file that is selected under “**FileUpload**” control like, “**FileName**”, “**ContentLength**”, “**ContentType**” etc, and provides a method known “**SaveAs**” to save the file at the desired location.

Uploading multiple files using FileUpload control: By default, “**FileUpload**” control provides an option to select only a single file for uploading, whereas if we want to provide an option for multi-selection then we need to set the “**AllowMultiple**” attribute of “**FileUpload**” control as **true** because by default it is **false**.

Add a new Web Form in the project naming it as “FileDemo2.aspx” and write the below code in its “<div>” tag:

```

<asp:FileUpload ID="FileUpload1" runat="server" AllowMultiple="true" />
<br /><br />
<asp:Button ID="btnUpload" runat="server" Text="Upload Files" />
<br />
<asp:Label ID="lblMsg" runat="server" ForeColor="Red" />

```

Now go to “**FileDemo2.aspx.cs**” file and write the below code under “**Upload Files**” button **Click Event Handler** by importing “**System.IO**” namespace:

```

string folderPath = Server.MapPath("~/Uploads/");
if (!Directory.Exists(folderPath)) {
    Directory.CreateDirectory(folderPath);
}
if (FileUpload1.HasFiles) {
    int Count = 0;
    foreach (HttpPostedFile selectedFile in FileUpload1.PostedFiles) {
        Count += 1;
        selectedFile.SaveAs(folderPath + selectedFile.FileName);
    }
    lblMsg.Text = Count + " file(s) uploaded to the server.";
}

```

```

    }
else {
    lblMsg.Text = "Please select a file(s) for uploading.";
}

```

“**PostedFiles**” is a property of “**FileUpload**” class which returns a List of “**HttpPostedFile**” class objects representing each file that is selected under “**FileUpload**” control.

Calendar Control

This control is used for providing the users an option for selecting a **Date**. **Calendar** control will render an **HTML Table**. This control is provided with the following set of members to work with it:

1. **SelectedDate:** this property can be used for setting or getting the **Date** selected under the control and the type of this property is **DateTime**.
2. **SelectedDates:** this property can be used for getting the **List of Dates** that are selected under the control in the form of an Array.
3. **Caption:** this property is used for setting a **Caption** to the Calendar control.
4. **DayNameFormat:** this property is to specify format for day “**Header Text**” which can be set with any of the following values: **Full**, **Short [d]**, **FirstLetter**, **FirstTwoLetters** or **Shortest**.
5. **FirstDayOfWeek:** this property is to specify which day of the week has to be displayed first in the **Calendar**; default is **Sunday** and can be changed to any day of the week.
6. **NextPrevFormat:** this property is to specify format for month **navigation** buttons and this property can be set with any of the following values: **ShortMonth**, **FullMonth** & **CustomText [d]**. If we are using **CustomText** then the text (caption) for “**Next Month Button**” and “**Previous Month Button**” can be specified by using the properties “**NextMonthText**”, who’s default value is: **>**” and “**PreviousMonthText**”, who’s default value is: **<**”.
7. **SelectionMode:** this property determines whether the selection should be **Day [d]** or **DayWeek** or **DayWeekMonth** or **None**.

Note: when the “**SelectionMode**” is set as “**DayWeek**” or “**DayWeekMonth**” then we can specify the text that has to be displayed for select week button and select month button by using the properties “**SelectWeekText**”, who’s default value is: **>**” and “**SelectMonthText**”, who’s default value is: **>>**”.

8. **SelectionChanged:** this is an event which fires when the **Date** selection is changed, and this is the default event of **Calendar**.
9. **DayRender:** this is also an event which fires for each day being **rendered** by the **Calendar** i.e., if we launch the **Calendar** for 1 time this event will fire for **42** times.
10. **VisibleMonthChanged:** this is also an **event** which fires when we navigate to **Next** or **Previous** months by clicking on “**Next Month Button**” and “**Previous Month Button**”.

To work with a **Calendar**, first download any **Calendar** image on to your computer and add to the **Icons** folder in our project. Now add a new WebForm in the project naming it as “**CalendarDemo.aspx**” and write the below code in its “**<div>**” tag:

```

<asp:ScriptManager ID="ScriptManager1" runat="server" />
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        Enter Date:

```

```

<asp:TextBox ID="txtDate" runat="server" />
<asp:ImageButton ID="imgDate" runat="server" ImageUrl("~/Icons/Calendar.ico" ImageAlign="AbsMiddle"
Width="20" Height="20" />
<asp:Calendar ID="cldDate" runat="server" Visible="false" />
</ContentTemplate>
</asp:UpdatePanel>

```

Now go to design view of the **Web Form** and then generate a “**Click**” Event Handler for “**ImageButton**” and “**SelectionChanged**” Event Handler for “**Calendar**” control and write the below code under “**CalendarDemo.aspx.cs**” file:

Code under Page Load Event:

```

if (!IsPostBack) {
    txtDate.Focus();
}

```

Code under Image Button Click Event:

```

if (cldDate.Visible == false) {
    cldDate.Visible = true;
}
else {
    cldDate.Visible = false;
}

```

Code under Calendar SelectionChanged Event:

```

txtDate.Text = cldDate.SelectedDate.ToShortDateString();
cldDate.Visible = false;

```

ASP.NET Web Configuration File

The Microsoft .NET Framework, and **ASP.NET**, uses **XML-formatted “.config” files** to configure applications. The **.NET Framework** relies on “**.config**” files to define configuration options. The **.config** files are text-based **XML** files. Multiple “**.config**” files can, and typically do, exist on a single system. The behavior of an **ASP.NET Application** is affected by different settings in the configuration files like “**machine.config**” and “**web.config**”.

Machine.config: System-wide configuration settings for **.NET Framework** are defined in the **Machine.config** file. The **Machine.config** file is in the **%SystemRoot%\Microsoft.NET\Framework\%VersionNumber%\CONFIG** folder. The default settings that are contained in the **Machine.config** file can be modified to affect the behavior of **Microsoft .NET Applications** on the whole system.

Web.config: You can change the **ASP.NET** configuration settings for a single application if you create a **Web.config** file in the root folder of the application. When you do this, the settings in the **Web.config** file override the settings in the **Machine.config** file. The **Web.config** file must contain only entries for configuration items that override the settings in the **Machine.config** file. At a minimum, the **Web.config** file must have the **<configuration>** element and the **<system.web>** element. These elements will contain individual configuration elements as following:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration><system.web></system.web></configuration>

```

The first line of the **Web.config** file describes the document as **XML-formatted** and specifies the character encoding type. This first line must be the same for all **".config"** files. The lines that follow mark the beginning and the end of the **<configuration>** element and the **<system.web>** element of the **Web.config** file. By themselves, these lines do nothing. However, the lines provide a structure that permits you to add future configuration settings. You add the majority of the **ASP.NET Configuration Settings** between the **<system.web>** and **</system.web>** lines.

The **ASP.NET** configuration system features an extensible infrastructure that enables you to define configuration settings at the time your **ASP.NET** applications are first deployed so that you can add or revise configuration settings at any time with minimal impact on operational **Web Applications** and **Servers**.

The ASP.NET configuration system provides the following benefits:

- Configuration information is stored in XML-based text files. You can use any standard text editor or XML parser to create and edit ASP.NET configuration files.
- Multiple configuration files, all named **Web.config**, can appear in multiple directories on an ASP.NET Web application server. Each **Web.config** file applies configuration settings to its own directory and all child directories below it. Configuration files in child directories can supply configuration information in addition to that inherited from parent directories, and the child directory configuration settings can override or modify settings defined in parent directories. The root configuration file named **Machine.config** provides ASP.NET configuration settings for the entire Web server.
- At run time, ASP.NET uses the configuration information provided by the **Web.config** files in a hierarchical virtual directory structure to compute a collection of configuration settings for each unique URL resource. The resulting configuration settings are then cached for all subsequent requests to a resource. Note that inheritance is defined by the incoming request path (the URL), not the file system paths to the resources on disk (the physical paths).
- ASP.NET detects changes to configuration files and automatically applies new configuration settings to Web resources affected by the changes. The server does not have to be rebooted for the changes to take effect. Hierarchical configuration settings are automatically recalculated and recached whenever a configuration file in the hierarchy is changed.
- The **ASP.NET** configuration system is extensible. You can define new configuration parameters and write configuration section handlers to process them.
- ASP.NET helps protect configuration files from outside access by configuring Internet Information Services (IIS) to prevent direct browser access to configuration files. HTTP access error 403 (forbidden) is returned to any browser attempting to request a configuration file directly.

Various Sections in Web.config file:

<system.web>: this element contains information about how the **ASP.NET** hosting layer manages application behavior.

<system.codedom>: this element contains compiler configuration settings for language providers installed on a computer in addition to the default providers that are installed with the **.NET Framework**, such as the **CSharpCodeProvider** and the **VBCodeProvider**.

<system.net>: this element contains settings that specify how the **.NET Framework** connects to the network i.e., it contains network connection details.

<appSettings>: this element stores custom application configuration information, such as file paths, XML Web service URLs, or any other custom configuration information for an application. Information is stored in the <appSettings> as key/value pairs and those elements are accessed in code using the ConfigurationSettings class.

<connectionStrings>: this element contains initialization information that is passed as a parameter from an application to a data source. Information is stored in the <connectionStrings> as key/value pairs and those elements are accessed in code using the ConfigurationSettings class.

Validation Controls

Validation is an important part of any **Web Application**. Whenever the user gives input in a **Web Page**, it must always be validated before sending it across various layers of an application. If we get the user input without any validation, then chances are that data might be wrong. So, **validation** is a good idea to do whenever we are taking input from the user. In **Web Application** validations are performed in 2 different levels:

1. Client-Side Validation
2. Server-Side Validation

Client-Side Validation: When a validation is done on the client's browser, then it is known as **Client-Side** validation. We use Java Script to do **Client-Side** validations. Since many years, developers are using Java Script for **Client-Side** validations. **Client-side** validation is very good, but we must be dependent on browser and scripting language support. Client-side validation is considered convenient for users as they get instant feedback. The main advantage is that it prevents a page from being **submitted** to the server until the client validation is executed successfully.

Server-Side Validation: When a validation occurs on the server, then it is known as **Server-Side** validation. **Server-Side** validation is a secure form of validation. The main advantage of **Server-Side** validation is that if the user somehow bypasses the **Client-Side** validation by disabling "**Java Script**" on his browser, we can still catch the problem on **Server**. **Server-Side** provides more security and ensures that no invalid data is processed by the application. **Server-Side** validation is done by writing the custom logic for validating the input, and developer point of view **Server-Side** is preferable because it will not fail, it is not dependent on browser and scripting languages.

Validation Controls in ASP.NET: To simplify validations in **ASP.NET** we are provided with **validation controls** and these controls will perform data validations on the "**Client Side**" as well as on the "**Server Side**" i.e., first they will perform **client-side** validations by using **Java Script**, and if at all validations fail it will stop the page being submitted to the server, and in-case if the client disables **Java Script** on his browser then page will be submitted to server even when the validations fail and in such scenarios **Validation Controls** will **re-validate** data on the **server-side** again. So, at any cost **server-side** validation will work, whether **client-side** validation is executed or not, so we have safety of validation check.

An important aspect of creating **ASP.NET Web Pages** for user input is to be able to check that the information users enter is **valid**. **ASP.NET** provides a set of controls known as **Validation Controls** that provide an **easy-to-use** and powerful way to check for **errors** and, if necessary, **display messages** to the user. The following is the list of **Validation Controls** we have in **ASP.NET**:

- ② RequiredFieldValidator
- ② CompareValidator
- ② RangeValidator
- ② RegularExpressionValidator
- ② CustomValidator

ValidationSummary

Validation Controls are classes which are inherited from the class “**BaseValidator**” hence they inherit its properties, events, and methods. Therefore, it would help us to look at the properties and the methods of that base class, which are common for all the validation controls:

1. **ControlToValidate**: by using this **property** we need to specify the “**Id**” of **Input Control** which must be **validated** by the **Validation Control**.
2. **Display**: this is an **Enumerated Property** using which we can set how the “**Error Message**” is shown on the browser and it can be set with any of the following values like: **Static[d]**, **Dynamic** and **None**.
3. **EnableClientScript**: this is a **Boolean property** which indicates whether **client-side validation** should take place or not. The **default value** is **true** and if set as **false** then **client-side validations** will not take place.
4. **Enabled**: this is a **Boolean property** with default value **true** and if set as **false** then **validator** gets **disabled** i.e., it will not **validate** the **input control**.
5. **ErrorMessage**: this **property** is used to specify an **Error Message** that must be **displayed** when the **validation fails**, and this message is also copied by the “**Validation Summary**” control.
6. **Text**: this is also like “**Error Message**” **property** only, but when both the **properties** are set with a **value**, **first preference** is given to “**Text**” only which is displayed at the place where validation control is placed whereas **Error Message** values will be used by the “**Validation Summary**” control.
7. **SetFocusOnError**: this is a **Boolean property** with default value **false** and when set as **true** will set the **focus** back in to the **Input Control** which has been validated by the **Validation Control** if the **validation fails**.
8. **ValidationGroup**: this **property** is used to **group** a **set of controls** on the **page**, so that **Validations** in 1 **group** will not affect **validations** of another **group** while submitting the **Page**.
9. **IsValid**: this is a **Boolean property** which indicates whether the **value** of the **input control** is **valid** or not.
10. **ForeColor**: this **property** is to specify the **color** in which **Error Messages** should be **displayed**.

ASP.NET Web Forms have provided **Validation Controls** since the initial releases. Prior to **ASP.NET 4.5** in a normal **validation** scenario, when we use a **validator** to **validate** any **control** and use **client-side validation**, some **Java Script** is generated and **rendered** as **HTML**. Also, some supportive **Java Script** files are also get loaded by the **browser** for the **validation**. **Unobtrusive Java Script** is the way of writing **Java Script** language in which we properly separate **Document Content** and **Script Content** thus allowing us to make a clear **distinction** between them. This approach is useful in so many ways as it makes our code **less error prone**, **easy to update** and **debug**.

From ASP.NET 4.5 we need to explicitly specify how **ASP.NET** globally enables the built-in validator controls to use **Unobtrusive Java Script** for **client-side validation logic** and we can set it with any of the 2 values like “**None**” or “**Web Forms**”. If this key’s value is set to “**None**”, the **ASP.NET** application will use the pre - 4.5 behavior (**Java Script inline in the pages**) for **client-side validation logic**. If this key’s value is set to “**Web Forms**”, **ASP.NET** uses **HTML 5 data-attributes** and late bound **Java Script** from an added script reference for client-side validation logic. We can set the unobtrusive validation mode in **Global.asax** file or in the individual **Web Form** or **Web.Config**.

Option 1: To set **Unobtrusive Validation Mode** in **Global.asax** file we need to set the **UnobtrusiveValidationMode** property of **ValidationSettings** class in the **Application_Start** Event Handler by importing the **namespace** “**System.Web.UI**” as following:

```
ValidationSettings.UnobtrusiveValidationMode = UnobtrusiveValidationMode.None;
```

Option 2: To set **Unobtrusive Validation Mode** in a Web Form we need to set the **UnobtrusiveValidationMode** property of **ValidationSettings** class in **Page_Load** Event Handler of each **Web Form** as following:

```
this.UnobtrusiveValidationMode = UnobtrusiveValidationMode.None;
```

Option 3: To set unobtrusive validation mode in **Web.config** file we need to add a new key under **<appSettings>** tag with the name “**ValidationSettings:UnobtrusiveValidationMode**” and value as “**none**” as below. If **<appSettings>** tag is present, then directly add the following:

```
<add key="ValidationSettings:UnobtrusiveValidationMode" value="none" />
```

If <appSettings> tag is not present, then write it as following:

```
<appSettings>
  <add key="ValidationSettings:UnobtrusiveValidationMode" value="none" />
</appSettings>
```

RequiredFieldValidator: This **validation control** is used to check whether the **input control** is left **empty or not** i.e., it should either be **entered** with a value or **selected** with a value. This control has a **property** which is specific to itself:

1. **InitialValue:** this property is to specify a **value**, which the **Validation Control** should not take into **consideration** while validating the **Input Control**.

Add a WebForm in the project naming it as “Validations1.aspx” and write the below code under its <div> tag:

```
Enter Name: <asp:TextBox ID="txtName" runat="server" />
<asp:RequiredFieldValidator ID="rfvName" runat="server" ControlToValidate="txtName" ForeColor="Red"
  ErrorMessage="Can't leave the field empty." Display="Dynamic" SetFocusOnError="true" />
<br />
Select Country:
<asp:DropDownList ID="ddlCountries" runat="server">
  <asp:ListItem Text="-Select Country-" Value="NS" />
  <asp:ListItem Text="India" Value="Country1" />
  <asp:ListItem Text="Japan" Value="Country2" />
  <asp:ListItem Text="China" Value="Country3" />
  <asp:ListItem Text="England" Value="Country4" />
  <asp:ListItem Text="America" Value="Country5" />
  <asp:ListItem Text="Australia" Value="Country6" />
</asp:DropDownList>
<asp:RequiredFieldValidator ID="rfvCountry" runat="server" ControlToValidate="ddlCountries" ForeColor="Red"
  ErrorMessage="Please select a country." InitialValue="NS" Display="Dynamic" SetFocusOnError="true" />
<br /><asp:Button ID="btnSubmit" runat="server" Text="Submit Data" />
<br /><asp:Label ID="lblMsg" runat="server" />
```

Now goto “aspx.cs” file and write the below code in it:

Code under Page_Load Event Handler:

```
if (!IsPostBack) {
  txtName.Focus();
}
```

Code under Submit Page Button Click Event Handler:

```
lblMsg.Text = $"Hello {txtName.Text}, the country you belongs to is: {ddlCountries.SelectedItem.Text}";
```

Now run the **Web Form** and watch the **output**, where we notice when the 2 **input controls** are not filled with a value and **Submit Button** is clicked, page will not be submitted to **Server** whereas when we fill values and click on **Submit Button** then the page will be submitted to **Server** and displays message under the **Label** control.

When we implement data validations logic by using **Java Script** we will face a problem i.e., if **Java Script** is **disabled** by client on his **browser**, then page will be submitted to server even when the validations fail and executes the logic on server even if the logic is dependent on the validations. To test this disable **Java Script** on your browser and test the page again so that now even if we did not fill the values in **input controls** and click on the **Submit Button** will submit page to **Server** and displays the previous message in **Label** control.

Whereas the advantage with our **Validation Controls** is even if **Java Script** is disabled and page is submitted to server, still validation controls will **re-validate** input controls on the server and sets the Validation Controls **"IsValid"** property with a **Boolean** value which is **true** when the validation is **success** and **false** if the validation **fails**. So we can implement logic under the server by checking the condition as following:

```
if (rfvName.IsValid && rfvCountry.IsValid)
{
    lblMsg.ForeColor = System.Drawing.Color.Green;
    lblMsg.Text = "Hello user the name you have entered is: " + txtName.Text + " and your selected country is: " +
                  ddlCountries.SelectedItem.Text;
}
else
{
    lblMsg.ForeColor = System.Drawing.Color.Red;
    lblMsg.Text = "Data validations failed.";
}
```

Note: the problem we face with above code is when we have multiple validations controls, checking each Validation Controls **"IsValid"** value will become complicated and we can avoid this by using Page Classes **"IsValid"** property because internally each validation control will first perform data validations and sets their **"IsValid"** property value as true or false based on success or failure of the validations and finally sets the Page Class **"IsValid"** property as true if all the Validation Controls **"IsValid"** value is **true**, whereas if any Validation Controls **"IsValid"** property is false then Page Class **"IsValid"** property value also will be false. We can simplify the **"if condition"** in our previous code as following:

```
if (rfvName.IsValid && rfvCountry.IsValid) => if (isValid)
```

Currently **validations** are getting performed when we click on the **Submit Button**, whereas if we want the validations to be performed when the **focus** is **leaving** the control add the following code in **"Page_Load"** Event **Handler** of the if condition:

```
txtName.Attributes.Add("onblur", "ValidatorValidate(rfvName)");
ddlCountries.Attributes.Add("onblur", "ValidatorValidate(rfvCountry)");
```

The above statements will bind the “Validation Controls”, “Java Script” function to the controls “onblur” event so that Event get fired when the focus is leaving the controls.

RangeValidator: this validation control is used to check whether the value entered into an input control is within a specified range or not. This control is associated with these properties in specific:

1. **MinimumValue:** this is to specify the least value that can be entered into the input control being validated.
2. **MaximumValue:** this is to specify the highest value that can be entered into the input control being validated.
3. **Type:** this is to specify the Data Type for the values which are being compared and can be set with any of the following values: String [d], Integer, Double, Date and Currency.

Note: none of the Validation Controls except RequiredFieldValidator can check for empty values, so while using other Validation Controls if we want to perform an empty value check we need to use a RequiredFieldValidator control also along with other validation controls. 1 validation control can be associated with 1 and only 1 input control whereas 1 input control can be associated with any no. of validation controls.

Add a new WebForm in our project naming it as “Validations2.aspx” and design it as following:

Journey Planner																																																							
Enter Name:	<input id="txtName" type="text"/>			<requiredfieldvalidator1></requiredfieldvalidator1>																																																			
Enter Age:	<input id="txtAge" type="text"/>			<requiredfieldvalidator2> RangeValidator1</requiredfieldvalidator2>																																																			
Date of Journey	<input id="txtDate" type="text"/> <div style="display: flex; justify-content: space-around;"> < November 2019 > </div> <table border="1" style="margin-top: 5px; border-collapse: collapse; text-align: center;"> <tr> <th>Sun</th><th>Mon</th><th>Tue</th><th>Wed</th><th>Thu</th><th>Fri</th><th>Sat</th></tr> <tr> <td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>1</td><td>2</td></tr> <tr> <td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr> <td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td></tr> <tr> <td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td></tr> <tr> <td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td></tr> <tr> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> </table>						Sun	Mon	Tue	Wed	Thu	Fri	Sat	27	28	29	30	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5	6	7
	Sun	Mon	Tue	Wed	Thu	Fri	Sat																																																
	27	28	29	30	31	1	2																																																
	3	4	5	6	7	8	9																																																
	10	11	12	13	14	15	16																																																
	17	18	19	20	21	22	23																																																
	24	25	26	27	28	29	30																																																
	1	2	3	4	5	6	7																																																
	<input type="button" value="Confirm Booking"/>																																																						
	<input type="button" value="btnConfirm"/>																																																						
Label1 => Set Id as lblIMsg																																																							

To design the above write the below code under <div> tag of the “Validations2.aspx” file:

```

<asp:ScriptManager ID="ScriptManager1" runat="server" />
<table align="center">
    <caption>Journey Planner</caption>
    <tr>
        <td>Enter Name:</td>
        <td><asp:TextBox ID="txtName" runat="server" /></td>
        <td><asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server" /></td>
    </tr>
    <tr>
        <td>Enter Age:</td>
        <td><asp:TextBox ID="txtAge" runat="server" /></td>
        <td><asp:RequiredFieldValidator ID="RequiredFieldValidator2" runat="server" /><br />
            <asp:RangeValidator ID="RangeValidator1" runat="server" /></td>
        </tr>

```

```

<tr>
    <td>Date of Journey:</td>
    <td><asp:UpdatePanel ID="UpdatePanel1" runat="server">
        <ContentTemplate>
            <asp:TextBox ID="txtDate" runat="server" />
            <asp:ImageButton ID="imgDate" runat="server" ImageUrl="~/Icons/Calendar.ico" Width="20"
                Height="20" ImageAlign="AbsMiddle" />
            <asp:Calendar ID="cldDate" runat="server" Visible="false" />
        </ContentTemplate>
    </asp:UpdatePanel></td>
    <td><asp:RequiredFieldValidator ID="RequiredFieldValidator3" runat="server" /><br />
        <asp:RangeValidator ID="RangeValidator2" runat="server" /></td>
    </tr>
    <tr>
        <td colspan="2" align="center"><asp:Button ID="btnConfirm" runat="server" Text="Confirm Booking" /></td>
    <td></td>
    </tr>
    <tr>
        <td colspan="3"><asp:Label ID="lblMsg" runat="server" /></td>
    <td></td>
    </tr>
</table>

```

In design view select all 5 Validation Controls at a time, hit "F4" and set these 3 properties in property window:

Display: Dynamic **ForeColor:** Red **SetFocusOnError:** True

Now go to properties of each Validation Control and set the below properties:

RequiredFieldValidator1:

Id: rfvName **ControlToValidate:** txtName **ErrorMessage:** Name field can't be left empty.

RequiredFieldValidator2:

Id: rfvAge **ControlToValidate:** txtAge **ErrorMessage:** Age field can't be left empty.

RequiredFieldValidator3:

Id: rfvDate **ControlToValidate:** txtDate **ErrorMessage:** Journey date can't be left empty.

RangeValidator1:

Id: rvAge **ControlToValidate:** txtAge **MinimumValue:** 18 **MaximumValue:** 65
ErrorMessage: Traveler's age should be between 18 - 65 years. **Type:** Integer

RangeValidator2:

Id: rvDate **ControlToValidate:** txtDate **Type:** Date
ErrorMessage: Travel date should be with-in 90 days from current date.

Now go to "Validations2.aspx.cs" file and write the following code:

Code under Page_Load:

```

rvDate.MinimumValue = DateTime.Now.ToShortDateString();
rvDate.MaximumValue = DateTime.Now.AddDays(90).ToShortDateString();

```

```
if(!IsPostBack) {  
    txtName.Focus();  
}
```

Code under Image Button Click:

```
if(cldDate.Visible)  
{  
    cldDate.Visible = false;  
}  
else  
{  
    cldDate.Visible = true;  
}
```

Code under Calendar Selection Changed:

