

# **Session-1**

**Introduction to .NET  
and  
.NET Framework**

# Contents

- Introduction to the .Net Framework
- Intermediate Language (IL)
- CLR and its functions
  - JIT Compilation
  - Garbage Collection and Memory Management
  - AppDomain Management
  - CLS, CTS
  - Security
- Assemblies and their types

# Introduction to .NET Framework

## What is .NET ?

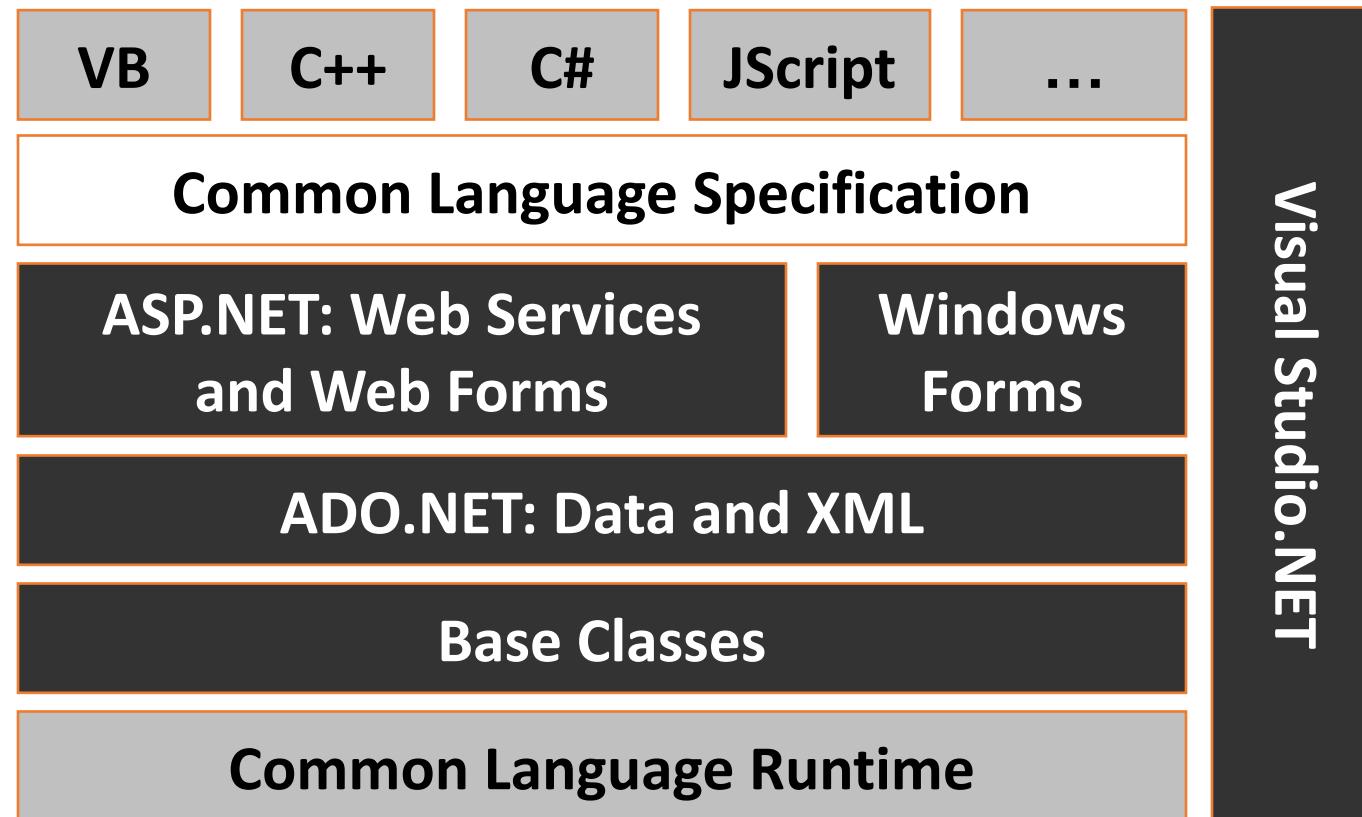
.NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications.