```
txtDate.Text = cldDate.SelectedDate.ToShortDateString();  
cldDate.Visible = false;
```

Code under Confirm Button Click:

```
if.IsValid) {  
    lblMsg.ForeColor = System.Drawing.Color.Green;  
    lblMsg.Text = "Your booking is confirmed.";  
}  
else {  
    lblMsg.ForeColor = System.Drawing.Color.Red;  
    lblMsg.Text = "Validations failed please re-check your data.";  
}
```

Note: When we run the **Web Form** we face a problem i.e., when we click on the **ImageButton** to launch the **Calendar** the **ImageButton** requires to perform a **CallBack**, but the **Validation Control** will not allow that to happen until all the **Validations** are successful so to avoid that problem use **ValidationGroup** property of **ImageButton** and set it with some value, for example: "**DateGroup**", so that when we click on **ImageButton** it will validate only the controls which comes under this group and currently we don't have any other control coming under this group.

CompareValidator: this control is used for performing 3 types of **validations**, those are:

1. Compare the value of an Input Control with another Input Control.
2. Compare the value of an Input Control with a given fixed value.
3. Data Type Check.

Properties specific to CompareValidator:

1. **ControlToCompare:** this property is to specify the Id of an Input Control with whom **ControlToValidate** has to be compared.
2. **ValueToCompare:** this property is to specify a value with what **ControlToValidate** must be compared.
3. **Operator:** this property is to specify the type of comparison that must be performed, and it can be set with any of the following values: **Equal [d]**, **NotEqual**, **GreaterThan**, **GreaterThanOrEqualTo**, **LessThan**, **LessThanOrEqualTo** and **DataTypeCheck**.

4. **Type:** this is to specify the Data Type for the values which are being compared and can be set with any of the following values: **String [d], Integer, Double, Date and Currency.**

Add a new WebForm in our project naming it as “Validations3.aspx” and design it as following:

Registration Form

Enter Name:	<input id="txtName" type="text"/>	RequiredFieldValidator1																																																	
Enter Password:	<input id="txtPwd" type="password"/>	RequiredFieldValidator2																																																	
Confirm Password:	<input id="txtCPwd" type="password"/>	RequiredFieldValidator3 CompareValidator1																																																	
Date of Birth:	<input id="txtDate" type="text"/> <div style="display: flex; justify-content: space-around; width: 100%;"> < July 2019 > </div> <table border="1" style="margin-top: 5px; border-collapse: collapse; text-align: center;"> <tr><td>Sun</td><td>Mon</td><td>Tue</td><td>Wed</td><td>Thu</td><td>Fri</td><td>Sat</td></tr> <tr><td>30</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> <tr><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr> <tr><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td></tr> <tr><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td></tr> <tr><td>28</td><td>29</td><td>30</td><td>31</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> </table>	Sun	Mon	Tue	Wed	Thu	Fri	Sat	30	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	1	2	3	4	5	6	7	8	9	10	btnDate
Sun	Mon	Tue	Wed	Thu	Fri	Sat																																													
30	1	2	3	4	5	6																																													
7	8	9	10	11	12	13																																													
14	15	16	17	18	19	20																																													
21	22	23	24	25	26	27																																													
28	29	30	31	1	2	3																																													
4	5	6	7	8	9	10																																													
	<input type="button" value="Register"/>	CompareValidator2 CompareValidator3																																																	
		 cld Date  Set visible false																																																	
		<input type="button" value="btnRegister"/>																																																	
Label1 => Set Id as lblMsg																																																			

To design the above write the below code under <div> tag of the “Validations3.aspx” file:

```

<asp:ScriptManager ID="ScriptManager1" runat="server" />


|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                   |     |     |     |     |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|-----|-----|-----|-----|-----|----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|----|---------|
| <b>Registration Form</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                   |     |     |     |     |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| Enter Name:              | <input id="txtName" runat="server" type="text"/>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | RequiredFieldValidator1                                                                                                                                                                                           |     |     |     |     |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| Enter Password:          | <input id="txtPwd" runat="server" textmode="Password" type="password"/>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | RequiredFieldValidator2                                                                                                                                                                                           |     |     |     |     |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| Confirm Password:        | <input id="txtCPwd" runat="server" textmode="Password" type="password"/>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | RequiredFieldValidator3<br>CompareValidator1                                                                                                                                                                      |     |     |     |     |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| Date of Birth:           | <input id="txtDate" runat="server" type="text"/> <div style="display: flex; justify-content: space-around; width: 100%;"> <span>&lt;</span> <span>July 2019</span> <span>&gt;</span> </div> <table border="1" style="margin-top: 5px; border-collapse: collapse; text-align: center;"> <tr><td>Sun</td><td>Mon</td><td>Tue</td><td>Wed</td><td>Thu</td><td>Fri</td><td>Sat</td></tr> <tr><td>30</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> <tr><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr> <tr><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td></tr> <tr><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td></tr> <tr><td>28</td><td>29</td><td>30</td><td>31</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> </table> | Sun                                                                                                                                                                                                               | Mon | Tue | Wed | Thu | Fri | Sat | 30 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | btnDate |
| Sun                      | Mon                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Tue                                                                                                                                                                                                               | Wed | Thu | Fri | Sat |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| 30                       | 1                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 2                                                                                                                                                                                                                 | 3   | 4   | 5   | 6   |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| 7                        | 8                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 9                                                                                                                                                                                                                 | 10  | 11  | 12  | 13  |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| 14                       | 15                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | 16                                                                                                                                                                                                                | 17  | 18  | 19  | 20  |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| 21                       | 22                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | 23                                                                                                                                                                                                                | 24  | 25  | 26  | 27  |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| 28                       | 29                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | 30                                                                                                                                                                                                                | 31  | 1   | 2   | 3   |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
| 4                        | 5                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 6                                                                                                                                                                                                                 | 7   | 8   | 9   | 10  |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
|                          | <input id="btnRegister" runat="server" type="button" value="Register"/>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | CompareValidator2<br>CompareValidator3                                                                                                                                                                            |     |     |     |     |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |  <b>cld Date</b><br> <b>Set visible false</b> |     |     |     |     |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |
|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | <input id="lblMsg" runat="server" type="button" value="lblMsg"/>                                                                                                                                                  |     |     |     |     |     |     |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |         |


```

```

        </ContentTemplate>
    </asp:UpdatePanel>
</td>
<td>
    <asp:CompareValidator ID="CompareValidator2" runat="server" /><br />
    <asp:CompareValidator ID="CompareValidator3" runat="server" />
</td>
</tr>
<tr>
    <td colspan="2" align="center">
        <asp:Button ID="btnRegister" runat="server" Text="Register" />
    </td>
    <td>&nbsp;</td>
</tr>
<tr>
    <td colspan="3"><asp:Label ID="lblMsg" runat="server" /></td>
</tr>
</table>

```

Now select all the 6 Validation Controls at a time, hit “F4” and set the following properties in property window:

Display: Dynamic

ForeColor: Red

SetFocusOnError: True

Now go to properties of each Validation Control and set the following properties:

RequiredFieldValidator1:

Id: rfvName **ControlToValidate:** txtName **ErrorMessage:** Name field can't be left empty.

RequiredFieldValidator2:

Id: rfvPwd **ControlToValidate:** txtPwd **ErrorMessage:** Password field can't be left empty.

RequiredFieldValidator3:

Id: rfvCPwd **ControlToValidate:** txtCPwd **ErrorMessage:** Confirm password field can't be left empty.

CompareValidator1:

Id: cvCPwd **ControlToValidate:** txtCPwd **ControlToCompare:** txtPwd **Type:** String
ErrorMessage: Confirm password should match with password. **Operator:** Equal

CompareValidator2:

Id: cvDate1 **ControlToValidate:** txtDate **Operator:** DataTypeCheck
ErrorMessage: Entered value is not in a valid date format. **Type:** Date

CompareValidator3:

Id: cvDate2 **ControlToValidate:** txtDate **Operator:** LessThanEqual
ErrorMessage: You need to attain 18 years of age for registration. **Type:** Date

Now goto “Validations3.aspx.cs” file and write the following code:

Code under Page_Load:

```

if(!IsPostBack) {
    txtName.Focus();
}

```

```
}
```

```
cvMajor.ValueToCompare = DateTime.Now.AddYears(-18).ToString();
```

Code under Image Button Click:

```
if (cldDate.Visible) {  
    cldDate.Visible = false;  
}  
else {  
    cldDate.Visible = true;  
}
```

Code under Calendar Selection Changed:

```
txtDate.Text = cldDate.SelectedDate.ToString();  
cldDate.Visible = false;
```

Code under Register Button Click:

```
if (isValid) {  
    lblMsg.ForeColor = System.Drawing.Color.Green;  
    lblMsg.Text = "Your registration is successful.";  
}  
else {  
    lblMsg.ForeColor = System.Drawing.Color.Red;  
    lblMsg.Text = "Validations failed please re-check your data.";  
}
```

RegularExpressionValidator: This control will validate input by using some special characters known as **Regular Expressions** or **Regex** and these are some special characters using which we can perform **data validations** without writing **complex logic**.

Understanding with Regular Expressions:

B => Braces => [] {} ()	
C => Carrot => ^	=> Represents start of an expression
D => Dollar => \$	=> Represents end of an expression

Braces:

[] => these are used to specify the characters which are allowed.

E.g: [a-m] or [A-K] or [0-6] or [A-Za-z0-9]

{ } => these are used to specify the no. of characters that are allowed.

E.g: {1} or {4, 7} or {4,}

() => these are used to specify a list of options which are accepted.

E.g: (com|net|in|edu)

[A-K] => accepts 1 alphabet between A to K.

[a-m]{5} => accepts 5 lower-case alphabets between a to m.

[a-z]{3}[0-9]{5} => accepts 3 lower-case alphabets in the first followed by 5 numeric.

Note: in all the above 3 cases after the validating expression, it will accept anything we enter over there and to overcome this problem we need to make the expressions rigid as following:

`^[a-z]{3}[0-9]{5}$` => same as the last expression above but this is rigid expression i.e., accepts only 8 chars.

Special characters in regular expressions: there are some special characters with pre-defined logic.

`\s` => accepts whitespace

`\S` => doesn't accept whitespace

`\d` => accepts only numeric values.

`\D` => accepts only non-numeric values.

`\w` => accepts only alpha-numeric values.

`\W` => accepts only non-alphanumeric values.

Validation Expressions using Regular Expressions:

<code>\d{6}</code>	=> Indian Postal Code
<code>\d{3}-\d{7}</code>	=> Indian Railway PNR
<code>\d{4} \d{4} \d{4}</code>	=> Aadhar Number
<code>\d{4} \d{4} \d{4} \d{4}</code>	=> Credit Card Number
<code>http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?</code>	=> Website URL Validation.
<code>\w+[\w-\.]*@\w+((-?\w+) (\w*))\.[a-z]{2,3}</code>	=> Email Validation
<code>[6-9]\d{9}</code>	=> Mobile No. validation that checks No. starts with 6 or 7 or 8 or 9 and will be maximum 10 and minimum 10 digits.
<code>[0][6-9]\d{9}</code>	=> Mobile No. validation that checks No. starts with 0 and after that 6 or 7 or 8 or 9 and will be maximum 11 and minimum 11 digits.
<code>^[0][6-9]\d{9}\$ ^[6-9]\d{9}\$</code>	=> Mobile No. validation which checks if the No. starts with 0 then it is 11 digit or else it is 10 digit.
<code>^\d{6,8}\$ ^[6-9]\d{9}\$</code>	=> Validation for either 6-8 digits landline no or 10 digit Mobile No. that starts with 6 or 7 or 8 or 9.

Note: We can use **Regular Expression** in ASP.NET by using the **RegularExpressionValidator** control and specify the **Regular Expression** under its "**ValidationExpression**" property.

Properties specific to RegularExpressionValidator:

1. **ValidationExpression:** by using this property we need to specify the "**Validation Expression**" for the validation, either by choosing from a list of pre-defined expressions and this can be done by clicking on the button beside the property which will launch "**Regular Expression Editor**" and it lists a set of pre-defined expressions for **Email Id Validation**, **Website URL Validation**, etc or else **default** value will be **Custom** using which we can enter or specify our own **Validation Expression**.

Add a new WebForm in our project naming it as "Validations4.aspx" and design it as following:

Company Registration Form

Company Name:	<input id="txtName" type="text"/>	RequiredFieldValidator1
Contact No:	<input id="txtPhone" type="text"/>	RegularExpressionValidator1
Email Id:	<input id="txtEmail" type="text"/>	RegularExpressionValidator2
Website URL:	<input id="txtUrl" type="text"/>	RegularExpressionValidator3
<input style="width: 100px; height: 30px; border: 1px solid black; background-color: #f0f0f0; color: black; font-weight: bold; font-size: 10pt; margin-bottom: 5px;" type="button" value="Register"/>		btnRegister
Label1 => Set Id as lblMsg		

```

<table align="center">
    <caption>Company Registration Form</caption>
    <tr>
        <td>Company Name:</td>
        <td><asp:TextBox ID="txtName" runat="server" /></td>
        <td><asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server" /></td>
    </tr>
    <tr>
        <td>Contact No:</td>
        <td><asp:TextBox ID="txtPhone" runat="server" /></td>
        <td><asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server" /></td>
    </tr>
    <tr>
        <td>Email Id:</td>
        <td><asp:TextBox ID="txtEmail" runat="server" /></td>
        <td><asp:RegularExpressionValidator ID="RegularExpressionValidator2" runat="server" /></td>
    </tr>
    <tr>
        <td>Website Url:</td>
        <td><asp:TextBox ID="txtUrl" runat="server" /></td>
        <td><asp:RegularExpressionValidator ID="RegularExpressionValidator3" runat="server" /></td>
    </tr>
    <tr>
        <td colspan="2" align="center"><asp:Button ID="btnRegister" runat="server" Text="Register" /></td>
        <td>&nbsp;</td>
    </tr>
    <tr>
        <td colspan="3" style="text-align: center; vertical-align: bottom; padding-top: 10px;"><asp:Label ID="lblMsg" runat="server" /></td>
    </tr>
</table>
```

Now select all the 4 Validation Controls at a time, hit "F4" and set the following properties in property window:

Display: Dynamic

ForeColor: Red

SetFocusOnError: True

Now go to properties of each Validation Control and set the following properties:

RequiredFieldValidator1:

Id: rfvName **ControlToValidate:** txtName **ErrorMessage:** Comapany name field can't be left empty.

RegularExpressionValidator1:

Id: revPhone **ControlToValidate:** txtPhone **ValidationExpression:** ^\d{6,8}\$ | ^[6-9]\d{9}\$
ErrorMessage: Contact no. can either be Land Phone (6-8 digits) or Mobile (10 digits).

RegularExpressionValidator2:

Id: revEmail **ControlToValidate:** txtEmail **ErrorMessage:** Given input is an invalid Email Id format.
ValidationExpression: Click on the button beside and select "Internet e-mail Address" in Regular Expression Editor.

RegularExpressionValidator3:

Id: revUrl **ControlToValidate:** txtUrl **ErrorMessage:** Given input is an invalid Website Url format.
ValidationExpression: Click on the button beside and select "Internet URL" in Regular Expression Editor.

Now go to aspx.cs file and write the following code:

Code under Page_Load:

```
if(!IsPostBack)
{
    txtName.Focus();
}
```

Code under Register Button Click:

```
if (IsValid) {
    lblMsg.ForeColor = System.Drawing.Color.Green;
    lblMsg.Text = "Your company registration is successful.";
}
else {
    lblMsg.ForeColor = System.Drawing.Color.Red;
    lblMsg.Text = "Your company registration failed because validation errors.";
}
```

CustomValidator: this control doesn't have any **pre-defined logic** to perform **Data Validations** i.e., we need to implement all the logic on our own just like implementing manual **Java Script** code but the advantage with this is we can implement logic to perform **Client Side Validation** by using **Java Script** code as well as we can also implement logic to **re-validate** the data by using **C#** code on **Server Side Validation**, so that in case **Java Script** is disabled at **Client's Browser** still data gets validated on the **Server**.

Note: We don't require (optional) setting "**ControlToValidate**" property for **CustomValidator** so this **Validator** is capable of validating more than 1 **Input Control** also.

Member's specific to CustomValidator:

1. **ClientValidationFunction:** by using this **property** we need to specify the name of **Java Script** function in which we have implemented logic for **Client-Side Validation**.

2. **ServerValidate:** this is an **event** under which we need implement the logic to perform **Server Side Validation**.

Add a new WebForm in our project naming it as "Validations5.aspx" and design it as following:

Feedback Form

Name:	txtName	RequiredFieldValidator1
Phone No:	txtPhone	RegularExpressionValidator1 CustomValidator1
Email Id:	txtEmail	RegularExpressionValidator2
Comments	txtComments	RequiredFieldValidator2 CustomValidator2
Submit  btnSubmit		
Label1 => Set Id as lblMsg		

```
<table align="center">
<caption>Feedback Form</caption>
<tr>
<td>Name:</td>
<td><asp:TextBox ID="txtName" runat="server" /></td>
<td><asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server" /></td>
</tr>
<tr>
<td>Phone No:</td>
<td><asp:TextBox ID="txtPhone" runat="server" /></td>
<td rowspan="2"><asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server" /><br />
<asp:CustomValidator ID="CustomValidator1" runat="server" /><br />
<asp:RegularExpressionValidator ID="RegularExpressionValidator2" runat="server" /></td>
</tr>
<tr>
<td>Email Id:</td>
<td><asp:TextBox ID="txtEmail" runat="server" /></td>
</tr>
<tr>
<td>Comments:</td>
<td><asp:TextBox ID="txtComments" runat="server" Rows="3" TextMode="MultiLine" /></td>
<td><asp:RequiredFieldValidator ID="RequiredFieldValidator2" runat="server" /><br />
<asp:CustomValidator ID="CustomValidator2" runat="server" /></td>
</tr>
<tr>
<td colspan="2" align="center"><asp:Button ID="btnSubmit" runat="server" Text="Submit" /></td>
<td>&nbsp;</td>
</tr>
```

```

</tr>
<tr>
    <td colspan="3"><asp:Label ID="lblMsg" runat="server" /></td>
</tr>
</table>

```

Now select all the 6 Validation Controls at a time, hit “F4” and set the following properties in property window:

Display: Dynamic

ForeColor: Red

SetFocusOnError: True

Now go to properties of each Validation Control and set the following properties:

RequiredFieldValidator1:

Id: rfvName **ControlToValidate:** txtName **ErrorMessage:** Name field can't be left empty.

RequiredFieldValidator2:

Id: rfvComments **ControlToValidate:** txtComments **ErrorMessage:** Comments field can't be left empty.

RegularExpressionValidator1:

Id: revPhone **ControlToValidate:** txtPhone **ValidationExpression:** ^\d{8}\$|^\d{6-9}\d{9}\$

ErrorMessage: Phone no. can either be landline (8 digits) or mobile (10 digits).

RegularExpressionValidator2:

Id: revEmail **ControlToValidate:** txtEmail **ErrorMessage:** Given input is an invalid Email Id format.

ValidationExpression: Click on the button beside and select “Internet e-mail Address” in Regular Expression Editor.

CustomValidator1:

Id: cvPhoneOrEmail **ErrorMessage:** Either Phone No. or Email Id must be provided.

ClientValidationFunction: PhoneOrEmail

Now go to events of **CustomValidator1** and double click on “**ServerValidate**” which will generate an Event Handler (**cvPhoneOrEmail_ServerValidate**) for implementing **server-side** validation code.

CustomValidator2:

Id: cvComments **ControlToValidate:** txtComments **ClientValidationFunction:** Check50Chars

ErrorMessage: Comments should be minimum 50 chars of length.

Now go to events of **CustomValidator2** and double click on “**ServerValidate**” which will generate an Event Handler (**cvComments_ServerValidate**) for implementing **server-side** validation code.

Now go to “Source View” and write the below code inside of <head> section:

```

<script>
    function PhoneOrEmail(source, args) {
        if (txtPhone.value.trim().length == 0 && txtEmail.value.trim().length == 0) {
            args.IsValid = false;
        }
        else {
            args.IsValid = true;
        }
    }

```

```

function Check50Chars(source, args) {
    if (args.Value.trim().length < 50) {
        args.IsValid = false;
    }
    else {
        args.IsValid = true;
    }
}
</script>

```

Now go to “aspx.cs” file and write the following code:

Code under Page_Load:

```

if(!IsPostBack) {
    txtName.Focus();
}

```

Code under cvPhoneOrEmail_ServerValidate:

```

if (txtPhone.Text.Trim().Length == 0 && txtEmail.Text.Trim().Length == 0) {
    args.IsValid = false;
}
else {
    args.IsValid = true;
}

```

Code under cvComments_ServerValidate:

```

if (args.Value.Trim().Length < 50) {
    args.IsValid = false;
}
else {
    args.IsValid = true;
}

```

Code under Submit Button Click:

```

if (isValid)
{
    lblMsg.ForeColor = System.Drawing.Color.Green;
    lblMsg.Text = "Your feedback is received, we will contact you asap.";
}
else
{
    lblMsg.ForeColor = System.Drawing.Color.Red;
    lblMsg.Text = "Your feedback has errors and not been taken.";
}

```

ValidationSummary: this control doesn't perform any **Data Validations** but used only for displaying **Error Messages** i.e., without displaying **Error Messages** beside the **Input Controls** we can display all the **Error Messages** at one

location either on top of the **Page** or bottom of the **Page**. When we place this **Control** on a **Page** it will automatically inherit the **Error Messages** that are associated with all **Validation Controls** that are present on that **Page**. To test workings with **ValidationSummary** Control in our current page i.e., “**Validations5.aspx**”, go to design view and, drag and drop the **ValidationSummary** Control just below the table and set the **ForeColor** property value as “**Red**”.

Now run the **Web Form** and watch the output i.e., if any **Validations** fail all those **Error Messages** will be displayed by the **ValidationSummary** Control in the place where we placed it because this **Control** will inherit all the **Error Messages** of the 6 **Validation Controls** present on the page. But in this case, we also see the **Error Messages** beside the **Input Controls** and to avoid them set the **Text** Property of **Validation Controls** so that **Text** value will be displayed beside **Input Controls** and **Error Messages** will be displayed by the **ValidationSummary** control and to test this go to the properties of all 6 **Validation Controls** on the **Page** and under the **Text** enter the value as “*****”. Now run and watch the difference in output which will display “*****” beside the **Input Control** where the validation failed, and all the **Error Messages** will be displayed by **ValidationSummary** control.

Properties specific to ValidationSummary Control:

1. **ShowSummary:** this is a boolean property with the default value true and if set as false will not display the **ErrorMessage** on Screen.
2. **ShowMessageBox:** this is a boolean property with the default value false and if set as true will display the **Error Messages** in **MessageBox**.
3. **ShowValidationErrors:** this is a boolean property with the default value as true and if set as false **Error Messages** will not be displayed either on the **Screen** or in a **Message Box**.
4. **DisplayMode:** this property is to specify the format how **Error Messages** must be displayed, and we can choose from any of the below 3 values: **BulletedList [d]**, **List** and **SingleParagraph**.

Master Pages

One attribute of a well-designed **Web Site** is a consistent **site-wide** **page layout**. Take the www.nareshit.com website for example; every page has the same content at the **top** and **bottom** of the page. At the very top of each page displays a light gray bar with a list of locations where the branches are located. Beneath that are the company logo, and the core sections: **Home**, **All Courses**, **Software Training**, **Services**, and so forth. Likewise, the bottom of the page includes information about **clients**, **Address**, **contact details**, **quick links** etc.

Another attribute of a well-designed site is the ease with which the site's appearance can be changed. For instance, the look and feel can be changed in the future, perhaps the menu items along the top will expand to include a new section or may be a radically new design with different colors, fonts, and layout will be unveiled. Applying such changes to the entire site should be a fast and simple process that does not require modifying the thousands of web pages that make up the site.

Creating a site-wide page template in **ASP.NET** is possible using **Master Pages**. In a nutshell, a **Master Page** is a special type of **ASP.NET** page that defines the markup that is common among all **Content Pages** (a **Content Page** is an **ASP.NET** page that is bound to the **Master Page**). Whenever a **Master Page**'s layout or formatting is changed, all its **Content Pages** output is immediately updated, which makes applying site-wide appearance and changes as easy by updating and deploying a single file i.e., **Master Page**.

Understanding How Master Pages Work: Building a website with a consistent site-wide page layout requires that each web page emit common formatting markup for example, while each page on www.nareshit.com have their own unique content, each of these pages also render a series of common <div> elements that display the top-level section links: Home, All Courses, Software Training, Services, and so on.

There are a variety of techniques for creating web pages with a consistent look and feel. A naive approach is to simply copy and paste the common layout markup into all web pages, but this approach has several downsides. For starters, every time a new page is created, you must remember to copy and paste the shared content into the page. Such copying and pasting operations are ripe for error as you may accidentally copy only a subset of the shared markup into a new page. And to top it off, this approach makes replacing the existing site-wide appearance with a new one a real pain because every single page in the site must be edited to use the new look and feel.

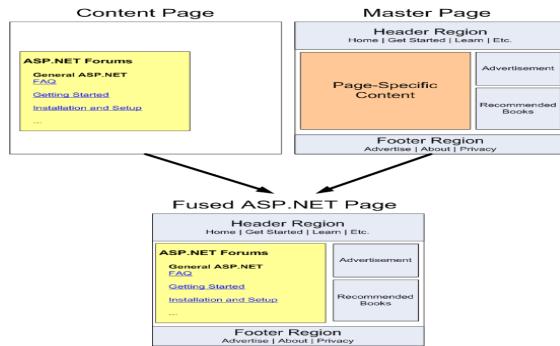
Prior to ASP.NET version 2.0, page developers often placed common markup in User Controls and then added these User Controls to each page. This approach requires the page developer to remember that they need to manually add the User Controls to every new page, and this allowed for easier site-wide modifications because when updating the common markup only the User Controls needed to be modified. Unfortunately, Visual Studio.NET 2002 and 2003 which used to create ASP.NET 1.x applications - rendered User Controls in the Design view as gray boxes. Consequently, page developers using this approach did not enjoy a WYSIWYG design-time environment.

The shortcomings of using User Controls were cited in ASP.NET version 2.0 and Visual Studio 2005 with the introduction of master pages. A master page is a special type of ASP.NET page that defines the site-wide markup and the regions where associated content pages can define their custom markup and those regions are defined by “ContentPlaceHolder” controls. The “ContentPlaceHolder” control simply denotes a position in the master page's control hierarchy where custom content can be injected by a content page.

Below figure shows how a master page might look like. Note that the master page defines the common site-wide layout - the markup at the top, bottom, and right of every page - as well as a “ContentPlaceHolder” in the middle-left where the unique content of each individual web page must come and sit.



Once a master page has been defined it can be bound to a new ASP.NET page and those ASP.NET pages (content pages), include a Content control for each of the master page's ContentPlaceHolder controls. When the content page is visited through a browser the ASP.NET engine creates the master page's control hierarchy and injects the content page's control hierarchy into the appropriate places and the combined control hierarchy is rendered and the resulting HTML is returned to the end user's browser. Below figure illustrates this concept:



Creating a Master Page: To add a Master Page in our site, open the “Add New Item” window and select the option “**WebForms Master Page**” and name it as “**Site.master**” (.master is the extension of a **Master Page**). The first line in the declarative markup is the **@Master** directive and this is like the **@Page** directive that appears in our **Web Pages** and all other attributes will be same as **@Page** directive attributes only.

Initially the code in Master Page will be as below:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.master.cs" Inherits="ControlsDemo.Site" %>
<!DOCTYPE html>
<html>
  <head runat="server">
    <title></title>
    <asp:ContentPlaceHolder ID="head" runat="server"></asp:ContentPlaceHolder>
  </head>
  <body>
    <form id="form1" runat="server">
      <div>
        <asp:ContentPlaceHolder ID="ContentPlaceholder1" runat="server"></asp:ContentPlaceHolder>
      </div>
    </form>
  </body>
</html>
```