## What is .NET Framework?

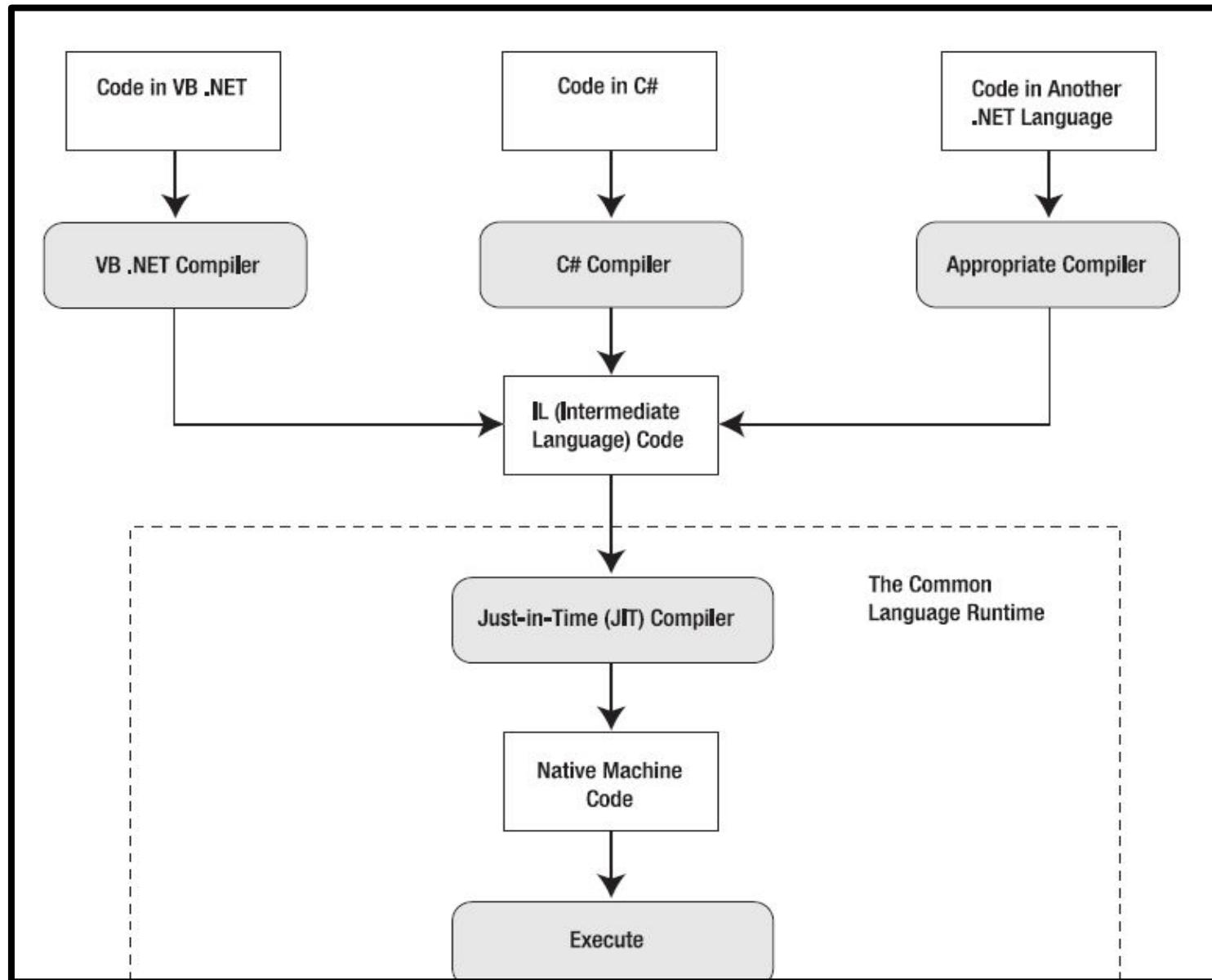
.NET Framework is the execution and development environment for .NET applications. It supports running websites, services, desktop apps, and more on Windows.