Now if we look into the code of the file we find a “**ContentPlaceHolder**” control named “**head**” and this control appears within the **<head>** section and can be used to declaratively add content in to the **<head>** element like style and script code. We also find another “**ContentPlaceHolder**” control named “**ContentPlaceholder1**” and this control appears within the **Web Form** and serves as the region for the content pages - user interface. Delete code which is present inside of **<form>** tag along with the “**ContentPlaceHeader**” and write the below code in it:

```
<div id="divHeader" style="background-color: lightpink; color: aqua; text-align: center; font-size: xx-large">
  Naresh I Technologies
</div>

<div id="divMenu" style="background-color: beige; text-align: center">
  <asp:HyperLink ID="hlHome" runat="server" NavigateUrl="~/Home.aspx" Text="Home" />&nbsp;
```

```

<asp:HyperLink ID="hlMission" runat="server" NavigateUrl "~/Mission.aspx" Text="Mission" />&nbsp;
<asp:HyperLink ID="hlAbout" runat="server" NavigateUrl "~/About.aspx" Text="About" />
</div>

<div id="divContent">
    <asp:ContentPlaceHolder ID="body" runat="server" />
</div>

<div id="divFooter" style="background-color: azure">
    <fieldset style="border: dotted blue; color: brown">
        <legend>Contact Us</legend>
        Address: Opp. Satyam Theatre, Ameerpet, Hyderabad – 16 <br />
        Phone: 2374 6666 <br />
        Email: info@nareshit.com <br />
        Website: www.nareshitc.om </fieldset>
</div>

```

Creating Content Pages: with the above master page created, we are now ready to start creating **ASP.NET Web Pages** that are bound to the **Master Page**. Such **Pages** are referred to as **Content Pages** and we need to create 3 **Content Pages** now with the name: **Home.aspx**, **Mission.aspx** and **About.aspx**.

Add a new **ASP.NET Web Form** to the project and bind it to “**Site.master**” **Master Page** and to do that open “**Add New Item**” window, select the “**Web Form with Master Page**” template, enter the name as “**Home.aspx**” and click “**Add**”, which opens “**Select a Master Page**” dialog box from where you can choose the **Master Page** to use. This action will add a **Content Page** which contains **Page Directive** that points to its **Master Page** and also we find “**Content**” controls binding to Master Page’s “**ContentPlaceHolder**” controls. Currently we have 2 “**ContentPlaceHolder**” controls on **Master Page**, so we find 2 “**Content**” controls on this page and the code will be as following in “**Home.aspx**”:

```

<%@ Page Title="" Language="C#" MasterPageFile("~/Site.master" AutoEventWireup="true"
    CodeBehind="Home.aspx.cs" Inherits="ControlsDemo.Home" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server"></asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="body" runat="server"></asp:Content>

```

Note: **Title** attribute is to set a **title** for our page, give a value to title as “**Home Page**” and then write the following code under the 2 “**Content**” controls:

```

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <style>
        p {
            text-align: justify; color: palevioletred; font-family: Arial; text-indent: 50px;
        }
        ol {
            color: cadetblue; background-color: burlywood
        }
    </style>
</asp:Content>

```

```

<asp:Content ID="Content2" ContentPlaceHolderID="body" runat="server">
<div style="background-color: aqua; color: coral;font-family:'Arial Unicode MS';font-size:larger;font-weight: bold">
    Home Page
</div>
<p>We have set the pace with online learning. Learn what you want, when you want, and practice with the instructor-led training sessions, on-demand video tutorials which you can watch and listen. </p>
<ol>
    <li>150+ Online Courses</li>
    <li>UNLIMITED ACCESS</li>
    <li>EXPERT TRAINERS</li>
    <li>VARIETY OF INSTRUCTION</li>
    <li>ON-THE-GO-LEARNING</li>
    <li>PASSIONATE TEAM</li>
    <li>TRAINING INSTITUTE OF CHOICE</li>
</ol>
</asp:Content>

```

Same as the above add 2 more Content Pages naming them as “Mission.aspx” and “About.aspx”, set the title as “Mission Page” for “Mission.aspx” and “About Page” for “About.aspx”, and write below code under them:

Mission.aspx:

```

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<style>
    p {
        text-align: justify; color: blue; font-family: Calibri; text-indent: 50px;
    }
</style>
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="body" runat="server">
<div style="background-color: aqua; color: coral;font-family:'Arial Unicode MS';font-size:larger;font-weight: bold">
    Our Mission
</div>
<p>We appreciate you taking the time today to visit our web site. Our goal is to give you an interactive tour of our new and used inventory, as well as allow you to conveniently get a quote, schedule a service appointment, or apply for financing. The search for a luxury car is filled with high expectations. Undoubtedly, that has a lot to do with the vehicles you are considering, but at Motors, we think you should also have pretty high expectations for your dealership. </p>
</asp:Content>

```

About.aspx:

```

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<style>
    p {
        text-align: justify; color: green; font-family: 'Agency FB'; text-transform: capitalize; text-indent: 50px;
    }
</style>

```

```

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="body" runat="server">
<div style="background-color: aqua; color: coral;font-family:'Arial Unicode MS';font-size:larger;font-weight: bold">
    About Us
</div>
<p>NareshIT (Naresh i Technologies) is a leading Software Training Institute and provides Job Guarantee Program through Nacre in Hyderabad, Chennai, Bangalore, Vijayawada and across the world with Online Training services. Managed and Lead by IT Professionals with more than a decade experience in leading MNC companies. We are most popular for our training approach that enables students to gain real-time exposure on cutting-edge technologies. </p>
</asp:Content>

```

Now run any of the above 3 **Content Pages** and we notice that they merge with Master Page we have created above and launches, click on any menu in the top will launch other pages in the site.

ViewState

Web applications are **stateless** i.e., these applications do not persist data of 1 request, for using it in the next request. When an application is **stateless**, the server does not store any state about the client session. To test this, add a new **WebForm** naming it as “**HitCount.aspx**” and write the following code in its **<div>** tag:

```

<asp:Button ID="btnCount" runat="server" Text="Hit Count"/>
<br/>
<asp:Label ID="lblCount" runat="server" ForeColor="Red"/>

```

Now goto “HitCount.aspx.cs” file and write the following code in it:

Declarations:

```
int Count = 0;
```

Code under Hit Count Button Click:

```
Count += 1;
lblCount.Text = "Hit Count: " + Count;
```

Now run the **WebForm** and click on the button for any no. of times still it will display the output as **1** only because every time we click on the button, server will create the instance of our page class, initializes “**Count**” with “**0**”, increments the value of Count to “**1**”, sends the output to client machine and destroys the page instance, and this process repeats every time we click on the button so every time the value of Count is “**1**” only and this is called as “**Stateless**” behavior. To overcome the above problem, we use **ViewState** which is a technique used to store user data on page at the time of **post back** of a web page.

Syntax of storing a value into ViewState:

```
ViewState[string name] = value (Object);
```

Syntax of accessing a value from ViewState:

```
Object value = ViewState[string name];
```

To resolve the problem in our previous page, rewrite the code under click of the button as following:

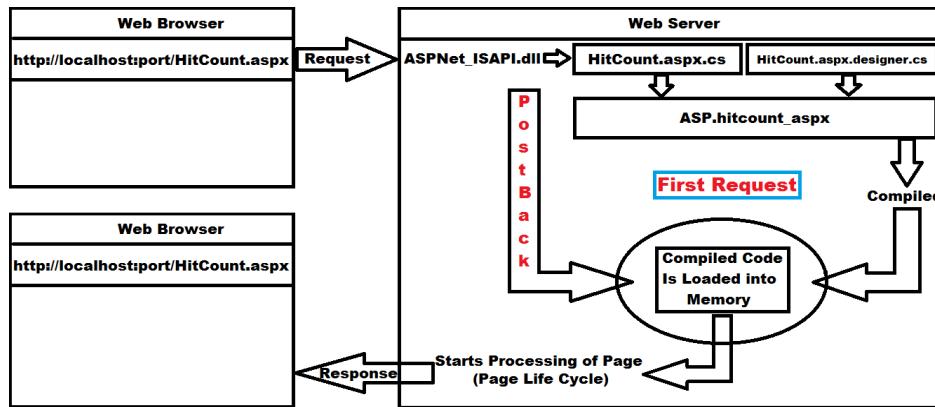
```

if (ViewState["Counter"] == null)
    Count = 1;
else
    Count = (int)ViewState["Counter"] + 1;
ViewState["Counter"] = Count;
lblCount.Text = "Hit Count: " + Count;

```

Note: Same as the above every **ASP.NET Server Control** will also maintain the state of its values and we call it as **ControlState** which will hold the values of **controls** and restores the value to **control** after page **Postback**.

Processing of a Web Page by the Web Server



When the request comes from a client for the first time to a page, `ASPNET_ISAPI.dll` on the **Web Server** will take the request, creates a new class "`ASP.hitcount_aspx`" inheriting from the class "`HitCount`" which is defined on "`HitCount.aspx.cs`" and "`HitCount.aspx.designer.cs`" files, compiles the new class and loads the compiled code into memory, and if at all the request comes for the second time (Post Back) for the page which is already compiled and loaded into memory, without recompiling the page again, the compiled code in the memory will be used for starting the life cycle of page.

Page Life Cycle

Life cycle of a page describes how a **Web Page** gets **processed** on the **server** i.e., when browser sends a request for that page.

Page Request: The page request occurs before the **page life cycle begins**, i.e., when the page is requested by a user, server determines whether the page needs to be **parsed** and **executed** (therefore beginning the **life cycle** of a page) or whether a **cached version** of the page can be sent in response without **processing** the page.

Various stages in the life cycle of a page will be as following:

Stage1: Start => in this stage the page properties such as **request** and **response** are created and set, and at this stage the **page** also determines whether the request is a **post back** or a **new request (get)** and sets the **IsPostBack** property, which will be **true** if it's a **Post Back** request or else **false** if it's a **first** or **get** request.

Stage2: Initialization => during this stage all the **controls** on the page will be **created** and each controls **Unique Id** property will be set.

Stage3: Load => in this stage if the current request is a **post back**, control properties are loaded with information retrieved from **View State (Control State)**.

Stage4: Validation => in this stage any **server-side validations** that are required will be performed and sets the **IsValid** property of **individual validator controls** and then of the **page**.

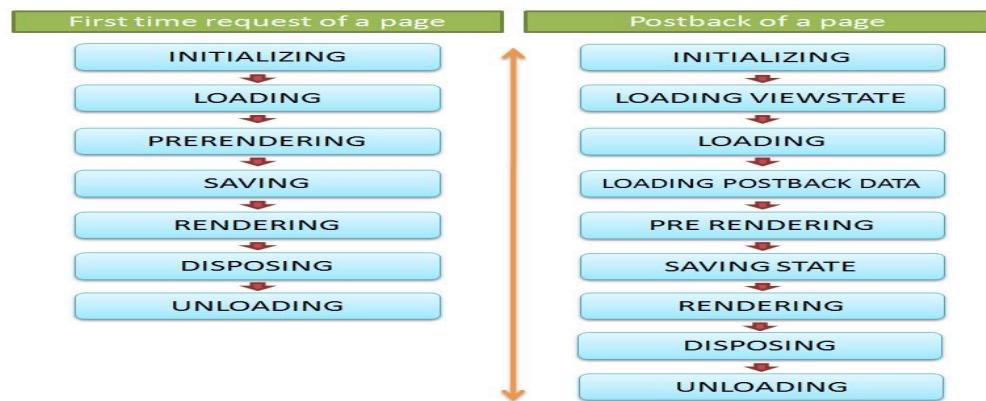
Stage5: Event Handling => if the request is a **post back**, then all the controls corresponding **event handlers** are called (**which is required only**) along with any **cached events** also.

Stage6: Rendering => in this stage the page is **rendered** i.e., converts into **Html** and before rendering **view state** and **control state** is saved for the **page** and all **controls**.

Note: During the rendering stage, page calls the **RenderControl** method on each control and writes its **output** to the **Output Stream** object of Page's **Response** property.

Stage7: UnLoad: => after the page has been **rendered** and the output is **sent** to client, it's ready to **discard** and **Unload** event is raised, and at this stage the **Request** and **Response** properties are destroyed, and **cleanup** is performed.

Series of actions that are performed between the first request and postback request of a page:



Exception Handling in Web Applications

Whenever a **runtime error** occurs in a program, **CLR** will internally create an **instance** of **Exception** class which is matching with the **Exception** that got occurred and **throws** that instance, which will now cause **abnormal termination** of the program and displays an **error message** describing the reason for termination.

When any **runtime error** occurs in a **Web Page** immediately details of that error will be show to end users on the browser screen and to check that add a new **WebForm** under project naming it as “**ExceptionHandling.aspx**” and design it as following:

Division Calculator

Enter 1 st number:	txtNum1
Enter 2 nd number:	txtNum2
Result of Division:	txtResult
Divide	Reset

```
<table align="center">
<caption>Division Calculator</caption>
<tr>
<td>Enter 1<sup>st</sup> number:</td>
<td><asp:TextBox ID="txtNum1" runat="server"/></td>
</tr>
<tr>
<td>Enter 2<sup>nd</sup> number:</td>
<td><asp:TextBox ID="txtNum2" runat="server"/></td>
</tr>
<tr>
<td>Result Obtained:</td>
<td><asp:TextBox ID="txtResult" runat="server"/></td>
</tr>
<tr>
<td colspan="2" align="center">
<asp:Button ID="btnDivide" runat="server" Text="Divide"/>
<asp:Button ID="btnReset" runat="server" Text="Reset"/>
</td>
</tr>
</table>
```

Now go to “aspx.cs” file and write the following code:

Code under Page_Load:

```
if (!IsPostBack)
{
    txtNum1.Focus();
}
```

Code under Divide Button Click:

```
int num1 = int.Parse(txtNum1.Text);
int num2 = int.Parse(txtNum2.Text);
int result = num1 / num2;
txtResult.Text = result.ToString();
```

Code under Reset Button Click:

```
txtNum1.Text = txtNum2.Text = txtResult.Text = "";
```

```
txtNum1.Focus();
```

Now run the **Web Page** without debugging i.e., Ctrl + F5 and check the output and here we have a chance of getting 3 types of Exceptions like **FormatException**, **DivideByZeroException** and **OverflowException** when we enter wrong values in the Textbox's and whenever we get the **Exception** it will abnormally terminate the program and shows an “**Yellow Screen**” displaying the details of **Exception**. The “**Yellow Screen**” which is shown has a problem i.e. it will display the source code of the line where we got error and along with that it also displays 2 lines of code before and 2 lines of code after the error line and the problem here is if those lines contain any complex logic, end users can view that logic and to avoid displaying that “**Yellow Screen**” to users we are provided with **3 Error Handling** options in **ASP.NET**, those are:

1. Block Level Error Handling
2. Page Level Error Handling
3. Application-Level Error Handling

Block Level Error Handling: this is performed by using the very traditional structured error handling technique i.e., try and catch blocks and when code is enclosed under these blocks exceptions get handled, stopping abnormal termination. To handle error, we need to use try and catch blocks as following:

```
try
{
    -Statements which will cause any runtime errors.
    -Statements which should not execute when the error got occurred.
}
catch(<exception class name><var>)
{
    -Statements which should execute only when the error got occurred.
}
```

To test “block level error handling” re-write the code under the divide button of our previous page as following:

```
try
{
    int num1 = int.Parse(txtNum1.Text);
    int num2 = int.Parse(txtNum2.Text);
    int result = num1 / num2;
    txtResult.Text = result.ToString();
}
catch (Exception ex) {
    Response.Redirect("ErrorPage.aspx?ErrorMessage=" + ex.Message);
}
```

Add a Web Form under the project naming it as "ErrorPage.aspx" and write the below code under its <div> tag:

```
<h1 style="background-color: yellowgreen; color: orangered; text-align:center">Error Page</h1>
<font size="4">There is an error in the application and the details of the error are:
<asp:Label ID="lblMsg" runat="server" ForeColor="Red" />
</font>
```

Now go to "ErrorPage.aspx.cs" file and write the following code under Page_Load method:

```
lblMsg.Text = Request.QueryString["ErrorMessage"];
```

After doing all the above, run “ExceptionHandling.aspx” page again and now whenever any Exception occurs in the page without displaying the “Yellow Screen” it will redirect to “ErrorPage.aspx” and displays the error details on that page.

Note: The drawback of “Block Level Error Handling” is we need to add these **try** and **catch** blocks under each method where there is a chance of getting any exception and it is a tedious and lengthy process, so to overcome this problem use “Page Level Error Handling”.

Page Level Error Handling: this is a process of handling errors that occur in the whole page by using an **Event Handler** method i.e., “**Page_Error**” and by implementing this method under the class any error that occur in the page can be handled. When this method is implemented under our page class, wherever the error occurs in the page, control will directly transfer to this method, so we need to implement logic for handling errors under this method.

To test using the above process, first delete try and catch blocks we have added to the logic under Divide button click event handler and add the following event handler method to the class:

```
protected void Page_Error(object sender, EventArgs e) {
    Exception ex = Server.GetLastError();
    Server.ClearError();
    Response.Redirect("ErrorPage.aspx?ErrorMessage=" + ex.Message);
}
```

Now run the **Web Page** again and watch the output which will be same as previous, but the only difference is here we can handle all errors of a page with single method without writing multiple try and catch blocks.

Application-Level Error Handling: in page level error handling we need to implement a separate **Page_Error** method for each **Web Page**, so when we have multiple **Web Pages**, the process becomes tedious and lengthy. To avoid this problem, we are provided with **Application-Level Error Handling** which can be performed by implementing an Event Handler method known as **Application_Error**.

Application_Error Event Handler is capable of handling exceptions that occur under any page of the site and this method should be implemented under a special class known as “**Global**”. We find this class under our project in a file “**Global.asax**” and this file will be present under every **Web Application** project by default. To view the **Global** class, open the **Global.asax** file present under our **Web Application** project and there we find the **Global** class inheriting from a pre-defined class “**HttpApplication**” of **System.Web** namespace and by default the class contains an **Event Handler** method with the name **Application_Start**.

To test the process of application-level error handling, first comment the “**Page_Error**” Event Handler method we defined earlier, now open **Global.asax** file, rename the method “**Application_Start**” present in the class as “**Application_Error**” and write the following code in it:

```
protected void Application_Error(object sender, EventArgs e)
{
    Exception ex = Server.GetLastError();
    Server.ClearError();
    if(ex.InnerException != null)
```

```
Response.Redirect("ErrorPage.aspx?Message=" + ex.InnerException.Message);
else
    Response.Redirect("ErrorPage.aspx?Message=" + ex.Message);
}
```

In the above case when an error occurs in a **Web Page** and was not handled over there, server will raise another Exception there i.e. **"HttpUnhandledException"** and then control is transferred to **Application_Error** event handler, so here **"ex.Message"** returns the error message associated with **HttpUnhandledException** class but not of the actual fired Exception, so to get the error message of the actual fired **Exception** we need to use the statement **"ex.InnerException.Message"**.

State Management

Web Applications are **stateless** i.e., we can never access the values of **1 request**, in the **next request** of the **same page** or **other pages** also. But sometimes we need the values of **1 request**, in the **next request** to **same page** or **other pages** and to overcome this problem and maintain the **state of values** between **multiple requests** to the **same page** or between **different pages** we are provided with the concept called as **State Management**.

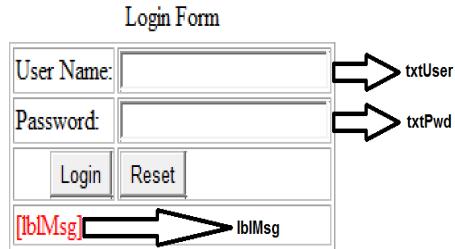
In ASP.NET to maintain the state of values we are provided with various techniques like:

1. View State
2. Query String
3. Hidden Field
4. Cookie
5. Session
6. Application

To understand **State Management** first open a new **ASP.NET Web Application** project, name it as **"ASPStateMgmt"**, specify the location to save as our personal folder, click ok and in the window opened choose **"Empty Project Template"**, check **"Web Forms CheckBox"** and click on **Create Button**. Now let's host the project under Local IIS and to do that open Solution Explorer and under the project - double click on **"Properties"** node which opens the Project Property window, click on the **"Web"** in LHS and now on the right, under **"Servers"** choose **"Local IIS"** option in the **"DropDownList"**, click on **"Create Virtual Directory"** button, and click on **"Save"** icon in Visual Studio Toolbar.

Option 1: ViewState is a mechanism to preserve the values of the Page and Controls between Post backs. It is a **Page-Level State Management** technique i.e., it maintains the state of values for a single user between multiple requests of 1 page (**Single User-Local Data**).

Creating a Login Page with 3 failure attempts: Add 3 new **Web Forms** under the project naming them as **"LoginWithFailureCount.aspx"**, **"SuccessWithFailureCount.aspx"** and **"FailureWithFailureCount.aspx"**, and design **"LoginWithFailureCount.aspx"** as below:



```

<table align="center">
<caption>Login Form</caption>
<tr>
<td>User Name:</td>
<td><asp:TextBox ID="txtUser" runat="server" /></td>
</tr>
<tr>
<td>Password:</td>
<td><asp:TextBox ID="txtPwd" runat="server" TextMode="Password" /></td>
</tr>
<tr>
<td colspan="2" align="center">
<asp:Button ID="btnLogin" runat="server" Text="Login" />
<asp:Button ID="btnReset" runat="server" Text="Reset" />
</td>
</tr>
<tr>
<td colspan="2">
<asp:Label ID="lblMsg" runat="server" ForeColor="Red" />
</td>
</tr>
</table>

```

[Now go to LoginWithFailureCount.aspx.cs file and write the below code:](#)

[Code under Page_Load Event Handler:](#)

```

if (!IsPostBack) {
    txtUser.Focus();
    ViewState["FailureCount"] = 0;
}

```

[Code under Login Button Click Event Handler:](#)

```

if (txtUser.Text == "admin" && txtPwd.Text == "admin") {
    Response.Redirect("SuccessWithFailureCount.aspx?Name=" + txtUser.Text);
}
else {
    int Count = (int)ViewState["FailureCount"] + 1;
    if(Count == 3)
    {
        Response.Redirect("FailureWithFailureCount.aspx?Name=" + txtUser.Text + "&Count=" + Count);
    }
}

```

```

        }
        ViewState["FailureCount"] = Count;
        lblMsg.Text = Count + " attempts failed and the maximum is 3.";
    }
}

```

Code under Reset Button Click Event Handler:

```

lblMsg.Text = "";
txtUser.Text = txtPwd.Text = "";
txtUser.Focus();

```

Now go to SuccessWithFailureCount.aspx.cs file and write the below code:

Code under Page_Load Event Handler:

```

string Name = Request.QueryString["Name"];
Response.Write("Hello " + Name + ", welcome to the site.");

```

Now go to FailureWithFailureCount.aspx.cs file and write the below code:

Code under Page_Load:

```

string Name = Request.QueryString["Name"];
int Count = int.Parse(Request.QueryString["Count"]);
Response.Write("Hello " + Name + ", you have failed all the " + Count + " attempts to login.");

```

Now run the “[Login Page](#)”, enter valid credentials, and click on [Login](#) button which will redirect to “[SuccessPage](#)” and if the credentials are wrong it will give a chance for another 2 times to correct the values and if at all the [3 attempts](#) fail then it will redirect to “[FailurePage](#)”.

In our above code to maintain the [FailureCount](#) we have used [ViewState](#) because [Web Applications](#) are [Stateless](#) and [ViewState](#) is used for maintaining the [state of values](#), which holds values of a page between multiple requests of that page for a single user i.e., “[Single User-Local Data](#)”.