# Introduction to .NET Framework

## Architecture of .NET Framework



# Intermediate Language (IL)



# Intermediate Language (IL)

## Two stages of compilation:

1. VB or C# code you write is compiled into an intermediate language called **Microsoft Intermediate Language (MSIL)** code, or **Common Intermediate Language (CIL)**. The compiled file with IL code is an *assembly (.dll or .exe file)*
2. IL code is compiled into low-level native machine code. This stage is known as *just-in-time (JIT) compilation*

# Common Language Runtime(CLR)

- The heart of the .NET Framework is the CLR
- It is a runtime environment that executes MSIL code
- CLR is present in every .NET framework version

# Common Language Runtime(CLR)

## JIT Compilation

The .NET CLR utilizes Just In Time (JIT) compilation technology to convert the IL code back to a platform/device-specific code. Three types of JIT compilers:

1. **Pre-JIT** : This JIT compiles an assembly's entire code into native code at one stretch. You would normally use this at installation time.
2. **Econo-JIT** : You would use this JIT on devices with limited resources. It compiles the IL code bit-by-bit, freeing resources used by the cached native code when required.
3. **Normal JIT** : The default JIT compiles code only as it is called and places the resulting native code in the cache.

# Common Language Runtime(CLR)

## Garbage Collection and Memory Management

- The CLR implements dynamic memory management through the use of garbage collection. The **programmer is responsible for allocating memory**, but it is the **CLR that clears up unused memory**.
- Code that is run under the garbage collection system is known as **managed code**
- However, at times you may want to **force the garbage collector to run**, perhaps before starting an operation that is going to require a large amount of memory. To do this, just call ***GC.Collect()***

# Common Language Runtime(CLR)

## Application Domains (or AppDomain)

- This are like **lightweight processes**
- An AppDomain, which may contain one or more assemblies, is **completely isolated** from any other AppDomains running in the same process so there's **no sharing of memory or data**.
- In fact, the separation is so complete that another AppDomain running in the same process is treated in exactly the same way as one residing on another machine

# Common Language Runtime(CLR)

## Common Language Specification (CLS)

It is responsible for converting the different .NET programming language **syntactical rules** and **regulations** into CLR understandable format. In simple words, CLS enables cross-language integration or Interoperability.

## Common Type System (CTS)

Every programming language has its own data type system, so CTS is responsible for understanding **all** the **data type systems** of .NET programming languages and converting them into CLR understandable format which will be a **common format**.