Where does Web Application store ViewState Values?

Ans: [Web Applications](#), store [ViewState](#) values on the same page only i.e., while rendering the output of a page it writes [ViewState](#) values into that rendered output and sends them to the [browser](#), so we can view those values using “[View Page Source](#)” option of the browser and there we find a “[Hidden Field](#)” with the name “[__ViewState](#)” that contains [ViewState](#) values stored in “[Base64 Encrypted String](#)” format. Next time when the page is [submitted](#) to the [Server](#) all the [ViewState](#) values are carried to the [Server](#) and there [Server](#) will [read](#) and [decrypt](#) those values.

Drawbacks of ViewState:

1. [ViewState](#) values are transported between [Browser](#) and [Server](#) every time we perform a [Post Back](#), so in case if we store huge volumes of data in [ViewState](#) then transporting all that data between [Browser](#) and [Server](#) will increase the [network traffic](#).

2. As discussed above [ViewState](#) values are accessible only with the same [Page](#) and that is the reason why we have explicitly transported “[Name & Failure Count](#)” values to “[FailureWithFailureCount.aspx](#)” as a [Query String](#), because in the second page we can’t access first page [ViewState](#) values.

Disabling ViewState in a WebPage: It’s possible to disable [ViewState](#) either for a particular [Web Page](#) or for the whole [Application](#) also which can be done in 2 different ways:

Page Level Disabling: by setting “EnableViewState” attribute of “Page Directive” value as “false” we can disable ViewState values for a particular Page. To test this, go to “LoginWithFailureCount.aspx” and add “EnableViewState” attribute to “Page Directive” as following:

```
<%@ Page EnableViewState="false" ... %>
```

Note: now if we run the WebForm we get “NullReferenceException” because ViewState is disabled.

Application Level Disabling: We can also disable ViewState application level i.e., under the Web.config file so that ViewState will not work in any page and to do that write the following code under <system.web> tag of Web.config file.

```
<pages enableViewState ="false"/>
```

Note: Control State is turned “On” by default for every control on the page regardless of whether it is used during a post-back or not and serializes the data, and we can’t turn off the Control State of controls.

Option 2: Query Strings this is another option by using which we can maintain the state of required values by concatenating those values to page URL as we have performed in our previous example.

Syntax: PageUrl?Key=Value&Key=Value&.....&Key=Value

For example, in our previous pages we have used query strings to transfer values to other pages as following:

```
Response.Redirect("SuccessWithFailureCount.aspx?Name=" + txtUser.Text);  
Response.Redirect("FailureWithFailureCount.aspx?Name=" + txtUser.Text + "&Count=" + Count);
```

Drawbacks of Query Strings:

1. By using this we can pass values only to a particular page and that to when we are transferring control either by using Server.Transfer or Response.Redirect methods.
 2. We can't carry more volumes of data by using a Query String because it supports only 2048 bytes of data.
 3. Passing values by Query Strings is not secured because those values are visible in the address bar of browser.
-

Option 3: Hidden Fields By using Hidden Field control also, we can maintain the state of values within a particular page and more over as discussed earlier ViewState maintains its values thru Hidden Fields only. So, without using ViewState we can use Hidden Fields and maintain the state of values on our page.

Hidden Fields Vs ViewState:

1. In case of ViewState, data is secured because it is stored in a “Base64 Encrypted String” format whereas Hidden Field values are not encrypted, and if encryption is required, we need to explicitly perform it by implementing our own logic.
2. ViewState values are not accessible to other pages whereas we can read Hidden Field values on the page where we are submitting, by using Request object.

Add 2 new Web Forms in the project naming them as “HitCountWithHiddenField.aspx” and “NewForm.aspx” and write the following code under “<div>” tag of 1st form:

```
<asp:Button id="Button1" runat="server" Text="Hit Count with View State" />  
<asp:Label ID="Label1" runat="server" ForeColor="red" />  
<br />
```

```

<asp:Button id="Button2" runat="server" Text="Hit Count with Hidden Field" />
<asp:HiddenField ID="hfCount" runat="server" Value="0" />
<asp:Label ID="Label2" runat="server" ForeColor="red" />
<br />
<asp:Button ID="Button3" runat="server" Text="Launch New Form"PostBackUrl="~/NewForm.aspx" />

```

Write the below code under Page_Load Event Handler of HitCountWithHiddenField.aspx:

```

if(!IsPostBack)
{
    ViewState["HitCount"] = 0;
}

```

Write the below code under "Hit Count with View State" button Click Event Handler of 1st form:

```

int Count = (int)ViewState["HitCount"] + 1;
ViewState["HitCount"] = Count;
Label1.Text = "Hit Count with View State: " + Count;

```

Write the below code under "Hit Count with Hidden Field" button Click Event Handler of 1st form:

```

int Count = int.Parse(hfCount.Value) + 1;
hfCount.Value = Count.ToString();
Label2.Text = "Hit Count with Hidden Field: " + Count;

```

Write the below code under Page_Load of NewForm.aspx:

```

int Count1 = int.Parse(Request.Form["hfCount"]);
Response.Write($"Value of Hidden Field is: {Count1}");
//int Count2 = (int)ViewState["Counter"]; //Error if un-commented

```

Now run the 1st form, click on the 2 **Hit Count** buttons, and allow the **Count** value to increment and at any point of time go to **"View Page Source"** option on browser and you can see the **Hidden Field** value but not **ViewState** value i.e., it will be in **encrypted** format. Now click on **"Launch New Form"** button which will display the **"Count"** value of previous page stored in **Hidden Field** on the new page.

Option 4: Cookies

A **Cookie** is a small piece of text that is used to store **user-specific information** and that information can be read by the **Web Application** whenever user visits the **Site**. When a user requests for a **Web Page**, **Web Server** sends not just a page, but also a cookie containing the **Date** and **Time**.

Cookies are stored in a folder on the user's hard disk and when the user requests for the **Web Page** again, browser looks on the hard disk for **Cookies** associated with the **Web Page** and sends them to the **Server**. Browser stores cookies separately for each different site visited.

By using **Cookies** also, we can maintain the **state of values** between **multiple pages** for a **single user**. **ASP.NET** provides us options for both **reading** and **writing** cookies on client machines.

Writing Cookies on client machine: we can write cookies on client machine in 2 different ways, those are:

1. Create the instance of **HttpCookie** class; store values into it as an array and then write that cookie to client machine by using **Response** object.

```
HttpCookie cookie = new HttpCookie("LoginCookie");
cookie["User"] = "Raju";
cookie["Pwd"] = "admin";
Response.Cookies.Add(cookie);
```

2. Write each value to browser as an individual cookie using Response object in a name/value combination.

```
Response.Cookies["User"].Value = "Raju";
Response.Cookies["Pwd"].Value = "admin";
```

Reading Cookies back on our WebPages: we can read Cookies present on the client machine in any WebPage of our application by using Request object.

1. If we write the cookie to browser by using the 1st approach, we need to read it as following:

```
HttpCookie cookie = Request.Cookies["LoginCookie"];
string User = cookie["User"];
string Pwd = cookie["Pwd"];
```

2. If we write the cookie to browser by using the 2nd approach, we need to read it as following:

```
string User = Request.Cookies["User"].Value;
string Pwd = Request.Cookies["Pwd"].Value;
```

Cookies are of 2 types:

- I. In-Memory Cookies
- II. Persistent Cookies

In-Memory cookies are stored in **browser's memory** so once the browser is closed, immediately all the cookies that are associated with that browser window will be destroyed, and by default every cookie is **In-Memory** only. Persistent Cookies are stored on **Hard Disk** of the client machines, so even after closing the browser window also they will be persisting and can be accessed next time we visit the site. To make a cookie as **persistent** we need to set "**Expires**" property of **Cookie** with a "**DateTime**" value.

Setting expires property of Cookie:

1. If we write the cookie to browser by using 1st approach, we need to set the expires property as following:

```
cookie.Expires = <DateTime>;
```

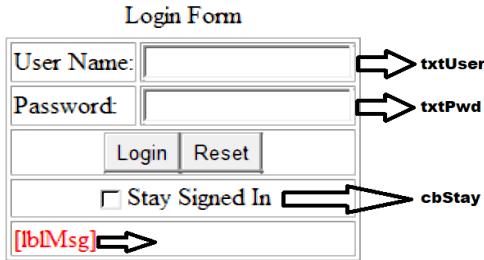
2. If we write the cookie to browser by using 2nd approach, we need to set the expires property as following:

```
Response.Cookies["user"].Expires = <DateTime>;
Response.Cookies["pwd"].Expires = <DateTime>;
```

To work with Cookies first add 3 new Web Forms under the project naming them as:

1. LoginWithStaySignedIn.aspx
2. SuccessWithStaySignedIn.aspx
3. FailureWithStaySignedIn.aspx

Design LoginWithStaySignedIn.aspx as below:



```


|                                                                                                                                         |                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| User Name:                                                                                                                              | <asp:textbox id="txtUser" runat="server"></asp:textbox>                    |
| Password:                                                                                                                               | <asp:textbox id="txtPwd" runat="server" textmode="Password"></asp:textbox> |
| <asp:button id="btnLogin" runat="server" text="Login"></asp:button> <asp:button id="btnReset" runat="server" text="Reset"></asp:button> |                                                                            |
| <input checked="checked" id="cbStay" name="cbStay" type="checkbox" value="true"/> Stay Signed In                                        |                                                                            |
| <asp:label forecolor="Red" id="lblMsg" runat="server"></asp:label>                                                                      |                                                                            |


```

[Now go to LoginWithStaySignedIn.aspx.cs file and write the below code:](#)

[Code under Page_Load Event Handler:](#)

```

if(Request.Cookies["LoginCookie"] != null) {
    Response.Redirect("SuccessWithStaySignedIn.aspx");
}
if(!IsPostBack) {
    txtUser.Focus();
    ViewState["FailureCount"] = 0;
}

```

[Code under Reset Button Click Event Handler:](#)

```

cbStay.Checked = false;
lblMsg.Text = txtUser.Text = txtPwd.Text = "";
txtUser.Focus();

```

Code under Login Button Click Event Handler:

```
if(txtUser.Text == "admin" && txtPwd.Text == "admin") {
    HttpCookie cookie = new HttpCookie("LoginCookie");
    cookie["User"] = txtUser.Text;
    cookie["Pwd"] = txtPwd.Text;
    if(cbStay.Checked) {
        cookie.Expires = DateTime.Now.AddDays(10);
    }
    Response.Cookies.Add(cookie);
    Response.Redirect("SuccessWithStaySignedIn.aspx");
}
else {
    int Count = (int)ViewState["FailureCount"] + 1;
    if(Count == 3) {
        Response.Cookies["User"].Value = txtUser.Text;
        Response.Cookies["Count"].Value = Count.ToString();
        Response.Redirect("FailureWithStaySignedIn.aspx");
    }
    ViewState["FailureCount"] = Count;
    lblMsg.Text = Count + " attempt(s) failed to login and maximum are 3 only.";
}
```

Now go to SuccessWithStaySignedIn.aspx.cs file and write the below code under Page_Load Event Handler:

```
if (Request.Cookies["LoginCookie"] != null) {
    HttpCookie cookie = new HttpCookie("LoginCookie");
    string Name = cookie["User"];
    string Pwd = cookie["Pwd"];
    Response.Write("Hello " + Name + ", welcome to the site.");
}
else {
    Response.Redirect("LoginWithStaySignedIn.aspx");
}
```

Now go to FailureWithStaySignedIn.aspx.cs file and write the below code under Page_Load Event Handler:

```
if (Request.Cookies["User"] != null && Request.Cookies["Count"] != null) {
    string User = Request.Cookies["User"].Value;
    string Count = Request.Cookies["Count"].Value;
    Response.Write($"Hello {User}, you have failed all the {Count} attempts to login.");
}
else {
    Response.Redirect("LoginWithStaySignedIn.aspx");
}
```

Now run “[Login Page](#)” and if we enter valid credentials, it will take you to “[SuccessPage](#)” and if we enter wrong credentials it will check for 3 times and will take you to “[Failure Page](#)”. When valid credentials are entered with “[StaySignedIn](#)” option checked, then it will not ask for credentials when we open the “[Login Page](#)” for next time and directly takes you to “[Success Page](#)” and in this case we can directly open “[Success Page](#)” from next time

whereas if the credentials or not supplied, then if we try to open “Success Page” or “Failure Page” it will redirect us back to “[Login Page](#)” because unless credentials are provided we can’t open any page of the site.

Drawbacks of Cookies:

1. We can create only **50 cookies** for each website, so every new cookie from the site will override the old cookie once after reaching the limit.
2. A cookie can store only **4 K.B.** of data that too of type **string** only.
3. Cookies are **not secured** because they are stored on client machines.
4. Because cookies are stored on client machines there is a problem like clients can either **delete the cookies** or even **disable cookies**.

Location of Cookies:

[Microsoft Edge](#): C:\Users\<User>\AppData\Local\Microsoft\Edge\User Data\Default

[Google Chrome](#): C:\Users\<User>\AppData\Local\Google\Chrome\User Data\Default

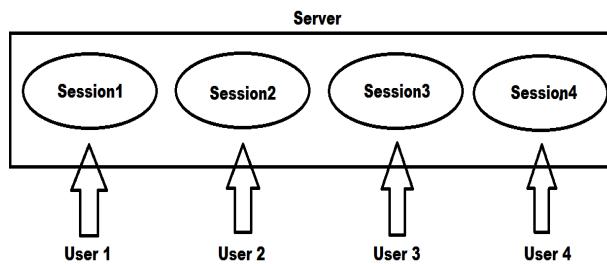
Disabling cookies on Brower: Click on “...” option beside the Address bar in browser => select settings and search for “Cookies” which shows “**Cookies and other site data**” option and use the appropriate option to disable cookies.

Delete Cookies on Browser: To delete cookies on our browser use the command **Ctrl + Shift + Delete** which will open “**Clear browsing data**” window, in that choose “**Cookies and other site data**” checkbox and click “**Clear Data**”.

Option 5: Session

In ASP.NET session is a state that is used to store and retrieve values of a user across all the pages of site i.e., it is “**Single User Global Data**”. It helps to identify requests from the same browser during a time (session). It is used to store value for the time. By default, **ASP.NET** session state is enabled for all **ASP.NET** applications. Each created session is stored in “**SessionStateItemCollection**” object. We can get current session value by using Page object’s Session property which is of type “**HttpSessionState**”.

Whenever a user sends his first request to server, server will provide a session for that user to store data that is associated with that user, giving the option of accessing that data across all the pages of site. **Session** doesn’t have any size limitation and more over they can store any type of data like **Scalar Types** and **Complex Types** also. **Sessions** are unique i.e., the session associated with one user is never accessible to other users.



Storing values into the Session:

Session[string key] = value (object)

Accessing values from Session:

object value = Session[string key]

To test Sessions, add 3 new Web Forms under the project naming them as:

1. PersonalDetails.aspx
2. FamilyDetails.aspx
3. DisplayDetails.aspx

Step 1: Design PersonalDetails.aspx as following:

Personal Details

First Name:	<input id="txtFName" style="width: 100%; height: 25px;" type="text"/>
Last Name:	<input id="txtLName" style="width: 100%; height: 25px;" type="text"/>
Email Id:	<input id="txtEmail" style="width: 100%; height: 25px;" type="text"/>
Phone No:	<input id="txtPhone" style="width: 100%; height: 25px;" type="text"/>
<input style="width: 150px; height: 30px; background-color: #ccc; border: 1px solid #ccc; font-weight: bold; color: #000;" type="button" value="Next Page"/>	

```
<table align="center">
    <caption>Personal Details</caption>
    <tr>
        <td>First Name:</td>
        <td><asp:TextBox ID="txtFName" runat="server" /></td>
    </tr>
    <tr>
        <td>Last Name:</td>
        <td><asp:TextBox ID="txtLName" runat="server" /></td>
    </tr>
    <tr>
        <td>Phone No:</td>
        <td><asp:TextBox ID="txtPhone" runat="server" /></td>
    </tr>
    <tr>
        <td>Email Id:</td>
        <td><asp:TextBox ID="txtEmail" runat="server" /></td>
    </tr>
    <tr>
        <td colspan="2" align="center">
            <asp:Button ID="btnNext" runat="server" Text="Next Page" />
        </td>
    </tr>
</table>
```

Now go to PersonalDetails.aspx.cs and write the below code:

Code under Page_Load Event Handler:

```
if (!IsPostBack) {
    txtFName.Focus();
}
```

Code under Next Page Button Click Event Handler:

```
Dictionary<string, object> userDetails = new Dictionary<string, object>();
userDetails.Add("First Name", txtFName.Text);
userDetails.Add("Last Name", txtLName.Text);
userDetails.Add("Phone No", txtPhone.Text);
userDetails.Add("EMail Id", txtEmail.Text);
Session["UserData"] = userDetails;
Response.Redirect("FamilyDetails.aspx");
```

Step 2: Design FamilyDetails.aspx as following:

Family Details

Spouse Name:	<input id="txtSName" type="text"/>
Father Name:	<input id="txtFName" type="text"/>
Mother Name:	<input id="txtMName" type="text"/>
Children if any:	<input id="txtChildren" type="text"/>
Next Page	

```
<table align="center">
  <caption>Family Details</caption>
  <tr>
    <td>Spouse Name:</td>
    <td><asp:TextBox ID="txtSName" runat="server" /></td>
  </tr>
  <tr>
    <td>Father Name:</td>
    <td><asp:TextBox ID="txtFName" runat="server" /></td>
  </tr>
  <tr>
    <td>Mother Name:</td>
    <td><asp:TextBox ID="txtMName" runat="server" /></td>
  </tr>
  <tr>
    <td>Children (if any):</td>
    <td><asp:TextBox ID="txtChildren" runat="server" /></td>
  </tr>
  <tr>
    <td colspan="2" align="center">
      <asp:Button ID="btnNext" runat="server" Text="Next Page" />
    </td>
  </tr>
</table>
```

```
</table>
```

Now go to FamilyDetails.aspx.cs file and write the below code in it:

Code under Page_Load Event Handler:

```
if (Session["UserDetails"] == null) {  
    Response.Redirect("PersonalDetails.aspx");  
}  
  
else {  
    txtSName.Focus();  
}
```

Code under Next Page Button Click Event Handler:

```
if (Session["UserData"] != null) {  
    Dictionary<string, object> userDetails = (Dictionary<string, object>)Session["UserData"];  
    userDetails.Add("Spouse Name", txtSName.Text);  
    userDetails.Add("Father Name", txtFName.Text);  
    userDetails.Add("Mother Name", txtMName.Text);  
    userDetails.Add("Children", txtChildren.Text);  
    Session["UserData"] = userDetails;  
    Response.Redirect("DisplayDetails.aspx");  
}  
  
else {  
    Response.Redirect("PersonalDetails.aspx");  
}
```

Step 3: Go to DisplayDetails.aspx.cs file and write the following code under Page_Load:

```
if (Session["UserData"] != null) {  
    Dictionary<string, object> userDetails = (Dictionary<string, object>)Session["UserData"];  
    foreach (string Key in userDetails.Keys) {  
        Response.Write(Key + ":" + userDetails[Key] + "<br />");  
    }  
}  
  
else {  
    Response.Redirect("PersonalDetails.aspx");  
}
```

Note: In the above example 1st we are storing data of "PersonalDetails" page in to a Dictionary and then putting that Dictionary into Session and redirecting to "FamilyDetails" page, and in that page we are picking the Dictionary from Session which is stored by "PersonalDetails" page, and we added "FamilyDetails" data also into that Dictionary and again putting that Dictionary into Session which is finally captured in "DisplayDetails" page to display them in the form of "name/value" pairs.

Run "PersonalDetails.aspx" page, fill in the details, and click on the "Next Page" button which will launch "FamilyDetails.aspx" and here also fill in the details and click on the "Next Page" button to launch "DisplayDetails.aspx", which will display all the values that are captured from the first 2 pages. Now copy the URL of the "DisplayDetails.aspx" page in Address Bar, open a new tab in the same browser window, paste the URL in Address Bar to execute, and here also we see the values that are captured in 1st and 2nd pages, whereas if we open a

new instance of a new browser and paste the URL it will not display the values but takes you to the "PersonalDetails.aspx" because Session values of 1 user are not shared with another user.

How is a session identified to which user it belongs to?

Ans: Whenever a Session is created for a user it is given with a Unique ID known as "Session Id" and this "Session Id" is written to client's browser in the form of a "In-Memory Cookie", so whenever the client comes back to the server in a next request, server will read the Cookie, picks "Session Id" and associates user with his exact Session.

Note: because "Session Id" is stored on client's browser in the form of an "In-Memory Cookie", other tabs under the browser instance can also access that Session, whereas a new instance of a new browser can't access the Session.

What happens to Sessions associated with clients if the client closes the browser?

Ans: Every Session will be having a "time-out" period of "20 Minutes (default)" from the last request (Sliding Expiration), so within 20 Minutes if the Session is not used by the User, Server will destroy that Session.

Note: we can change the default time-out period of "20 Mins" to our required value thru "Web.config" file by setting "timeout" attribute value of "sessionState" element as following:

```
<sessionState timeout="1" />
```

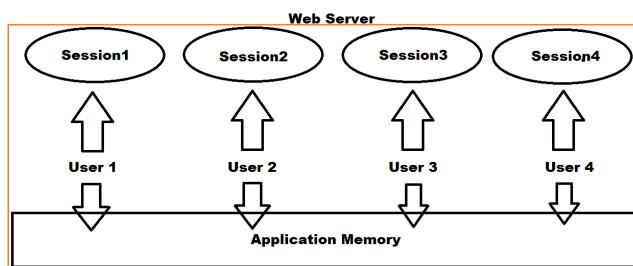
Can we explicitly destroy a Session associated with a User?

Ans: Yes, this can be performed by calling "Abandon()" method on the Session and this is what we generally do under "Sign Out" or "Log Out" options in a Web Site or Web Application.

E.g.: Session.Abandon();

Option 6: Application State:

Application-State is also a state management technique, and it is a global storage mechanism that is used to store data on the server which is shared between all users i.e., data stored in Application-State is accessible to all users and anywhere in the application (**Multi-User Global Data**). Application-State is stored in the memory of the Web Server and is faster than storing and retrieving information from a Database. Application-State is used in the same way as Session-State, but Session-State is specific for a single user, whereas Application-State is common for all users of the application.



Application-State does not have any default expiration period like Session-State, so when we recycle the worker process only the application object will be lost. Data is stored into Application-State in the form of name/value pairs only like we store in Session-State and ViewState. We store data into Application-State by using "Application" object of our parent class "Page" and that Application object is of type "HTTPApplicationState" class.

Storing values into Application-State:

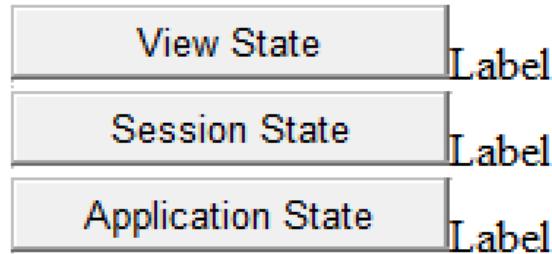
Application[string key] = value (object)

Accessing values from Application-State:

```
object value = Application[string key]
```

Note: Application Memory is not Thread-Safe, so to overcome the problem whenever we are dealing with Application-State data we need to call Lock() and UnLock() methods on Application object.

To compare the difference between "ViewState", "Session-State" and "Application-State" memory add a new Web Form under the project naming it as "CompareStates.aspx" and design it as following:



```
<h1 style="background-color:yellow;color:red;text-align:center;text-decoration:underline">Compare States</h1>
<asp:Button ID="Button1" runat="server" Text="View State" Width="200" />
<asp:Label ID="Label1" runat="server" ForeColor="Red" /><br />
<asp:Button ID="Button2" runat="server" Text="Session State" Width="200" />
<asp:Label ID="Label2" runat="server" ForeColor="Red" /><br />
<asp:Button ID="Button3" runat="server" Text="Application State" Width="200" />
<asp:Label ID="Label3" runat="server" ForeColor="Red" />
```

Now goto CompareStates.aspx.cs file and write the following code:

Code under View State Button:

```
int Count = 0;
if (ViewState["Counter"] == null)
{
    Count += 1;
}
else
{
    Count = (int)ViewState["Counter"] + 1;
}
ViewState["Counter"] = Count;
Label1.Text = "View State: " + Count;
```

Code under Session State Button:

```
int Count = 0;
if (Session["Counter"] == null)
{
    Count += 1;
}
else
{
    Count = (int)Session["Counter"] + 1;
}
```

```

}

Session["Counter"] = Count;
Label2.Text = "Session State: " + Count;
Code under Application State Button:
int Count = 0;
Application.Lock();
if (Application["Counter"] == null) {
    Count += 1;
}
else {
    Count = (int)Application["Counter"] + 1;
}
Application["Counter"] = Count;
Application.UnLock();
Label3.Text = "Application State: " + Count;

```

Now run the above **page** and click on the **3 buttons** to increment their count to **5**, then copy the **URL** of **page**, open a **new tab**, and call the **page** by using copied **URL**. When we click on **“ViewState”** button value will be **1** because it is **“Single User Local Data”**. When we click on **“Session State”** button the value will be **6** because it is **“Single User Global Data”** and the **“Session-Id”** is accessible between **tabs** of the **browser**, then open another **instance** of a new **browser**, open the page using copied **URL** and, in this case, **ViewState** value will **1**, **Session** value also will be **1** because **“Session-Id”** of the old browser is not carried to new **instance** of new **browser** but application value will be **“old + 1”** as this is **“Multi User Global Data”**.

Global.asax:

This file is also known as the **ASP.NET** application file, is an optional file that contains code for responding to **“Application-Level”** and **“Session-Level”** events raised by **ASP.NET**. The file contains a class with the name **“Global”** that extends the **HttpApplication** class. There can be only one **“Global.asax”** file per application and it should be in the application's root directory only. Every **ASP.Net Web Application** project contains this file by default. **Global** class contains methods for handling **Application & Session** events like:

1. Application_Start
2. Application_End
3. Application_Error
4. Session_Start
5. Session_End

Application_Start: This method is invoked when the application **first starts** i.e., whenever the **first request** comes to the application and executes **1 and only 1** time in the **life cycle** of an application. This **event handler** is a useful place to provide **application-wide** initialization code.

Application_End: This method is invoked just before an **application ends**. The end of an application can occurs because **IIS** is being **restarted** or because the application is transitioning to a new application domain in response to updated files or the **worker process** recycling and it typically contains application **cleanup** logic.

Application_Error: This method is invoked whenever an **unhandled exception** occurs in the application, and we implement all the **error handling code** in this.

Session_Start: This method is invoked each time a new session begins i.e., when the first request comes from a user. This is often used to initialize user-specific information.

Session_End: This method is invoked whenever the user's session ends. A session ends when your code explicitly releases it or when its time is out after there have been no more requests received within a given timeout period (typically 20 minutes) and this contains logic for cleanup of user specific data.

Note: Global.asax file is never called directly by the user, rather they are called automatically in response to Application and Session events. When Global.asax files changes, the framework reboots the application and the Application_Start event is fired once again when the next request comes in. Note that the Global.asax file does not need recompilation if no changes have been made to it.

A program to find out the “Total No. of Users” and “Total No. of Online Users” for a site by using "Global.asax":

Step 1: Open Global.asax file and write the below code under the class Global.

Code under Application_Start method:

```
Application["TNU"] = 0;  
Application["TOU"] = 0;
```

Code under Session_Start method:

```
Application.Lock();  
Application["TNU"] = (int)Application["TNU"] + 1;  
Application["TOU"] = (int)Application["TOU"] + 1;  
Application.UnLock();
```

Code under Session_End method:

```
Application.Lock();  
Application["TOU"] = (int)Application["TOU"] - 1;  
Application.UnLock();
```

Step 2: Add a new Web Form under the project, naming it as “UsersCount.aspx” and write the below code under its <div> tag.

```
Total Users: <asp:Label ID="Label1" runat="server" ForeColor="Red" /><br />  
Online Users: <asp:Label ID="Label2" runat="server" ForeColor="Red" /><br />  
<asp:Button ID="btnSignOut" runat="server" Text="Sign-Out" />
```

Step 3: Now goto UsersCount.aspx.cs file and write the below code.

Code under Page_Load:

```
Application.Lock();  
Label1.Text = Application["TNU"].ToString();  
Label2.Text = Application["TOU"].ToString();  
Application.UnLock();
```

Under Sign-Out Button:

```
Session.Abandon();
```

Run the page for multiple times and notice both “**TNU**” and “**TOU**” will increment and if at all we click on **Sign-Out** button then we notice “**TOU**” value getting decremented.

ADO.NET

Pretty much every application deal with data in some manner, whether that data comes from memory, databases, XML files, text files, or something else. The location where we store the data can be called as a Data Source or Data Store where a Data Source can be a file, database, address books or indexing server etc.

Programming Languages cannot communicate with Data Sources directly because each Data Source adopts a different Protocol (set of rules) for communication, so to overcome this problem long back Microsoft has introduced intermediate technologies like ODBC and OleDb which works like bridge between the Applications and Data Sources to communicate with each other.

ODBC (Open Database Connectivity) is a standard C programming language middleware API for accessing database management systems (DBMS). ODBC accomplishes DBMS independence by using an ODBC driver as a translation layer between the application and the DBMS. The application uses ODBC functions through an ODBC driver manager with which it is linked, and the driver passes the query to the DBMS. An ODBC driver will be providing a standard set of functions for the application to use and implementing DBMS-specific functionality. An application that can use ODBC is referred to as "ODBC-Compliant". Any ODBC-Compliant application can access any DBMS for which a driver is installed. Drivers exist for all major DBMS's as well as for many other data sources like Microsoft Excel, and even for Text or CSV files. ODBC was originally developed by Microsoft in 1992.

1. It's a collection of drivers, where these drivers sit between the App's and Data Source's to communicate with each other and more over we require a separate driver for every data source.
2. ODBC drivers comes along with your Windows O.S. and we can find them at the following location:
Control Panel => Administrative Tools => ODBC Data Sources
3. To consume these ODBC Drivers first we need to configure them with the data source by creating a "DSN" (Data Source Name).
4. ODBC drivers are open source i.e., there is an availability of these ODBC Drivers for all the leading Operation System's in the market.

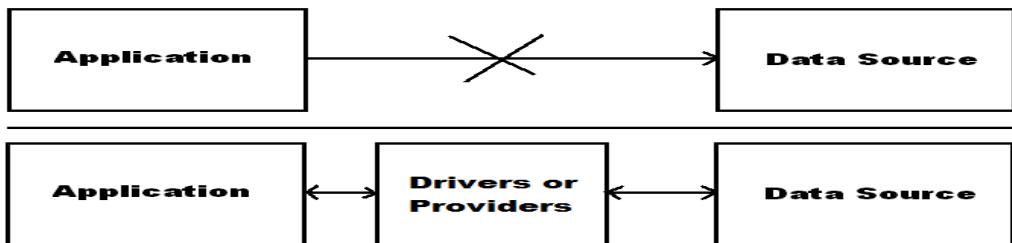
Drawbacks with ODBC Drivers:

1. These drivers must be installed on every machine where the application is executing from and then the application, driver and data source should be manually configured with each other.
2. ODBC Drivers are initially designed for communication with Relational DB only.

OLE DB (Object Linking and Embedding, Database, sometimes written as OLEDB or OLE-DB), an API designed by Microsoft, allows accessing data from a variety of data sources in a uniform manner. The API provides a set of interfaces implemented using the Component Object Model (COM) and SQL. Microsoft originally intended OLE DB as a higher-level replacement for, and successor to, ODBC, extending its feature set to support a wider variety of non-relational databases, such as object databases and spreadsheets that do not necessarily implement SQL. OLE DB is conceptually divided into consumers and providers. The consumers are the applications that need access to the data, and the providers are the software components that implement the interface and thereby provide the data to the consumer. An OLE DB provider is a software component enabling an OLE DB consumer to interact with a data source. OLE DB providers are alike to ODBC drivers. OLE DB providers can be created to access

such simple data stores as a text file and spreadsheet, through to such complex databases as Oracle, Microsoft SQL Server, and many others. It can also provide access to hierarchical data stores. These OLE DB Providers are introduced by Microsoft around the year 1996.

1. It's a collection of providers where these providers sit between the Applications and Data Source to communicate with each other, and we require a separate provider for each data source.
2. OleDb Providers are designed for communication with relational & non-relational data source also i.e., it provides support for communication with any Data Source.
3. OleDb Providers sits on server machine so they are already configured with data source and when we connect with any data source they will help in the process of communication.
4. OleDb Providers are developed by using COM and SQL Languages, so they are also un-managed.
5. Microsoft introduced OLEDB as a replacement for ODBC for its Windows Systems.
6. OleDb is a pure Microsoft technology which works only on Windows Platform.

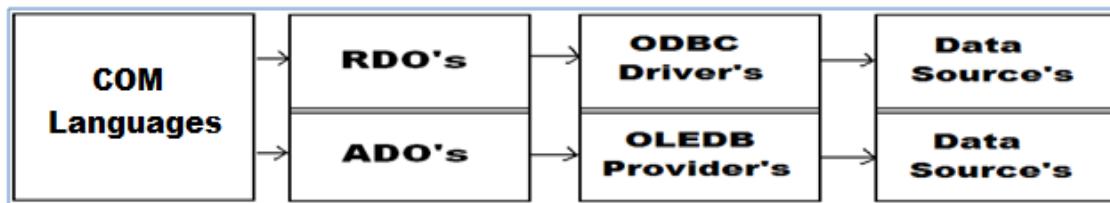


Things to remember while working with Odbc and OleDb:

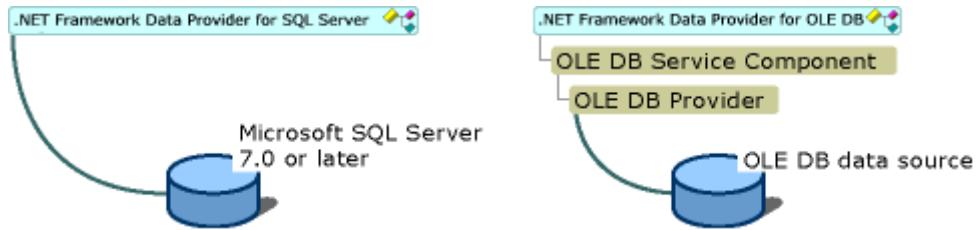
1. ODBC and OleDb are un-managed or platform dependent.
2. ODBC and OleDb are not designed targeting any particular language i.e., they can be consumed by any language like: C, CPP, Visual Basic, Visual CPP, Java, C# etc.

Note: If any language wants to consume ODBC Drivers or OleDb Providers they must use some built-in libraries of the language in which we are developing the application without writing complex coding.

RDO's and ADO's in COM Language: COM Language used RDO's ([Remote Data Objects](#)) and ADO's ([ActiveX Data Objects](#)) for data source communication without having to deal with the comparatively complex ODBC or OLEDB API.



.NET Framework Providers: The .NET Framework Data Provider for SQL Server uses its own protocol to communicate with SQL Server. It is lightweight and performs well because it is optimized to access a SQL Server directly without adding an OLE DB or ODBC layer and it supports SQL Server software version 7.0 or later. The .NET Framework Data Provider for Oracle (Oracle Client) enables data access to Oracle data sources through Oracle client connectivity software. The data provider supports Oracle client software version 8.1.7 or later.



ADO.Net: It is a set of types that expose data access services to the .NET programmer. ADO.NET provides functionality to developers writing managed code like the functionality provided to native COM developers by ADO. ADO.NET provides consistent access to data sources such as Microsoft SQL Server, as well as data sources exposed through OLE DB and XML. Data-sharing consumer applications can use ADO.NET to connect to these data sources and retrieve, manipulate, and update data. It is an integral part of the .NET Framework, providing access to relational data, XML, and application data. ADO.NET supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects used by applications or Internet browsers.

ADO.Net provides libraries for Data Source communication under the following namespaces:

- System.Data
- System.Data.Odbc
- System.Data.Oledb
- System.Data.SqlClient
- System.Data.OracleClient

Note: System.Data, System.Data.Odbc, System.Data.Oledb and System.Data.SqlClient namespaces are under the assembly System.Data.dll whereas System.Data.OracleClient is under System.Data.OracleClient.dll assembly.

System.Data: types of this namespace are used for holding and managing of data on client machines. This namespace contains following set of classes in it: **DataSet**, **DataTable**, **DataRow**, **DataColumn**, **DataView** and **DataRelation**.

System.Data.Odbc: types of this namespace can communicate with any **Relational Data Source** using **Un-Managed ODBC Drivers**.

System.Data.OleDb: types of this namespace can communicate with any **Data Source** using **OleDb Providers** (**Un-Managed COM Providers**).

System.Data.SqlClient: types of this namespace can purely communicate with **SQL Server Database** only using **SqlClient Provider** (**Managed .Net Framework Provider**).

System.Data.OracleClient: types of this namespace can purely communicate with **Oracle Database** only using **OracleClient Provider** (**Managed .Net Framework Provider**).

All the above 4 namespaces contain same set of types as following: **Connection**, **Command**, **DataReader**, **DataAdapter**, **Parameter** and **CommandBuilder**, but here each class is referred by prefixing with **Odbc**, **OleDb**, **Sql** and **Oracle** keywords before the class name to **discriminate** between each other as following:

OdbcConnection	OdbcCommand	OdbcDataReader	OdbcDataAdapter	OdbcCommandBuilder	OdbcParameter
----------------	-------------	----------------	-----------------	--------------------	---------------

OleDbConnection	OleDbCommand	OleDbDataReader	OleDbDataAdapter	OleDbCommandBuilder	OleDbParameter
SqlConnection	SqlCommand	SqlDataReader	SqlDataAdapter	SqlCommandBuilder	SqlParameter
OracleConnection	OracleCommand	OracleDataReader	OracleDataAdapter	OracleCommandBuilder	OracleParameter

Performing Operations on a DataSource: the operations we perform on a **Data Source** will be **Select, Insert, Update** and **Delete**, and every operation we perform on a **Data Source** involves in **3 steps**, like:

- Establishing a connection with Data Source.
- Sending a request to Data Source by using SQL.
- Capturing the results given by Data Source.

Establishing a Connection with Data Source: It's a process of opening a **channel for communication** between **Application** and **Data Source** that is present either on a **local** or **remote** machine to perform **Database** operations and to open the channel for communication we use **Connection** class.

Working with Connection class: To work with any class first we need to know the members of that class like Constructors, Properties, Methods, etc.

Constructors of the Class:

- Connection()
- Connection(string ConnectionString)

Note: **Connection String** is a collection of attributes that are required for **connecting** with a **Data Source**, those are:

- DSN
- Provider
- Data Source
- User Id and Password
- Integrated Security
- Database or Initial Catalog
- Extended Properties

DSN: this is the only attribute that is required if we want to connect with a data source by using ODBC Drivers and by using this attribute, we need to specify the DSN Name.

Provider: this attribute is required when we want to connect to the data source by using OleDb Providers. So, by using this attribute we need to specify the provider's name based on the data source we want to connect with.

Oracle: Msdaora or ORAOLEDB.ORACLE

SQL Server: SqlOledb

[MS-Access or MS-Excel:](#) Microsoft.Jet. Oledb.4.0 (32 Bit OS) Microsoft.Ace.Oledb.12.0 (64 Bit OS)

[MS-Indexing Server:](#) Msidxs

Data Source: this attribute is required to specify the server's name if the Data Source is a Database or else if the Data Source is a File, we need to specify path of the file and this attribute is required in case of any provider communication.

User Id and Password: This attribute is required to specify the credentials for connection with a database and this attribute is required in case of any provider communication.

Integrated Security: this attribute is used while connecting with [SQL Server Database only](#) to specify that we want to connect with the Server by using Windows Authentication and in this case, we should not use User Id and Password attributes and this attribute is required in case of any provider communication.

Database or Initial Catalog: these attributes are used while connecting with [Sql Server Database only](#) to specify the name of DB we want to connect with, and this attribute is required in case of any provider communication.

Extended Properties: this attribute is required only while connecting with [MS-Excel](#) using OleDb Provider.

List of attributes which are required in case of Odbc Drivers, Oledb and Framework Providers

<u>Attribute</u>	<u>ODBC Driver</u>	<u>OLEDB Provider</u>	<u>Framework Provider</u>
DSN	Yes	No	No
Provider	No	Yes	No
Data Source	No	Yes	Yes
User Id and Password	No	Yes	Yes
Integrated Security*	No	Yes	Yes
Database or Initial Catalog*	No	Yes	Yes
Extended Properties**	No	Yes	-

*Only for SQL Server

**Only for Microsoft Excel

Connection String for SQL Server to connect by using different options:

```
OdbcConnection con = new OdbcConnection("Dsn=<Dsn Name>");  
OleDbConnection con = new OleDbConnection("Provider=SqlOledb;Data Source=<Server Name>;  
                                         Database=<DB Name>;User Id=<User Name>;Password=<Pwd>");  
SqlConnection con = new SqlConnection("Data Source=<Server Name>;Database=<DB Name>;  
                                         User Id=<User Name>;Password=<Pwd>");
```

Note: in case of [Windows Authentication](#) in place of [User Id](#) and [Password](#) attributes we need to use [Integrated Security = SSPI](#) (Security Support Provider Interface).

Connection String for Oracle to connect by using different options:

```
OdbcConnection con = new OdbcConnection("Dsn=<Dsn Name>");  
OleDbConnection con = new OleDbConnection("Provider=Msdaora (o)r ORAOLEDB.ORACLE;  
                                         Data Source=<Server Name>;User Id=<User Name>;Password=<Pwd>");  
OracleConnection con = new OracleConnection("Data Source=<Server Name>;  
                                         User Id=<User Name>;Password=<Pwd>");
```

Connection String for MS-Excel to connect by using different options:

```
OdbcConnection con = new OdbcConnection("Dsn=<Dsn Name>");
```

```
OleDbConnection con = new OleDbConnection("Provider=Microsoft.Jet.Oledb.4.0;  
Data Source=<Path of Excel Document>;Extended Properties=Excel 8.0");
```

Members of Connection class:

1. **Open():** a method which opens a connection with data source.
2. **Close():** a method which closes the connection that is open.
3. **State:** an enumerated property which is used to get the status of connection.
4. **ConnectionString:** a property which is used to get or set a connection string that is associated with the connection object.

Object of class Connection can be created in any of the following ways:

```
Connnection con = new Connection();  
con.ConnectionString = "<connection string>";
```

or

```
Connection con = new Connection("<connection string>");
```

Testing the process of establishing a connection: create a new “ASP.Net Web Application” project, naming it as “DBExamples”, select “Empty Project Template”, check the “Web Forms” CheckBox and click on the “Create” button.

Add a WebForm in the project, name it as “TestConnection.aspx” and write the below code in its <div> tag:

```
<asp:Button ID="Button1" runat="server" Text="Connect with Sql Server by using Odbc Driver" /><br />  
<asp:Button ID="Button2" runat="server" Text="Connect with Sql Server by using Oledb Provider" /><br />  
<asp:Button ID="Button3" runat="server" Text="Connect with Sql Server by using SqlClient Provider" /><br />  
<asp:Button ID="Button4" runat="server" Text="Connect with Microsoft Excel by using Odbc Driver" /><br />  
<asp:Button ID="Button5" runat="server" Text="Connect with Microsoft Excel by using Oledb Provider" />
```

Create 2 DSN's for connecting with “SQL Server” and “Excel” Data Sources, and to do that go to Control Panel => Administrative Tools => click on “ODBC Data Sources” which opens a new window and configure the “DSN” as following => click on “Add” button => choose a driver for “SQL Server” => click “Finish” button => Enter “Name” as: “SqlDsn”, “Description” as: “Connects with SQL Server DB”, “Server” as: “<Your Server Name>”, click “Next” button, choose the Authentication Mode as “Windows” or “SQL Server” and if it is “SQL Server” then enter “User Id” and “Password” below, click “Next” for choosing the “Database”, default will be “Master”; leave the same, click “Next” button and click “Finish” button which creates a DSN for “SQL Server” Database. Same as this, again click on “Add” button => choose a driver for “Microsoft Excel” => click “Finish” button => Enter “Name” as: “ExcelDsn”, “Description” as: “Connects with Microsoft Excel”, click on “Select Workbook” button, choose the “Excel Document” from its physical location and click on “Ok” button which creates a new DSN for Excel.

Write the below code under “TestConnection.aspx.cs” file:

```
using System.Data.Odbc;  
using System.Data.OleDb;  
using System.Data.SqlClient;
```

Code under Button1 Click Event:

```
OdbcConnection con = new OdbcConnection("DSN=SqlDsn");  
con.Open();  
Response.Write($"<script>alert('Connection is {con.State} with SQL Server using Odbc Driver.')</script>");  
con.Close();  
Response.Write($"<script>alert('Connection is {con.State} with SQL Server using Odbc Driver.')</script>");
```

Code under Button2 Click Event:

```
OleDbConnection con = new OleDbConnection();
//con.ConnectionString = "Provider=SqlOledb;Data Source=Server;Database=Master;User Id=Sa;Password=123";
con.ConnectionString = "Provider=SqlOledb;Data Source=Server;Database=Master;Integrated Security=SSPI";
con.Open();
Response.Write($"<script>alert('Connection is {con.State} with Sql Server using OleDb Provider.')</script>");
con.Close();
Response.Write($"<script>alert('Connection is {con.State} with Sql Server using OleDb Provider.')</script>");
```

Code under Button3 Click Event:

```
SqlConnection con = new SqlConnection();
//con.ConnectionString = "Data Source=Server;Database=Master;User Id=Sa;Password=123";
con.ConnectionString = "Data Source=Server;Database=Master;Integrated Security=SSPI";
con.Open();
Response.Write($"<script>alert('Connection is {con.State} with Sql Server using Framework Provider.') </script>");
con.Close();
Response.Write($"<script>alert('Connection is {con.State} with Sql Server using Framework Provider.') </script>");
```

Code under Button4 Click Event:

```
OdbcConnection con = new OdbcConnection("DSN=ExcelDsn");
con.Open();
Response.Write($"<script>alert('Connection is {con.State} with MS Excel using Odbc Driver.')</script>");
con.Close();
Response.Write($"<script>alert('Connection is {con.State} with MS Excel using Odbc Driver.')</script>");
```

Code under Button5 Click Event:

```
OleDbConnection con = new OleDbConnection("Provider=Microsoft.Jet.Oledb.4.0; Extended Properties=Excel
8.0;Data Source=C:\\ExcelDocs\\School.xls");
con.Open();
Response.Write($"<script>alert('Connection is {con.State} with MS Excel using OleDb Provider.')</script>");
con.Close();
Response.Write($"<script>alert('Connection is {con.State} with MS Excel using OleDb Provider.')</script>");
```

Sending a request to Data Source by using SQL: in this process we send **SQL Statement's** like **Select, Insert, Update** and **Delete** to the **Database** for execution or call **Stored Procedures** in **Database** for execution, and to do that we need to use the **Command** class.

Constructors of the class:

1. Command()
2. Command(string CommandText, Connection con)
 - CommandText means it can be any SQL Stmt like Select or Insert or Update or Delete or SP Name.
 - Connection means instance of the Connection class which we have created in our 1st step.

Properties of Command Class:

1. **Connection:** sets or gets the connection object associated with command object.
2. **CommandText:** sets or gets the Sql Statement or SP Name associated with command object.
3. **CommandType:** gets or sets whether command has to execute a Sql Statement [d] or Stored Procedure.

The object of class Command can be created in any of the following ways:

```
Command cmd = new Command();
cmd.Connection = <con>;
cmd.CommandText = "<Sql Stmt or SP Name>";
```

Or

```
Command cmd = new Command("<Sql Stmt or SP Name>", <con>);
```

Note: if calling a Stored Procedure then add the below statement also, in both the cases:

```
cmd.CommandType = CommandType.StoredProcedure;
```

Methods of Command class:

1. ExecuteReader() => DataReader
2. ExecuteScalar() => object
3. ExecuteNonQuery() => int

Note: after creating object of Command class we need to call any of the 3 execute methods to execute the stmt's.

- ❑ Use **ExecuteReader()** method when we want to execute a Select Statement that returns data as rows and columns. The method returns an instance of class DataReader which holds the data that is retrieved from Data Source in the form of rows and columns.
- ❑ Use **ExecuteScalar()** method when we want to execute a Select Statement that returns a single value result. The method returns result of the query in the form of an object.
- ❑ Use **ExecuteNonQuery()** method when we want to execute an SQL statement other than select, like Insert or Update or Delete etc. The method returns an integer which tells the no. of rows affected by the Stmt.

Note: The above process of calling a suitable method to capture the results is our third step i.e., capturing the results sent back by the Data Source.

Accessing data from a DataReader: DataReader is a class which can hold data in the form of **rows and columns**, and to access data from **DataReader** it provides the following members:

1. **GetName(int ColumnIndex)** => string

This method returns name of the column for given index position.

2. **Read()** => bool

This method moves the record pointer from the current location to next row & returns a Boolean value which tells whether the row to which it moved contains data or not, which will be true if data is present or false if not present.

3. **FieldCount** => int

This is a read-only property which returns the no. of columns DataReader contains or retrieved from the table.

4. **NextResult()** => bool

This method moves the record pointer from the current table to next table & returns a boolean value which tells whether the location to which it moved contains a table or not, which will be true if present or false if not present.

5. **GetValue(int ColoumnIndex)** => object

This method is used for retrieving column values from the row to which pointer was pointing by specifying the column index position. We can also access the row pointed by pointer by using an **Indexer** defined in the class either by specifying column index position or name, as following:

<code><DataReader>[int ColumnIndex]</code>	<code>=></code>	object
<code><DataReader>[string ColumnName]</code>	<code>=></code>	object

To test the examples in this document first download the file “[Northwind.sql](#)” to create **Northwind Database** from your “[Google Class Room](#)” and run the **SQL (Script)** file and to do that double click on the “.sql” file which will prompt for **Credentials** to open under “[SQL Server Management Studio](#)”. After it got opened, execute the code which will create the **Northwind Database** on our **SQL Server**.