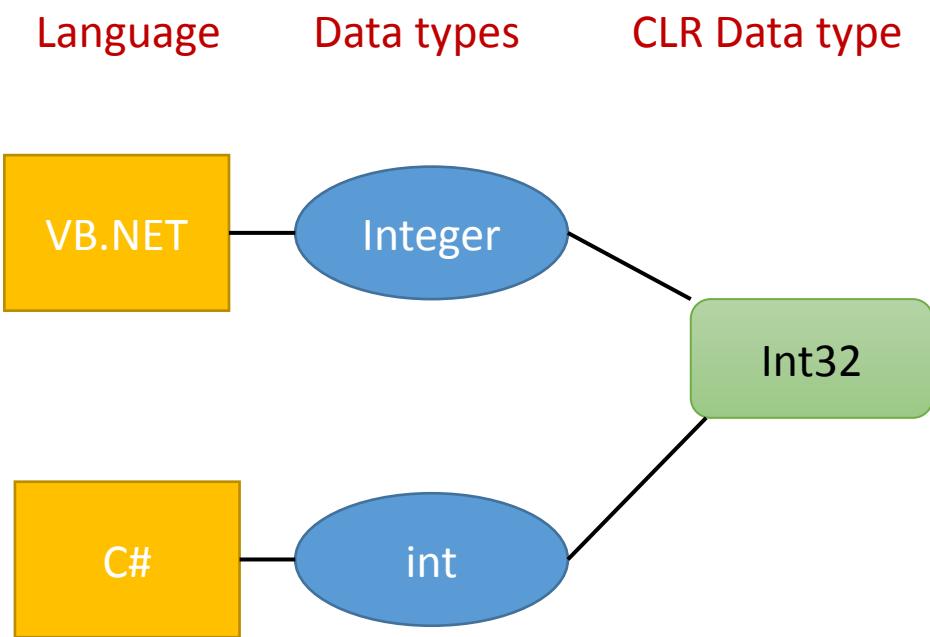


Figure : Common Type System

# Common Language Runtime(CLR)

## Security

- Security for .NET applications starts as soon as a **class is loaded** by the CLR.
- Before the class loader instantiates a class, security information—such as **accessibility rules** and **self-consistency requirements**—are **checked**.
- Essentially system of **security policies** that can be set by an administrator to allow certain levels of access based on the component's assembly information. The policies are set at three levels:
  - the enterprise
  - the individual machine
  - the user

# Assembly

- .NET applications are deployed as assemblies, which can be a **single executable** or a **collection of components** and form a **logical unit of functionality**
- Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications.
- When you create a .NET application, you are actually creating an assembly, which contains a **manifest** that **describes the assembly**.
- This **manifest data contains** the **assembly name**, **its versioning information**, **any assemblies referenced** by this assembly and their versions, a **listing of types in the assembly**, **security permissions**, its **product information** (company, trademark, and so on), and any **custom attributes**

# Assembly

## Types of Assemblies :

1. **Private** : requires us **to copy separately in all application folders** where we want to use that assembly's functionalities. Without copying, we cannot access the private assembly features and power. Private assembly means every time we have one, we exclusively copy into the BIN folder of each application folder.
2. **Shared** : **Only one copy is required in system level**, there is no need to copy the assembly into the application folder. Shared assembly should install in GAC. Shared assemblies (also called strong named assemblies) are copied to a single location (usually the Global assembly cache).

# Assembly

## Types of Assemblies :

3. **Satellite** : used for deploying language and culture-specific resources for an application.

# References

1. <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework>
2. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price

# **Session-2**

**Introduction to Visual Studio  
and  
.NET technologies**

# Contents

- Introduction to Visual Studio
- .NET Framework, .NET Code and Xamarin
- Managed and Unmanaged Code
- Using ILDASM utility

# Introduction to Visual Studio

Microsoft has a family of code editors and **Integrated Development Environments (IDEs)**, which include:

- Visual Studio Code
- Visual Studio 2019
- Visual Studio 2019 for Mac

# Introduction to Visual Studio

## Visual Studio Code

- The most modern and **lightweight code editor** and the only one from Microsoft that **is cross-platform**
- It is able to run on all common operating systems, including **Windows**, **macOS**, and many varieties of **Linux**, including **Red Hat Enterprise Linux (RHEL)** and **Ubuntu**.
- It has an extensive and **growing set of extensions** to **support many languages** beyond C#
- Using Visual Studio Code means a developer can use a **cross-platform code editor** to **develop cross-platform apps**.

# Introduction to Visual Studio

## Visual Studio 2019

- Microsoft Visual Studio 2019 only **runs on Windows**.
- You must run it on Windows 10 to create **Universal Windows Platform (UWP)** apps.
- It is the only Microsoft developer tool that **can create Windows apps**.

## Visual Studio 2019 for Mac

To create apps for the Apple operating systems like iOS to run on devices like iPhone and iPad, you must have **Xcode**, but that tool **only runs on macOS**.

- Although you **can use Visual Studio 2019 on Windows** with its **Xamarin extensions** to write a cross-platform mobile app, you still need macOS and Xcode to compile it.

# Differences .NET Framework, .NET Core and Xamarin

## .NET Framework

- .NET Framework is a **development platform** that includes a **Common Language Runtime (CLR)**, which **manages the execution of code**, and a **Base Class Library (BCL)**, which provides a **rich library of classes** to build applications.
- All of the apps on a computer written for the .NET Framework share the same version of the CLR and libraries stored in the **Global Assembly Cache (GAC)**, which can lead to issues if some of them need a specific version for compatibility.

# Differences .NET Framework, .NET Core and Xamarin

## .NET Core

- Today, we live in a truly **cross-platform world** where **modern mobile** and **cloud development** have made Windows, as an operating system, much less important.
- Because of that, Microsoft has been working on an **effort to decouple .NET** from its close ties with **Windows**.
- While rewriting .NET Framework to be truly cross-platform, they've taken the opportunity to **refactor** and **remove major parts** that are **no longer considered core**.
- This new product was branded **.NET Core** and includes a cross-platform implementation of the CLR known as **CoreCLR** and a streamlined library of classes known as **CoreFX**.