Add a **Web Form** in the project naming it as “[Northwind_Customers_Select.aspx](#)” and write the below code in “[Northwind_Customers_Select.aspx.cs](#)” file:

```
using System.Text;
using System.Data.SqlClient;
```

Code under Page Load Event:

```
SqlConnection con = new SqlConnection("Data Source=Server;User Id=Sa;Password=123;Database=Northwind");
SqlCommand cmd = new SqlCommand("Select CustomerID, CompanyName, ContactName, City, Country, Phone,
PostalCode From Customers", con);
con.Open();
SqlDataReader dr = cmd.ExecuteReader();
StringBuilder sb = new StringBuilder();
sb.Append("<table border='1' align='center'>");
sb.Append("<caption>Customer Details</caption>");
sb.Append("<tr>");
for (int i = 0; i < dr.FieldCount; i++) {
    sb.Append("<th>" + dr.GetName(i) + "</th>");
}
sb.Append("</tr>");
while (dr.Read()) {
    sb.Append("<tr>");
    for (int i = 0; i < dr.FieldCount; i++) {
        sb.Append("<td>" + dr[i] + "</td>");
    }
    sb.Append("</tr>");
}
sb.Append("</table>");
con.Close();
Response.Write(sb);
```

Creating our own database on SQL Server to work with:

- Open Sql Server Management Studio and Create a new Database in it with the name “[ASPDB](#)”.
- Now under the DB create 2 new tables with the names Department and Employee as following:

```
Create Table Department (Did Int Constraint Did_PK Primary Key Identity(10, 10), Dname Varchar(50),
Location Varchar(50));
Create Table Employee (Eid Int Constraint Eid_PK Primary Key Identity(101, 1), Ename Varchar(50), Job
Varchar(50), Salary Money, Did Int Constraint Did_FK References Department(Did));
```

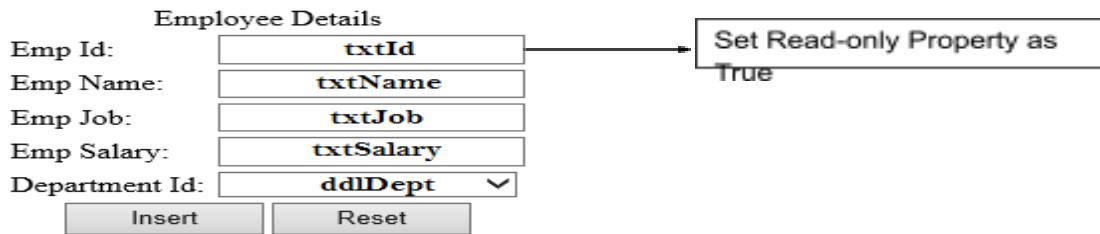
- Now insert 4 records into the Department table as following:

```

Insert Into Department Values ('Marketing', 'Mumbai');
Insert Into Department Values ('Sales', 'Chennai');
Insert Into Department Values ('Accounting', 'Hyderabad');
Insert Into Department Values ('Finance', 'Delhi');

```

Add a new WebForm in the project naming it as “ASPDB_Employee_Insert.aspx” and design it as following:



Html code for creating the above form which should be implemented under <div> tag:

```

<table align="center">
    <caption>Employee Details</caption>
    <tr>
        <td>Emp Id:</td>
        <td><asp:TextBox ID="txtId" runat="server" ReadOnly="true" /></td>
    </tr>
    <tr>
        <td>Emp Name:</td>
        <td><asp:TextBox ID="txtName" runat="server" /></td>
    </tr>
    <tr>
        <td>Emp Job:</td>
        <td><asp:TextBox ID="txtJob" runat="server" /></td>
    </tr>
    <tr>
        <td>Emp Salary:</td>
        <td><asp:TextBox ID="txtSalary" runat="server" /></td>
    </tr>
    <tr>
        <td>Department Id:</td>
        <td><asp:DropDownList ID="ddlDept" runat="server" Width="169px" /></td>
    </tr>
    <tr>
        <td colspan="2" align="center">
            <asp:Button ID="btnInsert" runat="server" Text="Insert" Width="60" />
            <asp:Button ID="btnReset" runat="server" Text="Reset" Width="60" />
        </td>
    </tr>
</table>

```

C# code which should be implemented in “ASPDB_Employee_Insert.aspx.cs” file:

```
using System.Data.SqlClient;
```

Declarations:

```
SqlCommand cmd;
```

```
SqlConnection con;
```

Code under Page Load Event:

```
con = new SqlConnection("Data Source=Server;User Id=Sa;Password=123;Database=ASPDB");
cmd = new SqlCommand();
cmd.Connection = con;
if(!IsPostBack) {
    LoadDept();
    txtName.Focus();
}
```

```
private void LoadDept() {
    cmd.CommandText = "Select Did, Dname from Department Order By Did";
    con.Open();
    SqlDataReader dr = cmd.ExecuteReader();
    ddlDept.DataSource = dr;
    ddlDept.DataTextField = "Dname";
    ddlDept.DataValueField = "Did";
    ddlDept.DataBind();
    ddlDept.Items.Insert(0, "-Select Department-");
    con.Close();
}
```

Code under Insert Button Click Event:

```
if (ddlDept.SelectedIndex > 0) {
    cmd.CommandText =
        $"Insert Into Employee Values('{txtName.Text}', '{txtJob.Text}', {txtSalary.Text}, {ddlDept.SelectedValue})";
    con.Open();
    if(cmd.ExecuteNonQuery() > 0) {
        cmd.CommandText = "Select Max(Eid) From Employee";
        txtId.Text = cmd.ExecuteScalar().ToString();
    }
    else {
        Response.Write("<script>alert('Failed inserting record into the table.')</script>");
    }
    con.Close();
}
else {
    Response.Write("<script>alert('Please choose a department to insert.')</script>");
}
```

Code under Reset Button Click Event:

```
txtId.Text = txtName.Text = txtJob.Text = txtSalary.Text = "";
ddlDept.SelectedIndex = 0;
txtName.Focus();
```

Add WebForm in the project naming it as “ASPDB_Employee_SelectUpdateDelete.aspx” and design it as below:

Employee Details

Emp Id:	<input style="width: 100%;" type="text" value="ddlEmp"/>		
Emp Name:	<input style="width: 100%;" type="text" value="txtName"/>		
Emp Job:	<input style="width: 100%;" type="text" value="txtJob"/>		
Emp Salary:	<input style="width: 100%;" type="text" value="txtSalary"/>		
Department Id:	<input style="width: 100%;" type="text" value="ddlDept"/>		
<input style="width: 100px; height: 25px; margin-right: 10px; border-radius: 5px; background-color: #ccc; border: none; font-size: 10pt; font-weight: bold; color: black;" type="button" value="Update"/> <input style="width: 100px; height: 25px; border-radius: 5px; background-color: #ccc; border: none; font-size: 10pt; font-weight: bold; color: black;" type="button" value="Delete"/>		Set AutoPostBack Property as True	

Html code for creating the above form which should be under <div> tag:

```
<table align="center">
  <caption>Employee Details</caption>
  <tr>
    <td>Emp Id:</td>
    <td><asp:DropDownList ID="ddlEmp" runat="server" Width="169px" AutoPostBack="true" /></td>
  </tr>
  <tr>
    <td>Emp Name:</td>
    <td><asp:TextBox ID="txtName" runat="server" /></td>
  </tr>
  <tr>
    <td>Emp Job:</td>
    <td><asp:TextBox ID="txtJob" runat="server" /></td>
  </tr>
  <tr>
    <td>Emp Salary:</td>
    <td><asp:TextBox ID="txtSalary" runat="server" /></td>
  </tr>
  <tr>
    <td>Department Id:</td>
    <td><asp:DropDownList ID="ddlDept" runat="server" Width="169px" /></td>
  </tr>
  <tr>
    <td colspan="2" align="center">
      <asp:Button ID="btnUpdate" runat="server" Text="Update" Width="60" />
      <asp:Button ID="btnDelete" runat="server" Text="Delete" Width="60" />
    </td>
  </tr>
</table>
```

C# code which should be implemented in “ASPDB_Employee_SelectUpdateDelete.aspx.cs” file:

```
using System.Data;
using System.Data.SqlClient;
```

Declarations:

```
SqlDataReader dr;
```

```

SqlCommand cmd;
SqlConnection con;


---


Code under Page Load Event:
con = new SqlConnection("Data Source=Server;User Id=Sa;Password=123;Database=ASPDB");
cmd = new SqlCommand();
cmd.Connection = con;
if(!IsPostBack) {
    LoadEmp();
    LoadDept();
    ddlEmp.Focus();
}


---


private void LoadEmp() {
    cmd.CommandText = "Select Eid from Employee Order By Eid";
    if (con.State != ConnectionState.Open) {
        con.Open();
    }
    dr = cmd.ExecuteReader();
    ddlEmp.DataSource = dr;
    ddlEmp.DataTextField = "Eid";
    ddlEmp.DataValueField = "Eid";
    ddlEmp.DataBind();
    ddlEmp.Items.Insert(0, "-Select Employee-");
    con.Close();
}


---


private void LoadDept()
{
    cmd.CommandText = "Select Did, Dname from Department Order By Did";
    con.Open();
    dr = cmd.ExecuteReader();
    ddlDept.DataSource = dr;
    ddlDept.DataTextField = "Dname";
    ddlDept.DataValueField = "Did";
    ddlDept.DataBind();
    ddlDept.Items.Insert(0, "-Select Department-");
    con.Close();
}

```

Code under DropDownList SelectedIndexChanged Event:

```

if(ddlEmp.SelectedIndex > 0) {
    cmd.CommandText = "Select Ename, Job, Salary, Did From Employee Where Eid=" + ddlEmp.SelectedValue;
    con.Open();
    dr = cmd.ExecuteReader();
    if(dr.Read()) {
        txtName.Text = dr["Ename"].ToString();
        txtJob.Text = dr["Job"].ToString();
    }
}

```

```

txtSalary.Text = dr["Salary"].ToString();
ddlDept.SelectedValue = dr["Did"].ToString();
con.Close();
}
else {
    Response.Write("<script>alert('Selected employee-id is not existing in the database.')</script>");
    txtName.Text = txtJob.Text = txtSalary.Text = "";
    ddlDept.SelectedIndex = 0;
    dr.Close();
    LoadEmp();
}
}
else {
    txtName.Text = txtJob.Text = txtSalary.Text = "";
    ddlDept.SelectedIndex = 0;
    ddlEmp.Focus();
}

```

Code under Update Button Click Event:

```

if (ddlEmp.SelectedIndex > 0) {
    if (ddlDept.SelectedIndex > 0) {
        cmd.CommandText = $"Update Employee Set Ename='{txtName.Text}', Job='{txtJob.Text}', Salary={txtSalary.Text}, Did={ddlDept.SelectedValue} Where Eid={ddlEmp.SelectedValue}";
        con.Open();
        if (cmd.ExecuteNonQuery() > 0) {
            Response.Write("<script>alert('Record updated in the table.')</script>");
        }
        else {
            Response.Write("<script>alert('Failed updating record in the table.')</script>");
        }
        con.Close();
    }
    else {
        Response.Write("<script>alert('Please choose a department to update.')</script>");
    }
}
else {
    Response.Write("<script>alert('Please choose an employee to update.')</script>");
}

```

Code under Delete Button Click Event:

```

if (ddlEmp.SelectedIndex > 0) {
    cmd.CommandText = "Delete From Employee Where Eid=" + ddlEmp.SelectedValue;
    con.Open();
    if (cmd.ExecuteNonQuery() > 0) {
        Response.Write("<script>alert('Record deleted from the table.')</script>");
        txtName.Text = txtJob.Text = txtSalary.Text = "";
        ddlDept.SelectedIndex = 0;
    }
}

```

```

        LoadEmp();
        ddlEmp.Focus();
    }
    else {
        Response.Write("<script>alert('Failed deleting the record from table.')</script>");
        con.Close();
    }
}
else {
    Response.Write("<script>alert('Please choose an employee to delete.')</script>");
}

```

Loading multiple tables into DataReader: it is possible to load more than 1 table into a **DataReader**, by passing multiple **Select Statements** as **CommandText** to **Command** separated by a **semicolon**. We need to use the **NextResult** method of **DataReader** class to navigate the next tables after processing a table. To test this add a new **Web Form** in the project naming it as “**Northwind_MultipleTables.aspx**”, place 2 **GridView** controls on it and write the following code under “**Northwind_MultipleTables.aspx.cs**” file:

```
using System.Data.SqlClient;
```

Code under Page Load Event:

```
SqlConnection con = new SqlConnection("Data Source=Server;User Id=Sa;Password=123;Database=Northwind");
SqlCommand cmd = new SqlCommand("Select CustomerId, ContactName, City, Country, Phone From Customers;
                                Select EmployeeId, LastName, FirstName, BirthDate, HireDate from Employees", con);
con.Open();
SqlDataReader dr = cmd.ExecuteReader();
GridView1.DataSource = dr;
GridView1.DataBind();
dr.NextResult();
GridView2.DataSource = dr;
GridView2.DataBind();
con.Close();
```

Note: **GridView** is a control which is used for displaying data in a **table format** without manually creating a table as we have done in our first example.

DataReader: it's a class designed for holding the **data** on client machines in the form of **Rows** and **Columns**.

Features of DataReader:

1. **Faster** access to data from the **Data Source**.
2. Capable of holding **multiple tables** in it at a time.

Drawbacks of DataReader:

1. It is **connection oriented** so once the connection is **closed** between **Application** and **Data Source** all the data is lost, so **State Management** is not possible with a **DataReader**.
 2. It provides **forward-only** access to data i.e., it allows **navigating to next record or next table** only.
 3. It is **read-only**, which will not allow any **changes** to data that is present in it.
-

DataSet: it's a class which is present under “`System.Data`” namespace capable of holding data in the form of a `Table` just like a `DataReader`.

Differences between DataReader and DataSet:

1. DataReader's work in `connection oriented architecture` whereas DataSet's work in `dis-connected architecture` i.e., to hold data in `DataReader` we need to keep the connection `open` whereas in case of `DataSet` even after closing the connection also data will be persisting so `state management` is possible.
2. DataReader's provide `forward only` access to the data whereas DataSet's provides `scrollable navigation` to data which allows us to move in any direction i.e., either `top to bottom` or `bottom to top`.
3. DataReader's are `non-updatable` whereas DataSet's are `updatable` i.e. changes can be made to data present in DataSet and finally those changes can be sent back to `Database` for saving.
4. DataReader and DataSet can hold `multiple tables` whereas in case of `DataReader` all those tables must be loaded only from a `single Data Source` but in case of `DataSet` each and each table can be loaded from `different Data Sources`.

ADO.Net supports 2 different architectures for Data Source communication, like:

1. Connection Oriented Architecture
2. Disconnected Architecture

In the first case we require the connection to be kept `open` between the `Web Server` and `Data Source` for accessing the `Data` whereas in the second case we don't require the connection to be kept `open` continuously between the `Web Server` and `Data Source` i.e., we need the connection to be opened only for `loading` the Data from Data Source but not for `accessing` the Data.

Working with DataSet's: The class which is responsible for loading data into DataReader from a Data Source is Command and in the same way DataAdapter class is used for communication between Data Source and DataSet.

`DataReader <= Command => DataSource`
`DataSet <=> DataAdapter <=> DataSource`

Note: DataAdapter is internally a collection of 4 commands like “`SelectCommand`”, “`InsertCommand`”, “`UpdateCommand`” and “`DeleteCommand`” where each command is an instance of `Command` class, and by using these Commands DataAdapter will perform Select, Insert, Update and Delete operations on a DB Table.

Constructors of DataAdapter class:

- ❑ `DataAdapter()`
- ❑ `DataAdapter(Command SelectCmd)`
- ❑ `DataAdapter(string SelectCmdText, Connection con)`
- ❑ `DataAdapter(string SelectCmdText, string ConnectionString)`

Note: “Select Command Text” means it can be a Select Statement or a Stored Procedure which contains a Select Statement.

Instance of DataAdapter class can be created in any of the following ways:

```
Connection con = new Connection("<Connection String>");  
Command cmd = new Command("<Select Stmt or SP Name>", con);  
DataAdapter da = new DataAdapter();  
da.SelectCommand = cmd;
```

Or

```
Connection con = new Connection("<Connection String>");  
Command cmd = new Command("<Select Stmt or SP Name>", con);  
DataAdapter da = new DataAdapter(cmd);
```

Or

```
Connection con = new Connection("<Connection String>");  
DataAdapter da = new DataAdapter("<Select Stmt or SPName>", con);
```

Or

```
DataAdapter da = new DataAdapter("<Select Stmt or SPName>", "<Connection String>");
```

Properties of DataAdapter class:

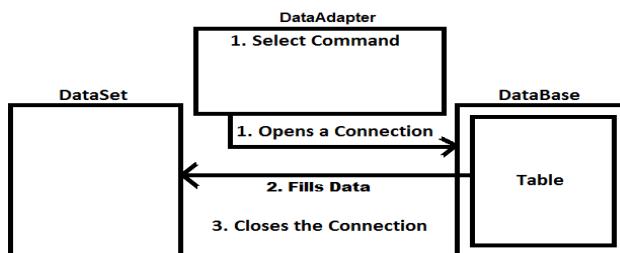
1. SelectCommand
2. InsertCommand
3. UpdateCommand
4. DeleteCommand

Methods of DataAdapter class:

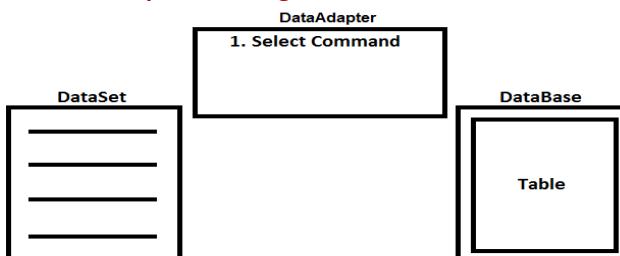
1. Fill(DataSet ds, string tableName) Data Source => DataAdapter => DataSet
2. Update(DataSet ds, string tableName) Data Source <= DataAdapter <= DataSet

When we call Fill method on DataAdapter following actions takes place internally:

1. Opens a connection with the Data Source.
2. Executes the SelectCommand that is present in DataAdapter, on Data Source and loads data from Data Source to DataSet.
3. Closes the connection with Data Source.



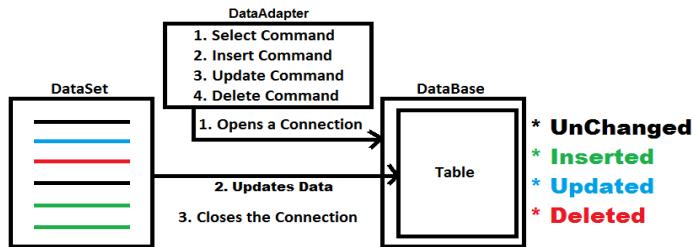
Once the execution of Fill method is completed data gets loaded into the DataSet as below:



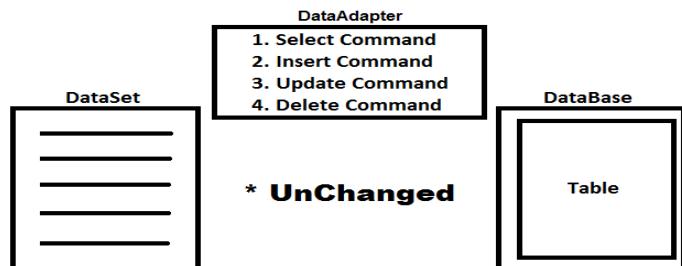
As we are discussing **DataSet** is updatable i.e., we can make changes to the data that is loaded into it like adding, modifying and deleting of records, and after making changes to data in DataSet if we want to send those changes back to Data Source we need to call **Update** method on **DataAdapter**, which performs the following:

1. Re-opens a connection with the Data Source.

2. Changes that are made to data present in DataSet will be sent back to corresponding Table and in this process it will make use of **Insert**, **Update** and **Delete** commands that are present in it.
3. Closes the connection with Data Source.



Once Update method execution is completed data gets re-loaded into DataSet as below with all unchanged rows:



Accessing data from DataSet: DataReader's provides **pointer** based access to the data, so we can get data only in a **sequential order** whereas DataSet provides **index** based access to the data, so we can get data from any location **randomly**. DataSet is a collection of tables where each table is a class: **DataTable**, and identified by its **index** position or **name**. Every **DataTable** is again collection of Rows and collection of Columns where each row is a class: **DataRow**, and identified by its **index** position and each column is a class: **DataColumn**, and identified by its **index** position or **name**.

- **Accessing a DataTable from DataSet:** <dataset>.Tables[index] or <dataset>.Tables[name]
 E.g.: ds.Tables[0] or ds.Tables["Customer"]
- **Accessing a DataRow from DataTable:** <datatable>.Rows[index]
 E.g.: ds.Tables[0].Rows[0]
- **Accessing a DataColumn from DataTable:** <datatable>.Columns[index] or <datatable>.Columns[name]
 E.g.: ds.Tables[0].Columns[0] or ds.Tables[0].Columns["Custid"]
- **Accessing a Cell from DataTable:** <datatable>.Rows[row][col]
 E.g.: ds.Tables[0].Rows[0][0] or ds.Tables[0].Rows[0]["Custid"]

To work with **DataSet** 1st create a table in our “ASPDB” with the name “Customer” and also **insert** some records into it.

```
Create Table Customer (Custid Int Constraint Custid_PK Primary Key, Name Varchar(50), Balance Money, City Varchar(50), Status Bit Default 1)
```

Now add a new WebForm in the project naming it as “ASPDB_Customer_Select.aspx” and design it as following:

Customer Details

Customer Id:	txtId
Customer Name:	txtName
Customer Balance:	txtBalance
Customer City:	txtCity
Customer Status:	<input type="checkbox"/> cbStatus

Html code for creating the above form which should be implemented under <div> tag:

```

<table align="center">
  <caption>Customer Details</caption>
  <tr>
    <td>Customer Id:</td>
    <td><asp:TextBox ID="txtId" runat="server" /></td>
  </tr>
  <tr>
    <td>Customer Name:</td>
    <td><asp:TextBox ID="txtName" runat="server" /></td>
  </tr>
  <tr>
    <td>Customer Balance:</td>
    <td><asp:TextBox ID="txtBalance" runat="server" /></td>
  </tr>
  <tr>
    <td>Customer City:</td>
    <td><asp:TextBox ID="txtCity" runat="server" /></td>
  </tr>
  <tr>
    <td>Customer Status:</td>
    <td><asp:CheckBox ID="cbStatus" runat="server" /></td>
  </tr>
  <tr>
    <td colspan="2" align="center">
      <asp:Button ID="btnFirst" runat="server" Text="First" Width="50" />
      <asp:Button ID="btnPrev" runat="server" Text="Prev" Width="50" />
      <asp:Button ID="btnNext" runat="server" Text="Next" Width="50" />
      <asp:Button ID="btnLast" runat="server" Text="Last" Width="50" />
    </td>
  </tr>
</table>

```

C# code which should be implemented in “ASPDB_Customer_Select.aspx.cs” file:

```

using System.Data;
using System.Data.SqlClient;

```

Declarations:

DataSet ds;

```
int RowIndex;

---

Code under Page Load Event:  
if(Session["CustomerDS"] == null) {  
    SqlDataAdapter da = new SqlDataAdapter("Select Custid, Name, Balance, City, Status From Customer Order By  
        Custid", "Data Source=Server;User Id=Sa;Password=123;Database=ASPDB");  
    ds = new DataSet();  
    da.Fill(ds, "Customer");  
    Session["CustomerDS"] = ds;  
    ShowData();  
}  
else {  
    ds = (DataSet)Session["CustomerDS"];  
    RowIndex = (int)Session["RowIndex"];  
}

---

private void ShowData()  
{  
    txtId.Text = ds.Tables[0].Rows[RowIndex]["Custid"].ToString();  
    txtName.Text = ds.Tables[0].Rows[RowIndex]["Name"].ToString();  
    txtBalance.Text = ds.Tables[0].Rows[RowIndex]["Balance"].ToString();  
    txtCity.Text = ds.Tables[0].Rows[RowIndex]["City"].ToString();  
    cbStatus.Checked = (bool)ds.Tables[0].Rows[RowIndex]["Status"];  
    Session["RowIndex"] = RowIndex;  
}
```

Code under First Button Click Event:

```
RowIndex = 0;  
ShowData();
```

Code under Previous Button Click Event:

```
if (RowIndex > 0) {  
    RowIndex -= 1;  
    ShowData();  
}  
else {  
    Response.Write("<script>alert('You are at first record of the table.')</script>");  
}
```

Code under Next Button Click:

```
if (RowIndex < ds.Tables[0].Rows.Count - 1) {  
    RowIndex += 1;  
    ShowData();  
}  
else {  
    Response.Write("<script>alert('You are at last record of the table.')</script>");  
}
```

Code under Last Button Click:

```
RowIndex = ds.Tables[0].Rows.Count - 1;  
ShowData();
```

Storing Connection Strings in Web.config file: in our previous examples whenever we are connecting to a **Data Source** we are writing the “**Connection String**”. Writing this “**Connection String**” each and every time will have an effect on the **size** of the application and moreover if any changes occur in the future to “**Connection String**”, then we need to modify that in each and every **Web Form**, which will be a **complex task** again. To overcome the above problems, without writing “**Connection String**” in each and every **Web Form** we can put that “**Connection String**” in “**Web.config**” file, so that we don’t require to write that in multiple **Web Forms** and also when there is a change we can modify it directly in “**Web.config**” file.