# Differences .NET Framework, .NET Core and Xamarin

## .NET Core

- .NET Core is **smaller than the current version** of .NET Framework due to the fact that legacy technologies have been removed.
- For example, **Windows Forms** and **Windows Presentation Foundation (WPF)** can be used to build graphical user interface (GUI) applications, but they are tightly bound to the **Windows ecosystem**, so they **have been removed** from .NET Core on macOS and Linux.
- One of the new features of .NET Core 3.0 is support for running old Windows Forms and WPF applications using the **Windows Desktop Pack**

# Differences .NET Framework, .NET Core and Xamarin

## .NET Core

- **ASP.NET Web Forms** and **Windows Communication Foundation (WCF)** are old web application and service technologies that fewer developers are choosing to use for new development projects today, so they **have also been removed** from .NET Core.
- Instead, developers prefer to use **ASP.NET MVC** and **ASP.NET Web API**. These two technologies have been refactored and combined into a new product that runs on .NET Core, named **ASP.NET Core**.

# Differences .NET Framework, .NET Core and Xamarin

## Xamarin

- Microsoft purchased Xamarin in 2016 and now gives away what used to be an expensive **Xamarin extension for free** with **Visual Studio 2019**.
- Microsoft renamed the **Xamarin Studio** development tool, which could only **create mobile apps**, to **Visual Studio for Mac** and gave it the ability to create other types of apps.

# Differences .NET Framework, .NET Core and Xamarin

We can summarize and compare .NET technologies, as shown in the following table:

Technology	Description	Host OS
.NET Core	for cross-platform and new apps and services	Windows, macOS, Linux
.NET Framework	for legacy apps	Windows only
Xamarin	Mobile apps only.	Android, iOS, macOS

# Managed Code and Unmanaged Code

## Managed Code:

- It is executed by managed **runtime environment CLR**.
- It provides **security** to the application written in .NET Framework.
- **Memory buffer overflow does not** occur.
- It **provide runtime services** like **Garbage Collection, exception handling**, etc.
- It also provides **reference checking** which means it checks whether the **reference point** to the **valid object** or not
- The application is written in the languages like C#, VB.Net, etc. are always managed code.

# Managed Code and Unmanaged Code

## Unmanaged Code:

- It is executed directly by the operating system.
- It does not provide security to the application.
- In unmanaged code, the memory allocation, type safety, security etc are managed by the developer. Due to this, there are several problems related to memory occur like buffer overflow, memory leak etc.
- It provides the low-level access to the programmer.
- The application written in VB 6.0, C, C++, etc are always in unmanaged code

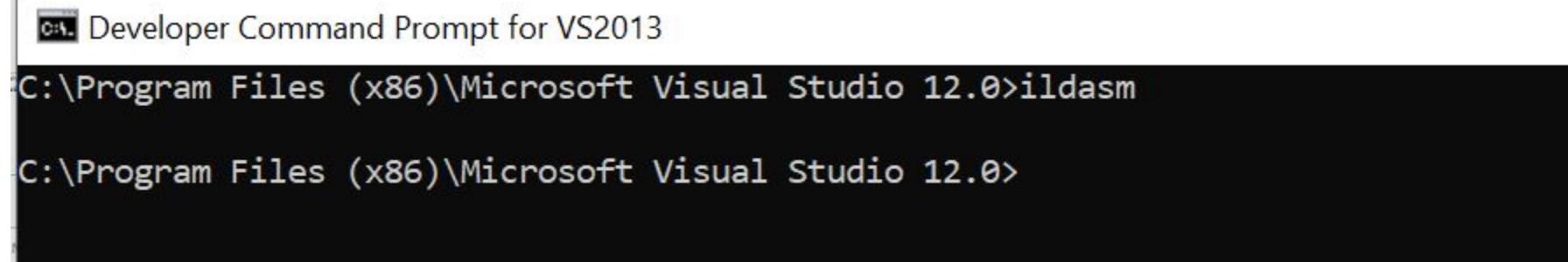
# ILDASM Utility

- The **ILDASM** (Intermediate Language Disassembler) is an utility which shows the **information of Assembly** in **human-readable format** by parsing any .NET Framework DLL.
- It also shows the **namespaces** and **types** and as well as their **interfaces**.
- The advantage of ildasm.exe utility is that it can **examine some native assemblies** like such as **Mscorlib.dll**, as well as **.NET Framework assemblies** provided by others or created by yourself.