To store “**Connection String**” in the “**Web.config**” file we use “**<connectionStrings>**” tag and to test it open “**Web.config**” file and write the below code under “**<configuration>**” tag:

```
<connectionStrings>
<add name="ASPDPCS" connectionString="User Id=Sa;Password=123;Data Source=Server;
Database=ASPDB;TrustServerCertificate=true" providerName="System.Data.SqlClient" />
<add name="NorthwindCS" connectionString="User Id=Sa;Password=123;Data Source=Server;
Database=Northwind;TrustServerCertificate=true" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Reading values from Web.config file into our application: to read values from “**Web.config**” file in a **Web Form** we need to take the help of “**ConfigurationManager**” class which is present in “**System.Configuration**” namespace as following:

```
string ASPDBConStr = ConfigurationManager.ConnectionStrings["ASPDPCS"].ConnectionString;
string NorthwindConStr = ConfigurationManager.ConnectionStrings["NorthwindCS"].ConnectionString;
```

From now without writing “**Connection String**” again & again in our **Web Forms**, we can read it from “**Web.config**” file by using the above statement and consume it wherever it is required as following:

```
SqlConnection AspCon = new SqlConnection(ASPDBConStr);
SqlConnection NWCon = new SqlConnection(NorthwindConStr);
```

To simply the process of reading **connection string** values into our **Web Form** without writing lengthy code every time, add a new class into our **Project** naming it as “**ReadCS.cs**” and write the below code under the class:

```
using System.Configuration;
public class ReadCS
{
    public static string ASPDB
    {
        get { return ConfigurationManager.ConnectionStrings["ASPDPCS"].ConnectionString; }
    }
    public static string Northwind
    {
        get { return ConfigurationManager.ConnectionStrings["NorthwindCS"].ConnectionString; }
    }
}
```

Now in our **Web Forms** we can read the **Connection String** directly by calling the **static properties** we have defined as following:

```
SqlConnection AspCon = new SqlConnection(ReadCS.ASPDB);
SqlConnection NwCon = new SqlConnection(ReadCS.Northwind);
```

GridView Control

This **Control** is designed to display the data on a **Web Form** in a **Table** structure and also this **Control** has various features like **Paging, Sorting, Customizing, Editing**, etc.

Paging with GridView: this is a mechanism of displaying data on a **Web Form** as “**Blocks of Records**”, generally used when we have huge volumes of data. To perform paging with a **GridView Control** first we need to set “**AllowPaging**” property as “**true (default is false)**” and then we need to specify the no. of records that has to be displayed within each page by setting “**PageSize**” Property with a numeric value “**(default is 10)**”.

PagerSettings is a property of **GridView** using which we need to specify the type of paging the **Control** has to use with the help of “**Mode**” attribute and this can be set as “**NextPrevious or Numeric[d]**” or **NextPreviousFirstLast** or **NumericFirstLast**”. Here we can also specify the position where “**Navigation Buttons**” should be displayed and also the “**Text or Image**” values for **Next, Previous, First** and **Last** buttons.

PageIndexChanging is an event under which we need to implement the logic for navigating to next pages by setting the “**PageIndex**” property, which is “**0**” by default. To identify the **Index** of page where user wants to navigate in runtime, we can use “**NewPageIndex**” attribute of “**PageIndexChanging**” event.

To test paging, add a new **Web Form** naming it as “**Northwind_Customers_GridView_Paging.aspx**”, place a **GridView Control** on it, set the “**AllowPaging**” property value as “**true**” and write the below code in “**aspx.cs**” file:

```
using System.Data;
using System.Data.SqlClient;
```

Declarations:

```
DataSet ds;
```

Code under Page Load Event:

```
if (Session["CustomersDS"] == null) {
    SqlDataAdapter da = new SqlDataAdapter("Select CustomerID, CompanyName, ContactName, City, Country,
        PostalCode, Phone From Customers", ReadCS.Northwind);
    ds = new DataSet();
    da.Fill(ds, "Customers");
    Session["CustomerDS"] = ds;
    LoadData();
}
```

```
else {
```

```
    ds = (DataSet)Session["CustomerDS"];
}
```

```
private void LoadData() {
```

```
    GridView1.DataSource = ds;
    GridView1.DataBind();
}
```

Code under GridView PageIndexChanged Event:

```
GridView1.PageIndex = e.NewPageIndex;  
LoadData();
```

Sorting with GridView: this is a mechanism of arranging data on a **WebForm** in **ascending** or **descending** order based on any particular column of table. To perform “**Sorting**” with “**GridView**” Control, first we need to set “**AllowSorting**” property as “**true (default is false)**” so that “**Column Names**” in **GridView** looks like “**Hyper Links**” providing an option to click on them.

“**Sorting**” is an event under which we need to implement the logic that is required for **sorting** the data based on **Selected Column** which can be identified in our code by using “**SortExpression**” property of the “**Sorting**” event and this event fires when the column names (**Hyper Links**) are clicked by the user in runtime.

To test sorting with **GridView** add a new **Web Form** naming it as “**Northwind_Employees_GridView_Sorting.aspx**”, place a **GridView** control on it, set the “**AllowSorting**” property as **true** and write the below code in “**aspx.cs**” file:

```
using System.Data;  
using System.Data.SqlClient;
```

Declarations:

```
DataSet ds;
```

Code under Page Load Event:

```
if (Session["EmployeesDS"] == null) {  
    SqlDataAdapter da = new SqlDataAdapter("Select EmployeeId, LastName, FirstName, Title, BirthDate, City,  
        Country, PostalCode, HomePhone From Employees Order By EmployeeId Asc", ReadCS.Northwind);  
    ds = new DataSet(); da.Fill(ds, "Employees");  
    Session["EmployeesDS"] = ds;  
    GridView1.DataSource = ds;  
    GridView1.DataBind();  
    Session["SortOrder"] = "EmployeeId Asc";  
}  
else {  
    ds = (DataSet)Session["EmployeesDS"];  
}
```

Code under GridView Sorting Event:

```
string[] sarr = Session["SortOrder"].ToString().Split(' ');  
if(sarr[0] == e.SortExpression) {  
    if (sarr[1] == "Asc") {  
        Session["SortOrder"] = e.SortExpression + " Desc";  
    }  
    else {  
        Session["SortOrder"] = e.SortExpression + " Asc";  
    }  
}  
else {  
    Session["SortOrder"] = e.SortExpression + " Asc";  
}
```

```
}

DataView dv = ds.Tables["Employees"].DefaultView;
dv.Sort = Session["SortOrder"].ToString();
GridView1.DataSource = dv;
GridView1.DataBind();
```

Customizing a GridView: generally, when we bind a “Data Source” to “GridView” Control it will automatically display all the columns that are being retrieved by us, because **GridView** control has a mechanism of “auto-generating” columns and display the data as a **HTML Table**., so that “Column Names” will be displayed as “Header Text” and “Column Values” will be displayed in the body. We can customize a **GridView** so that we can display different “Header Text” apart from “Column Names”, define styles for “Header Text”, “Item Text” as well as we can also use different type of controls for displaying data and even, we can perform “Editing” of the data, but to do these all first we need to set “AutoGenerateColumns” property of the control as “false (default is true)”. Once we set “AutoGenerateColumns” property value as “false”, it is our responsibility to add each and every column individually to the **GridView** control with the help of some special **fields** like:

- ② **BoundField:** Displays the value of a column in a Data Source. This is the default column type of the **GridView** control.
 - ② **ButtonField:** Displays a command button for each item in the **GridView** control. This enables you to create a column of custom button controls, such as the Add or the Remove button.
 - ② **CheckBoxField:** Displays a check box for each item in the **GridView** control. This column field type is commonly used to display fields with a Boolean value.
 - ② **CommandField:** Displays pre-defined command buttons to perform Select, Edit, or Delete operations.
 - ② **HyperLinkField:** Displays the value of a field in a Data Source as a hyperlink. This column field type enables you to bind a second field to the hyperlink's URL.
 - ② **ImageField:** Displays an image for each item in the **GridView** control.
 - ② **TemplateField:** Displays user-defined content for each item in the **GridView** control, according to a specified template. This column field type enables you to create a custom column field.
-

To test these, add a new Web Form, naming it as “**ASPDB_Customer_GridView_Customize.aspx**” and write the following code under its **<div>** tag:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="false" Caption="Customer Details"
    HorizontalAlign="Center" >
<HeaderStyle BackColor="LightYellow" ForeColor="PaleVioletRed" />
<RowStyle BackColor="Tan" ForeColor="Teal" />
<AlternatingRowStyle BackColor="Teal" ForeColor="Tan" />
<Columns>
    <asp:BoundField DataField="Custid" HeaderText="Cust-ID" ItemStyle-HorizontalAlignment="Center" />
    <asp:BoundField DataField="Name" HeaderText="Name" />
    <asp:BoundField DataField="Balance" HeaderText="Balance" ItemStyle-HorizontalAlignment="Right" />
    <asp:BoundField DataField="City" HeaderText="City" />

    <asp:CheckBoxField DataField="Status" HeaderText="Is-Active" ItemStyle-HorizontalAlignment="Center" />
    Or
    <asp:TemplateField HeaderText="Is-Active" ItemStyle-HorizontalAlignment="Center">
```

```

<ItemTemplate>
    <asp:RadioButton ID="rbStatus" runat="server" Enabled="false" Checked='<%# Eval("Status") %>' />
</ItemTemplate>
</asp:TemplateField>

</Columns>
</asp:GridView>

```

Now go to “ASPDB_Customer_GridView_Customize.aspx.cs” file and write the following code:

```
using System.Data.SqlClient;
```

Code under Page Load Event:

```

SqlConnection con = new SqlConnection(ReadCS.ASPDB);
SqlCommand cmd = new SqlCommand(
    "Select Custid, Name, Balance, City, Status From Customer Order By Custid", con);
con.Open();
SqlDataReader dr = cmd.ExecuteReader();
GridView1.DataSource = dr;
GridView1.DataBind();
con.Close();

```

Data editing with GridView: we can edit data that is present in “GridView Control” and update those changes back to the corresponding table in **Database**. To perform editing of the data, first “GridView Control” requires an “Edit” & “Delete” button, and to generate these buttons we have 3 options:

1. Set the properties **AutoGenerateDeleteButton** and **AutoGenerateEditButton** as true.
2. Add **Edit** & **Delete** options using “**CommandField**”, by manually generating the columns as following:
`<asp:CommandField ShowEditButton="true" ShowDeleteButton="true" />`
3. Manual generation of those buttons by using **Template Field**.

To perform editing operations, we need to write logic under 4 events of the **GridView** those are: **RowEditing**, **RowCancelingEdit**, **RowUpdating** and **RowDeleting**.

- **RowEditing:** Occurs when a row's **Edit** button is clicked which changes **Edit** & **Delete** buttons to **Update** & **Cancel** buttons and here, we need to set the **GridView**'s **EditIndex** to selected rows Index.
- **RowCancelingEdit:** Occurs when the **Cancel** button of a row in edit mode is clicked which changes back to **Edit** & **Delete** buttons, and here we need to set the **GridView**'s **EditIndex** to -1.
- **RowUpdating:** Occurs when a row's **Update** button is clicked, and here we need to implement the logic for updating the row and finally set the **GridView**'s **EditIndex** to -1.
- **RowDeleting:** Occurs when a row's **Delete** button is clicked, and here we need to implement the logic for deleting the row.

Now to test editing with **GridView**, add a new **Web Form** in the project naming it as “**ASPDB_Customer_GridView_Editing.aspx**” and write the following code under its **<div>** tag:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="false" Caption="Customer Editing"
```

```

        HorizontalAlign="Center">
<HeaderStyle BackColor="Yellow" ForeColor="Red" />
<RowStyle ForeColor="Tomato" BackColor="YellowGreen" />
<AlternatingRowStyle ForeColor="SlateGray" BackColor="Wheat" />
<EditRowStyle BackColor="LightCoral" ForeColor="WindowFrame" />
<Columns>
    <asp:BoundField HeaderText="Customer Id" DataField="Custid" ReadOnly="true"
                    ItemStyle-HorizontalAlign="Center" />
    <asp:BoundField HeaderText="Name" DataField="Name" />
    <asp:BoundField HeaderText="Balance" DataField="Balance" ItemStyle-HorizontalAlign="Right" />
    <asp:BoundField HeaderText="City" DataField="City" />
    <asp:CheckBoxField HeaderText="Is-Active" DataField="Status" ItemStyle-HorizontalAlign="Center"
                       ReadOnly="true" />
<asp:TemplateField HeaderText="Actions" ItemStyle-BackColor="White" ItemStyle-ForeColor="Blue">
    <ItemTemplate>
        <asp:LinkButton ID="btnEdit" runat="server" Text="Edit" CommandName="Edit" />
        <asp:LinkButton ID="btnDelete" runat="server" Text="Delete" CommandName="Delete"
                       OnClientClick="return confirm('Are you sure of deleting the current record?')"/>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:LinkButton ID="btnUpdate" runat="server" Text="Update" CommandName="Update" />
        <asp:LinkButton ID="LinkCancel" runat="server" Text="Cancel" CommandName="Cancel" />
    </EditItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>

```

Now go to design view of the **Web Form**, select “**GridView**”, open the **Property Window**, go to its **Events**, double click on “**RowEditing**”, “**RowCancelingEdit**”, “**RowUpdating**” and “**RowDeleting**” events to generate **Event Handlers** for implementing the logic and write the below code in “**ASPDB_Customer_GridView_Editing.aspx.cs**”:

```

using System.Data;
using System.Data.SqlClient;

```

Declarations:

```

SqlCommand cmd;
SqlConnection con;

```

Code under Page Load Event:

```

con = new SqlConnection(ReadCS.ASPDB);
cmd = new SqlCommand();
cmd.Connection = con;
if (!IsPostBack) {
    LoadData();
}

```

```

private void LoadData()
{
    try {

```

```

cmd.CommandText = "Select Custid,Name,Balance,City,Status From Customer Where Status=1 Order By Custid";
if (con.State != ConnectionState.Open) {
    con.Open();
}
SqlDataReader dr = cmd.ExecuteReader();
GridView1.DataSource = dr;
GridView1.DataBind();
}
catch (Exception ex) {
    Response.Write("<script>alert('" + ex.Message.Replace("'", "") + "')</script>");
}
finally {
    if (con.State != ConnectionState.Closed)
        con.Close();
}

```

Code under GridView RowEditing Event:

```

GridView1.EditIndex = e.NewEditIndex;
LoadData();

```

Code under GridView RowCancelingEdit Event:

```

GridView1.EditIndex = -1;
LoadData();

```

Code under GridView RowUpdating Event:

```

try
{
    int Custid = int.Parse(GridView1.Rows[e.RowIndex].Cells[0].Text);
    string Name = ((TextBox)GridView1.Rows[e.RowIndex].Cells[1].Controls[0]).Text;
    decimal Balance = decimal.Parse(((TextBox)GridView1.Rows[e.RowIndex].Cells[2].Controls[0]).Text);
    string City = ((TextBox)GridView1.Rows[e.RowIndex].Cells[3].Controls[0]).Text;

    cmd.CommandText =
        $"Update Customer Set Name='{Name}', Balance={Balance}, City='{City}' Where Custid={Custid}";
    con.Open();
    if (cmd.ExecuteNonQuery() > 0)
    {
        GridView1.EditIndex = -1;
        LoadData();
    }
}
catch (Exception ex) {
    Response.Write("<script>alert('" + ex.Message.Replace("'", "") + "')</script>");
}
finally
{

```

```

    if (con.State != ConnectionState.Closed)
        con.Close();
}

```

Code under GridView RowDeleting Event:

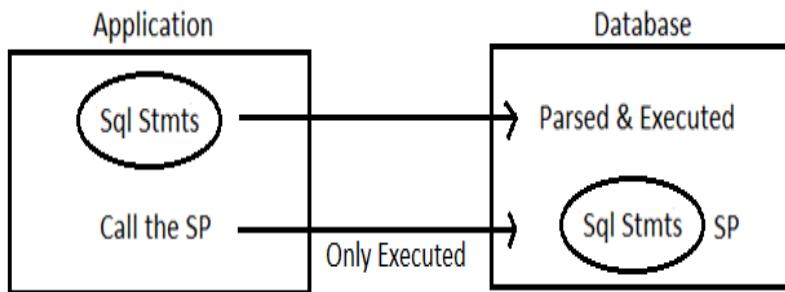
```

try {
    int Custid = int.Parse(GridView1.Rows[e.RowIndex].Cells[0].Text);
    cmd.CommandText = $"Update Customer Set Status=0 Where Custid={Custid}";
    con.Open();
    if (cmd.ExecuteNonQuery() > 0) {
        LoadData();
    }
}
catch (Exception ex) {
    Response.Write("<script>alert(\"" + ex.Message.Replace(" ", "") + "\")</script>");
}
finally {
    if (con.State != ConnectionState.Closed)
        con.Close();
}

```

Stored Procedures

Whenever we want to interact with a **Database** from an application, we use **SQL Statements**. When we use **SQL Statements** within the application we have a problem i.e., when the application runs **SQL Statements** will be sent to **Database** for execution where the statements will be parsed (compiled) and then executed. The process of parsing takes place each time we run the application, because of this performance of our application decreases. To overcome the above drawback, write **SQL Statements** directly under **Database** only, with in an object known as **Stored Procedure** and call them for execution. As a **Stored Procedure** is a pre-compiled block of code that is ready for execution will directly execute the statements without parsing each time.



Syntax to define a Stored Procedure:

Create Procedure <Name> [(<Parameter List>)]

As

Begin

<Stmt's>

End;

- SP's are similar to methods in our language.

public void Test()	//Method
Create Procedure Test()	//Stored Procedure

- If required we can also define parameters to Stored Procedures but only optional. If we want to pass parameters to a SQL Server Stored Procedure prefix the special character "@" before parameter name.

public void Test(int x)	//CSharp
Create Procedure Test(@x int)	//SQL Server

- A Stored Procedure can also return values, to return a value we use Out or Output keyword in SQL Server.

public void Test(int x, out int y)	//CSharp
Create Procedure Test(@x int, @y int out)	//SQL Server

Creating a Stored Procedure: we can create **Stored Procedure** in **SQL Server** either by using **SQL Server Management Studio** or **Visual Studio** also. To create a **Stored Procedure** using **Visual Studio** open the “**Server Explorer**” window, this will display “**ASPDB**” and “**Northwind**” Database there and those **Databases** are configure based on the “**Connection String**” value we have used in “**Web.config**” file. Expand our “**ASPDB**” database, right click on the node **Stored Procedures**, select “**Add New Stored Procedure**” which opens a window write code in it for creating a **Stored Procedure** and finally right click on the document window and select “**Execute**” which will create the **Stored Procedure** on **Database Server**. Now let’s create the below **Procedures** under “**ASPDB**” Database.

```
Create Procedure Customer_Select(@Custid Int=NULL, @Status Bit=NULL)
As
Begin
If @Custid Is Null And @Status Is Null      --Fetches all records from the table
    Select Custid, Name, Balance, City, Status From Customer Order By Custid;
Else If @Custid Is Not Null And @Status Is Null   --Fetches a record from the table based on given Custid
    Select Custid, Name, Balance, City, Status From Customer Where Custid=@Custid;
Else If @Custid Is Null And @Status Is Not Null   --Fetches records from the table based on given Status
    Select Custid, Name, Balance, City, Status From Customer Where Status=@Status Order By Custid;
Else If @Custid Is Not Null And @Status Is Not Null --Fetches a record from the table based on given Id & Status
    Select Custid, Name, Balance, City, Status From Customer Where Custid=@Custid And Status=@Status;
End;
```

```
Create Procedure Customer_Insert(@Name Varchar(50), @Balance Money, @City Varchar(100), @Status Bit,
@Custid Int Out)
```

```
As
Begin
Begin Transaction
Select @Custid = IsNull(Max(Custid), 100) + 1 From Customer;
Insert Into Customer Values (@Custid, @Name, @Balance, @City, @Status);
Commit Transaction;
End;
```

```
Create Procedure Customer_Update(@Custid Int, @Name Varchar(50), @Balance Money, @City Varchar(100))
As
```

```
Update Customer Set Name=@Name, Balance=@Balance, City=@City Where Custid=@Custid;
```

```
Create Procedure Customer_Delete(@Custid Int)
```

As

Update Customer Set Status=0 Where Custid=@Custid;

Calling a Stored Procedure from .NET application: To call a **Stored Procedure** from .NET Application's we use **Command** class and the process of calling will be as following:

1. Create instance of class **Command** by specifying **Stored Procedure Name** as **CommandText**.

```
Command cmd = new Command("<SP Name>", con);
or
Command cmd = new Command();
cmd.Connection = con;
cmd.CommandText = "<SP Name>";
```

2. Change the value of **CommandType** property of **Command** class instance as "**StoredProcedure**" because by default the value of **CommandType** property is "**Text**", which means it can execute **SQL Statements** only, so by changing it as "**StoredProcedure**" we are configuring **Command** to call **Stored Procedures**.

```
cmd.CommandType = CommandType.StoredProcedure;
```

3. If the **Stored Procedures** has any **parameters**, then we need to send values to those **parameters**, if they are **input parameters** or else, we need to capture the values from those **parameters** if they are **output parameters**, and to do this we need to first add those parameters to **Command** by calling the method **AddWithValue** for **input parameters** and **Add** method for **output parameters**.
4. If the **Stored Procedure** we want to **execute** is a **Query Procedure**, then call **ExecuteReader()** method on **Command** instance which executes the **Stored Procedure** and loads data into **DataReader**, whereas if we want to load data into **DataSet** then create **DataAdapter** class instance by passing **Command** class instance as a parameter to the **Constructor** and then call **Fill** method. If the **Stored Procedure** we want to call is a **non-Query Procedure**, then call **ExecuteNonQuery** method of **Command** to execute the **Stored Procedure**.

Adding Parameter values to Command: SP can be defined with parameters either to send values for execution or receive values after execution. While calling a SP with parameters from .NET Application for each parameter of the SP we need to add a matching parameter under Command i.e., for input parameter matching input parameter has to be added and for output parameter a matching output parameter has to be added. Every parameter has 5 attributes to it like **Name**, **Value**, **DbType**, **Size** and **Direction** which can be **Input** (d) or **Output**.

- **Name** refers to name of the parameter that is defined in SP.
- **Value** refers to value being assigned in case of input or value we are expecting in case of output.
- **DbType** refers to data type of the parameter in terms of the Database where the SP exists.
- **Size** refers to size of data.
- **Direction** specifies whether parameter is **Input** or **Output**.

Based on SP's Input or Output parameters we need to add them under Command by specifying below attributes:

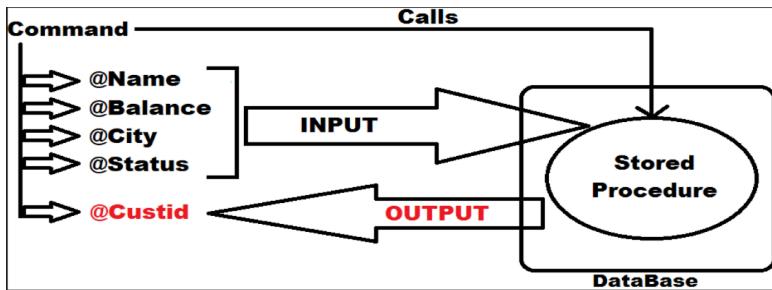
	<u>Input</u>	<u>Output</u>
Name	Yes	Yes
Value	Yes	No
DB Type	Yes [*]	Yes
Size	No	Yes [**]

Direction	No	Yes
-----------	----	-----

*Required only if the value supplied is null.

**Required only in case the output parameter type is a variable length type.

If we want to call the **Insert - Stored Procedure** i.e., “Customer_Insert” which we defined above we need to add both **Input** and **Output** parameters under **Command** as below:



In the above case the **4 Input parameters** (@Name, @Balance, @City, @Status) are sent to the **SP** for execution and once after the **Procedure** got executed will send the **Output parameter** (@Custid) as a result.

Adding input parameters under Command:

```
cmd.Parameters.AddWithValue(string <pname>, object <pvalue>)
```

Adding output parameters under Command:

```
cmd.Parameters.Add(string <pname>, DbType <type>[, int <size>]).Direction = ParameterDirection.Output;
```

Capturing output parameter values after execution of SP:

```
Object obj = cmd.Parameters[string <pname>].Value;
```

Note: **ParameterDirection** is an **enum** which contains all the **directions** that are supported like **Input**, **Output**, **InputOutput** and **ReturnValue**.

Now to call the above **Stored Procedures** in our **Web Forms** add a new class naming it as “Customer.cs” and write methods with all the necessary logic for calling the **Stored Procedure’s**, so that we can call those methods directly in our **Web Forms** without writing the logic every time when we want to call the **Procedure**.

```

using System.Data;
using System.Data.SqlClient;
public class Customer
{
    SqlCommand cmd;
    SqlConnection con;
    public Customer()
    {
        con = new SqlConnection(ReadCS.ASPDB);
        cmd = new SqlCommand();
        cmd.Connection = con;
        cmd.CommandType = CommandType.StoredProcedure;
    }
}

```

```

}

public DataSet Customer_Select(int? Custid, bool? Status)
{
    DataSet ds;
    try {
        cmd.CommandText = "Customer_Select";
        cmd.Parameters.Clear();
        if (Custid != null && Status == null)
            cmd.Parameters.AddWithValue("@Custid", Custid);
        else if (Custid == null && Status != null)
            cmd.Parameters.AddWithValue("@Status", Status);
        else if(Custid != null && Status != null) {
            cmd.Parameters.AddWithValue("@Custid", Custid);
            cmd.Parameters.AddWithValue("@Status", Status);
        }
        SqlDataAdapter da = new SqlDataAdapter(cmd);
        ds = new DataSet();
        da.Fill(ds, "Customer");
        return ds;
    }
    catch(Exception ex) {
        throw ex;
    }
}

public int Customer_Insert(string Name, decimal? Balance, string City, bool? Status, ref int? Custid)
{
    int Count = 0;
    try {
        cmd.CommandText = "Customer_Insert";
        cmd.Parameters.Clear();
        cmd.Parameters.AddWithValue("@Name", Name);
        cmd.Parameters.AddWithValue("@Balance", Balance);
        cmd.Parameters.AddWithValue("@City", City);
        cmd.Parameters.AddWithValue("@Status", Status);
        cmd.Parameters.Add("@Custid", SqlDbType.Int).Direction = ParameterDirection.Output;
        con.Open();
        Count = cmd.ExecuteNonQuery();
        Custid = Convert.ToInt32(cmd.Parameters["@Custid"].Value);
    }
    catch (Exception ex) {
        throw ex;
    }
    finally {
        con.Close();
    }
    return Count;
}

```

```

}

public int Customer_Update(int? Custid, string Name, decimal? Balance, string City)
{
    int Count = 0;
    try {
        cmd.CommandText = "Customer_Update";
        cmd.Parameters.Clear();
        cmd.Parameters.AddWithValue("@Custid", Custid);
        cmd.Parameters.AddWithValue("@Name", Name);
        cmd.Parameters.AddWithValue("@Balance", Balance);
        cmd.Parameters.AddWithValue("@City", City);
        con.Open();
        Count = cmd.ExecuteNonQuery();
    }
    catch (Exception ex) {
        throw ex;
    }
    finally {
        con.Close();
    }
    return Count;
}
public int Customer_Delete(int? Custid)
{
    int Count = 0;
    try {
        cmd.CommandText = "Customer_Delete";
        cmd.Parameters.Clear();
        cmd.Parameters.AddWithValue("@Custid", Custid);
        con.Open();
        Count = cmd.ExecuteNonQuery();
    }
    catch (Exception ex) {
        throw ex;
    }
    finally {
        con.Close();
    }
    return Count;
}
}

```

Now we can call the above methods that are defined in **Customer** class to perform **Select**, **Insert**, **Update** and **Delete** operations in any **WebForm** and to test this add a new Web Form in our project naming it as “**ASPDB_Customer_GridView_Editing_SP.aspx**” and design it same as “**ASPDB_Customer_GridView_Editing.aspx**” and write the following code in “**ASPDB_Customer_GridView_Editing_SP.aspx.cs**” file:

Declarations:

```
Customer obj = new Customer();
```

Code under Page Load Event:

```
if (!IsPostBack) {  
    LoadData();  
}  
  
private void LoadData() {  
    GridView1.DataSource = obj.Customer_Select(null, true);  
    GridView1.DataBind();  
}
```

Code under Page Error Event:

```
Exception ex = Server.GetLastError();  
Server.ClearError();  
Response.Write("<script>alert('" + ex.Message.Replace("'", "") + "')</script>");
```

Code under GridView RowEditing Event:

```
GridView1.EditIndex = e.NewEditIndex;  
LoadData();
```

Code under GridView RowCancelingEdit Event:

```
GridView1.EditIndex = -1;  
LoadData();
```

Code under GridView RowUpdating Event:

```
int Custid = int.Parse(GridView1.Rows[e.RowIndex].Cells[0].Text);  
string Name = ((TextBox)GridView1.Rows[e.RowIndex].Cells[1].Controls[0]).Text;  
decimal Balance = decimal.Parse(((TextBox)GridView1.Rows[e.RowIndex].Cells[2].Controls[0]).Text);  
string City = ((TextBox)GridView1.Rows[e.RowIndex].Cells[3].Controls[0]).Text;  
if(obj.Customer_Update(Custid, Name, Balance, City) > 0) {  
    GridView1.EditIndex = -1;  
    LoadData();  
}  
else {  
    Response.Write("<script>alert('Failed updating the record in table.')</script>");  
}
```

Code under GridView RowDeleting Event:

```
int Custid = int.Parse(GridView1.Rows[e.RowIndex].Cells[0].Text);  
if (obj.Customer_Delete(Custid) > 0) {  
    LoadData();  
}  
else {  
    Response.Write("<script>alert('Failed deleting the record from table.')</script>");  
}
```

Storing Images in Database

We can store **images**, **audio** and **video** data into **Database** and while saving them we can either save **path** of those files in a column or content of those files in **Database** by converting them into **byte[]** or **binary** format. To store **byte[]** or **binary** data, column should use **Image** or **Varbinary** data type.

Create a table under our ASPDB database representing a Student entity as following:

Create Table Student (Sid Int Constraint Sid_Pk Primary Key, Name Varchar(50), Class Int, Fees Money, PhotoName Varchar(50), PhotoBinary VarBinary(Max), Status Bit Not Null Default 1);

Add a WebForm in the project naming it as “ASPDB_Student_DataMgmt.aspx” and design it as following:

Student Details			
Student Id.:	<input id="txtId" type="text"/>		
Student Name:	<input id="txtName" type="text"/>		
Student Class:	<input id="txtClass" type="text"/>		
Annual Fees:	<input id="txtFees" type="text"/>		
Select	<input type="button" value="Insert"/>	<input type="button" value="Choose File"/>	No file chosen
Update	<input type="button" value="Delete"/>	Upload Image	
Reset All			

imgPhoto

Html code for creating the above form which should be implemented under <div> tag:

```
<table align="center" style="background-color: bisque">
    <caption>Student Data Management</caption>
    <tr>
        <td>Student Id:</td>
        <td><asp:TextBox ID="txtId" runat="server" /></td>
        <td rowspan="4">
            <asp:Image ID="imgPhoto" runat="server" Height="200px" Width="200px" BorderStyle="Groove" />
        </td>
    </tr>
    <tr>
        <td>Student Name:</td>
        <td><asp:TextBox ID="txtName" runat="server" /></td>
    </tr>
    <tr>
        <td>Student Class:</td>
        <td><asp:TextBox ID="txtClass" runat="server" /></td>
    </tr>
    <tr>
        <td>Annual Fees:</td>
        <td><asp:TextBox ID="txtFees" runat="server" /></td>
    </tr>
    <tr>
        <td colspan="3" style="text-align: center; padding: 5px;">
            <asp:Button ID="btnSelect" runat="server" Text="Select" Width="100px" />
            <asp:Button ID="btnInsert" runat="server" Text="Insert" Width="100px" />
            <asp:FileUpload ID="FileUpload1" runat="server" />
        </td>
    </tr>
</table>
```

```

<tr>
<td colspan="3">
<asp:Button ID="btnUpdate" runat="server" Text="Update" Width="100px" />
<asp:Button ID="btnDelete" runat="server" Text="Delete" Width="100px" />
<asp:Button ID="btnUpload" runat="server" Text="Upload Image" Width="270px" />
</td>
</tr>
<tr>
<td colspan="3"><asp:Button ID="btnReset" runat="server" Text="Reset All" Width="478px" /></td>
</tr>
</table>
<asp:Label ID="lblMsgs" runat="server" ForeColor="Red" />

```

C# code which should be implemented in “ASPDB_Student_DataMgmt.aspx.cs” file:

```

using System.IO;
using System.Data;
using System.Data.SqlClient;

```

Declarations:

```

SqlCommand cmd;
SqlConnection con;

```

Code under Page Load Event:

```

if(!IsPostBack) {
    txtId.Focus();
}
con = new SqlConnection(ReadCS.ASPDB);
cmd = new SqlCommand();
cmd.Connection = con;

```

Code under Upload Image Button Click Event:

```

if(FileUpload1.HasFiles) {
    HttpPostedFile selectedFile = FileUpload1.PostedFile;
    string fileExtension = Path.GetExtension(selectedFile.FileName);
    if(fileExtension == ".jpg" || fileExtension == ".bmp" || fileExtension == ".png") {
        string imgName = selectedFile.FileName;
        string folderPath = Server.MapPath("~/Images/");
        if(!Directory.Exists(folderPath)) {
            Directory.CreateDirectory(folderPath);
        }
        selectedFile.SaveAs(folderPath + imgName);
        imgPhoto.ImageUrl = "~/Images/" + imgName;
        BinaryReader br = new BinaryReader(selectedFile.InputStream);

        //Converting the image into byte[] (Binary Format)
        byte[] imgData = br.ReadBytes(selectedFile.ContentLength);

        //Storing image name & image binary values in session to access them in Insert & Update
        Session["PhotoName"] = "~/Images/" + imgName;
    }
}

```

```

Session["PhotoBinary"] = imgData;
}
else {
    Response.Write("<script>alert('Supported image file formats are .jpg, .bmp and .png only.')</script>");
}
}
else {
    Response.Write("<script>alert('Please select an image file to upload.')</script>");
}

```

```

private void ClearData()
{
    txtId.Text = txtName.Text = txtClass.Text = txtFees.Text = lblMsgs.Text = imgPhoto.ImageUrl = "";
    Session["PhotoName"] = Session["PhotoBinary"] = null;
    txtId.Focus();
}

```

Code under Reset All Button Click Event:

```
ClearData();
```

Code under Select Button Click Event:

```

try
{
    cmd.CommandText =
        "Select Sid, Name, Class, Fees, PhotoName, PhotoBinary From Student Where Status=1 and Sid=" + txtId.Text;
    con.Open();
    SqlDataReader dr = cmd.ExecuteReader();
    if (dr.Read()) {
        txtName.Text = dr["Name"].ToString();
        txtClass.Text = dr["Class"].ToString();
        txtFees.Text = dr["Fees"].ToString();
        if (dr["PhotoName"] != DBNull.Value) {
            imgPhoto.ImageUrl = dr["PhotoName"];
            Session["PhotoName"] = dr["PhotoName"].ToString();
        }
        else {
            imgPhoto.ImageUrl = "";
            Session["PhotoName"] = null;
        }
        if (dr["PhotoBinary"] != DBNull.Value) {
            Session["PhotoBinary"] = (byte[])dr["PhotoBinary"];
        }
        else {
            Session["PhotoBinary"] = null;
        }
    }
    else {
        Response.Write("<script>alert('No student exists with given ID.')</script>");
    }
}

```

```

        ClearData();
    }
}
catch (Exception ex) {
    lblMsgs.Text = ex.Message;
}
finally {
    con.Close();
}



---


private void AddParameters()
{
    cmd.Parameters.AddWithValue("@Sid", txtId.Text);
    cmd.Parameters.AddWithValue("@Name", txtName.Text);
    cmd.Parameters.AddWithValue("@Class", txtClass.Text);
    cmd.Parameters.AddWithValue("@Fees", txtFees.Text);
    if (Session["PhotoName"] != null) {
        cmd.Parameters.AddWithValue("@PhotoName", Session["PhotoName"].ToString());
    }
    else {
        cmd.Parameters.AddWithValue("@PhotoName", DBNull.Value);
        cmd.Parameters["@PhotoName"].SqlDbType = SqlDbType.VarChar;
    }
    if (Session["PhotoBinary"] != null) {
        cmd.Parameters.AddWithValue("@PhotoBinary", (byte[])Session["PhotoBinary"]);
    }
    else {
        cmd.Parameters.AddWithValue("@PhotoBinary", DBNull.Value);
        cmd.Parameters["@PhotoBinary"].SqlDbType = SqlDbType.VarBinary;
    }
}

Code under Insert Button Click Event:
try {
    cmd.CommandText = "Insert Into Student (Sid, Name, Class, Fees, PhotoName, PhotoBinary) Values(@Sid,
                                                @Name, @Class, @Fees, @PhotoName, @PhotoBinary)";
    AddParameters();
    con.Open();
    cmd.ExecuteNonQuery();
    Response.Write("<script>alert('Record inserted into the table.')</script>");
}
catch(Exception ex) {
    lblMsgs.Text = ex.Message;
}
finally {
    con.Close();
}



---


Code under Update Button Click Event:

```

```

try {
    cmd.CommandText = "Update Student Set Name=@Name, Class=@Class, Fees=@Fees,
                      PhotoName=@PhotoName, PhotoBinary=@PhotoBinary Where Sid=@Sid";
    AddParameters();
    con.Open();
    cmd.ExecuteNonQuery();
    Response.Write("<script>alert('Record updated in the table.')</script>");
}
catch (Exception ex) {
    lblMsgs.Text = ex.Message;
}
finally {
    con.Close();
}

```

Code under Delete Button Click Event:

```

try {
    cmd.CommandText = "Update Student Set Status=0 Where Sid=@Sid";
    cmd.Parameters.AddWithValue("@Sid", txtId.Text);
    con.Open();
    cmd.ExecuteNonQuery();
    ClearData();
    Response.Write("<script>alert('Record deleted from the table.')</script>");
}
catch (Exception ex) {
    lblMsgs.Text = ex.Message;
}
finally {
    con.Close();
}

```

Caching

One of the most important factors in building high-performance, scalable Web applications is the ability to store items, whether data objects or pages, or parts of a page, in memory the initial time they are requested. You can cache, or store, these items on the Web Server or other software in the request stream, such as the Proxy Server or Browser. This allows you to avoid recreating information that satisfied a previous request, particularly information that demands significant processor time or other resources. ASP.NET caching allows you to use a technique to store page output or application data across HTTP Requests and reuse it.

An application can often increase performance by storing data in memory that is accessed frequently and that requires significant processing time to create. For example, if your application processes large amounts of data using complex logic and then returns the data as a report accessed frequently by users, it is efficient to avoid re-creating the report every time that a user requests it. Similarly, if your application includes a page that processes complex data but that is updated only in-frequently, it is in-efficient for the server to re-create that page on every request. To help us increase application performance in these situations, ASP.NET provides caching which is divided into 2 categories like:

1. Output Caching
2. Data Caching

Output Caching: Output caching allows you to store dynamic page and user control responses on any HTTP cache-capable device in the output stream, from the originating server to the requesting browser. On subsequent requests, the page or user control code is not executed; the cached output is used to satisfy the request. To enable Output Caching we need to use “**OutputCache Directive**” either in a WebForm or a WebUserControl as following:

```
<%@ OutputCache Location="None/Any/Client/Server/Downstream" Duration=<time in seconds>
    VaryByParam="<none/value>" %>
```

- Location is to specify where the cache must be stored, and the default is Any.
- Duration is to specify the time output should be stored in cache memory, in seconds.
- VaryByParam is to specify the parameter based on which the cached output should vary.

Create a new “**ASP.Net Web Application**” project naming it as “**CachingSite**”. Open “**Web.config**” file and write the “**Connection String**” under “**<configuration>**” tag for connecting to SQL Server Database:

```
<connectionStrings>
<add name="ConStr" providerName="System.Data.SqlClient"
connectionString="User Id=Sa;Password=123;Database=ASPDB;Data Source=Server;TrustServerCertificate=true" />
</connectionStrings>
```

Add a new class in the project naming it as “**ReadCS.cs**” and write the below code in it by importing the namespace “**System.Configuration**”.

```
public class ReadCS {
    public static string ASPDB {
        get { return ConfigurationManager.ConnectionStrings["ConStr"].ConnectionString; }
    }
}
```

Add a Web Form in the Project naming it as “OutputCache.aspx” and write the below code under “<div>” tag:

```
<asp:GridView ID="GridView1" runat="server" />
Client Machine Date & Time: <script>document.write(Date());</script>
```

Write the below code under “OutputCache.aspx.cs” file:

```
using System.Data;
using System.Data.SqlClient;
```

Code under Page Load Event:

```
SqlDataAdapter da = new SqlDataAdapter("Select * from Customer Where Status=1", ReadCS.ASPDB);
DataSet ds = new DataSet();
da.Fill(ds, "Customer");
GridView1.DataSource = ds;
GridView1.DataBind();
Response.Write("Server Machine Date & Time: " + DateTime.Now.ToString());
```

Now if you run the Web Form and watch the output the Server Time value and Browser Time value will be same. Now refresh the page and watch then also both those values will be same, whereas if we enable caching then the Server Time and Browser Time values will be different, to test that go to “[OutputCache.aspx](#)” file and write the following code below the Page Directive:

```
<%@ OutputCache Location="Server" Duration="30" VaryByParam="none" %>
```

VaryByParam attribute of Caching: add a new Web Form under the project naming it as “[MasterDetail.aspx](#)” and write the below code under its “`<div>`” tag:

```
<table>
<tr><td align="center"><asp:DropDownList ID="ddlDept" runat="server" AutoPostBack="true" /></td></tr>
<tr><td><asp:GridView ID="gvEmp" runat="server" /></td></tr>
</table>
Client Machine Date & Time:<script>document.write(Date());</script>
```

Write the below code under “MasterDetail.aspx.cs” file:

```
using System.Data;
using System.Data.SqlClient;
```

Declarations:

```
DataSet ds;
SqlDataAdapter da;
SqlConnection con;
```

Code under Page Load Event:

```
con = new SqlConnection(ReadCS.ASPDB);
if (!IsPostBack) {
    da = new SqlDataAdapter("Select * From Department", con);
    ds = new DataSet();
    da.Fill(ds, "Department");

    da.SelectCommand.CommandText = "Select * From Employee";
    da.Fill(ds, "Employee");
    ddlDept.DataSource = ds.Tables["Department"];
    ddlDept.DataTextField = "Dname";
    ddlDept.DataValueField = "Did";
    ddlDept.DataBind();
    ddlDept.Items.Insert(0, "All");

    gvEmp.DataSource = ds.Tables["Employee"];
    gvEmp.DataBind();
}
```

Code under DropDownList SelectedIndexChanged Event:

```
if (ddlDept.SelectedIndex == 0) {
    da = new SqlDataAdapter("Select * From Employee", con);
}
else {
    da = new SqlDataAdapter("Select * From Employee Where Did=" + ddlDept.SelectedValue, con);
```

```

}
ds = new DataSet();
da.Fill(ds, "Employee");
gvEmp.DataSource = ds.Tables["Employee"];
gvEmp.DataBind();
Response.Write("Server Machine Date & Time: " + DateTime.Now);

```

Now turn on caching under the above Web Form by using the OutputCache directive as following:

```
<%@ OutputCache Location="Server" Duration="300" VaryByParam="none" %>
```

Now run the page and select a Department Name from DropDownList control and data gets cached at that point of time, but when we change the selection of Name it will still display the old, cached output only because the cached output duration is 300 seconds, but we require the output to be changing based on the Department Name we select and to achieve that we need to use the “VaryByParam” attribute of “OutputCache” directive which should be as following:

```
<%@ OutputCache Location="Server" Duration="30" VaryByParam="ddlDept" %>
```

Now run the page again and watch the difference in output where we will notice the data being cached based on the value selected under the DropDownList control.

Note: If in any situation if we want the output should be varying based on multiple parameters, we can even specify a semicolon separated list of values under “VaryByParam”.

Data Caching: it's a process of storing an amount of data into cache memory under a Web Form, which is very similar like storing data in “View State” or “Session State” or “Application State”.

Syntax of storing value into cache:

Cache[string key] = value (accepts value of type object)

Syntax of accessing values from cache:

Object value = Cache[string key]

Note: apart from the above we can also store the data by using “Cache.Add” and “Cache.Insert” methods also.

Storing data in Cache is very similar to storing data in “Application State” i.e., both are “multi-user global data”, whereas the difference between Application State and Cache are:

1. Application State is not Thread Safe, whereas Cache Memory is Thread Safe.
2. We can access Application State data in Global.asax file, whereas we can't access Cache Memory in Global.asax.
3. In Application State the data stays for a long time i.e., until the server restarts or explicitly we assign “null”, whereas data in Cache Memory will not stay for long time i.e., server can remove the values in Cache Memory at any point of time.
4. In case of Application State, data will not have any dependencies, whereas in case of Cache Memory, data can have 3 types of dependencies:
 - a) Time Based Dependency.
 - b) File Based Dependency.
 - c) SQL Based Dependency.

Time Based Dependency: in this case the values in Cache Memory will be removed based on a time i.e., when the time expires the data in the Cache Memory is deleted. This is again divided in to 2 types:

1. **Absolute Expiration:** In this case the data in the Cache Memory stays for a specified time which is calculated from the first time it was stored in the Cache and once the time elapses data in Cache Memory is removed.
2. **Sliding Expiration:** In this case the data in the Cache Memory stays for a specified time which is calculated from the last visit and once the time elapses data in Cache Memory is removed.

To test this, add a new Web Form under the project naming it as “DataCachingTimeBased.aspx”, place a GridView control on it and write the below code in “DataCachingTimeBased.aspx.cs” file.

```
using System.Data;
using System.Web.Caching;
using System.Data.SqlClient;


---

Declarations:
DataSet ds;


---

Code under Page_Load:
if (Cache["CustomerDS"] == null)
{
    SqlDataAdapter da = new SqlDataAdapter("Select * From Customer", ReadCS.ASPDB);
    ds = new DataSet();
    da.Fill(ds, "Customer");
    Cache.Insert("CustomerDS", ds);
    Response.Write("Data loaded from database server.");
}
else
{
    ds = (DataSet)Cache["CustomerDS"];
    Response.Write("Data loaded from cache memory");
}
GridView1.DataSource = ds;
GridView1.DataBind();
```

In the above case we haven't applied a dependency on the cache, so the values get stored in the Cache Memory until the Server explicitly removes it, whereas we can apply any of the 3 types of dependencies we have discussed above. Now let us implement Time Based Dependency for the above program:

Implementing Absolute Expiration: to implement this re-write the “Cache.Insert” method code in the above program as following:

```
Cache.Insert("CustomerDS", ds, null, DateTime.UtcNow.AddSeconds(30), Cache.NoSlidingExpiration);
```

In the above case because we have applied **Absolute Expiration** to **Cache**, data stays in the **Cache Memory** for 30 seconds from the point it was stored in the cache.

Implementing Sliding Expiration: to implement this re-write the “Cache.Insert” method code in the above program as following:

```
Cache.Insert("CustomerDS", ds, null, Cache.NoAbsoluteExpiration, TimeSpan.FromSeconds(30));
```

In the above case because we have applied **Sliding Expiration** to **Cache**, data stays in the **Cache Memory** for 30 seconds from the last visit to the page.

File Based Dependency: in this case the data in the **Cache** is dependent on some **File** and whenever the data in the file changes automatically the cache data gets flushed out. To use file based dependency we need to monitor the file present on the hard disk by using the class "**CacheDependency**" which is present under the namespace "**System.Web.Caching**" i.e., we need to create the instance of "**CacheDependency**" class by passing "**Filename**" which we want to monitor as a parameter to the Constructor and then "**CacheDependency**" class instance should be passed as a parameter to "**Cache.Insert**" method, so that whenever the data in file changes immediately it gets flushed out from the cache memory. To test this, add an XML File under the project naming it as "**Products.xml**" and write the below code in the file:

```
<Products>
  <Product>
    <Id>101</Id>
    <Name>Television</Name>
  </Product>
  <Product>
    <Id>102</Id>
    <Name>Microwave</Name>
  </Product>
  <Product>
    <Id>103</Id>
    <Name>Washing Machine</Name>
  </Product>
  <Product>
    <Id>104</Id>
    <Name>Refrigerator</Name>
  </Product>
  <Product>
    <Id>105</Id>
    <Name>Home Theatre</Name>
  </Product>
</Products>
```

Now add a new Web Form under the project naming it as "**DataCachingFileBased.aspx**", place a "**GridView**" control on it and write the below code under "**DataCachingFileBased.aspx.cs**" file.

```
using System.Data;
using System.Web.Caching;


---


Code Under Page Load Event:
DataSet ds;
if (Cache["ProductDS"] == null)
{
  ds = new DataSet();
```

```

ds.ReadXml(Server.MapPath("~/Products.xml"));
CacheDependency cd = new CacheDependency(Server.MapPath("~/Products.xml"));
Cache.Insert("ProductDS", ds, cd);
Response.Write("Data loaded from XML File.");
}
else
{
    ds = (DataSet)Cache["ProductDS"];
    Response.Write("Data loaded from Cache.");
}
GridView1.DataSource = ds.Tables[0];
GridView1.DataBind();

```

Run the above Web Form and watch the output and in this case data in **Cache Memory** will be staying in **Cache** until the file is modified and once the file is modified it will immediately flush the data from cache.

SQL Based Dependency: Flushing out data from the cache memory based on the changes made in the database table is known as “SQL Based Dependency” i.e., whenever the data in the table changes, automatically the data in the Cache is flushed out. To work with SQL Based Dependency we need to first perform 3 actions:

1. We need to enable SQL Cache Dependency on the database and table explicitly by using “aspnet_regsql” tool.
2. We need to specify the Cache Details under the config file.
3. We need to create the SQLCacheDependency class instance by specifying the Database Name and Table Name which should be passed as a parameter to the Insert method of Cache.

Step 1: open Visual Studio Developer Command Prompt and enable **Cache Dependency** to the **Database** and **Table** as following:

Enabling Cache Dependency for Database and Table:

SQL Authentication: aspnet_regsql -S Server -U Sa -P 123 -d ASPDB -ed -t Customer -et

Windows Authentication: aspnet_regsql -S Server -E -d ASPDB -ed -t Customer -et

Note: we can also disable the enabled Cache Dependency on Database with the following statement:

SQL Authentication: aspnet_regsql -S Server -U Sa -P 123 -d ASPDB -dd

Windows Authentication: aspnet_regsql -S Server -E -d ASPDB -dd

Step 2: open “Web.config” file and specify the Cache details under “**<system.web>**” tag as following:

```

<caching>
    <sqlCacheDependency enabled="true">
        <databases>
            <add name="ASPDB" connectionName="ConStr" pollTime="2000" />
        </databases>
    </sqlCacheDependency>
</caching>

```

Note: `pollTime` is an attribute to specify when IIS must contact the Database Engine for any change notifications and the default `pollTime` is 500 milliseconds.

Now add a new Web Form under the project naming it as “`DataCachingSQLBased.aspx`”, place a “`GridView`” control on it and write the below code under “`DataCachingSQLBased.aspx.cs`” file.

```
using System.Data;
using System.Web.Caching;
using System.Data.SqlClient;


---


Code under Page Load Event:
DataSet ds;
if (Cache["CustomerDS"] == null) {
    SqlDataAdapter da = new SqlDataAdapter("Select * From Customer", ReadCS.ASPDB);
    ds = new DataSet();
    da.Fill(ds, "Customer");
    SqlCacheDependency cd = new SqlCacheDependency("ASPDB", "Customer");
    Cache.Insert("CustomerDS", ds, cd);
    Response.Write("Data loaded from database server.");
}
else {
    ds = (DataSet)Cache["CustomerDS"];
    Response.Write("Data loaded from cache memory");
}
GridView1.DataSource = ds.Tables[0];
GridView1.DataBind();
```

Now any changes that are made in the Database to the table will flush the Data in cache and reloads it when the next request comes to the page.

Note: we can also explicitly remove values from Cache Memory by calling “`Cache.Remove`” method by passing the “`Key Name`” string as a parameter to the method.

E.g.: `Cache.Remove(string key)`