# ILDASM Utility

## Steps to see the use of ildasm utility in C#:

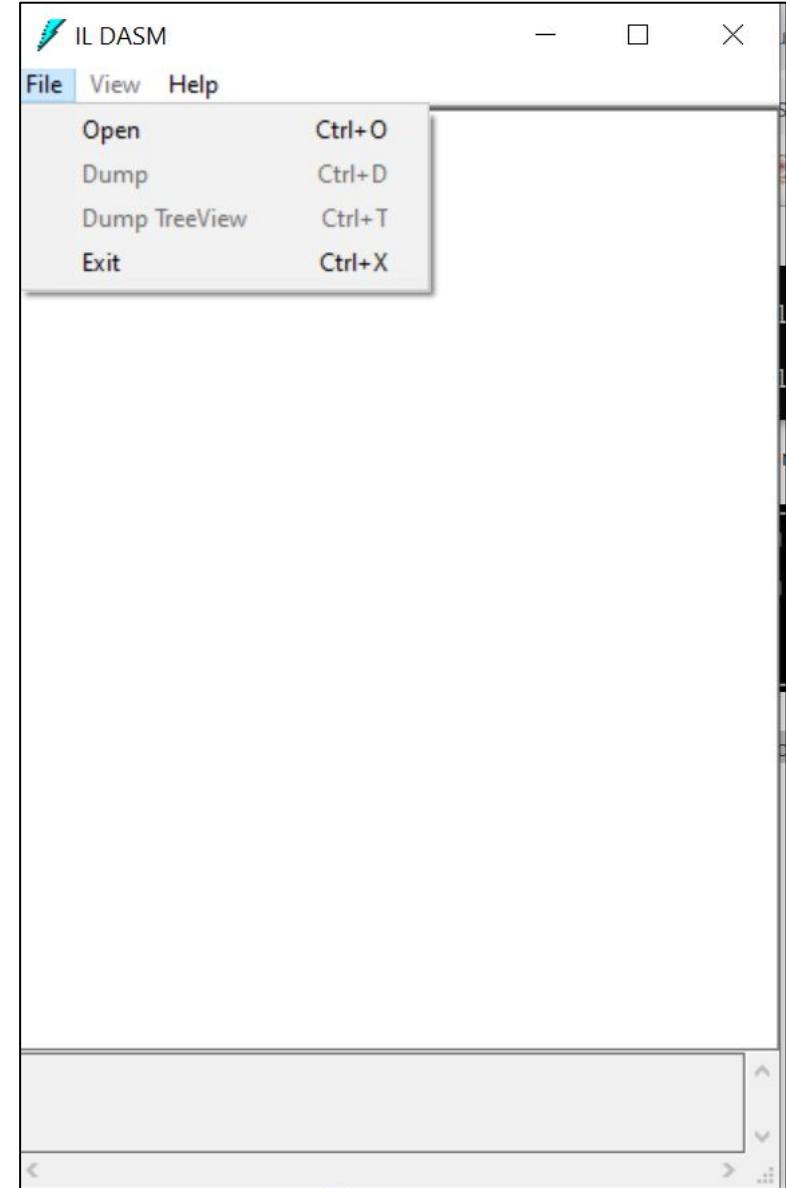
1. Firstly open the Visual Studio 2013 Command Prompt and write the command like this.



```
Developer Command Prompt for VS2013
C:\Program Files (x86)\Microsoft Visual Studio 12.0>ildasm
C:\Program Files (x86)\Microsoft Visual Studio 12.0>
```

# ILDASM Utility

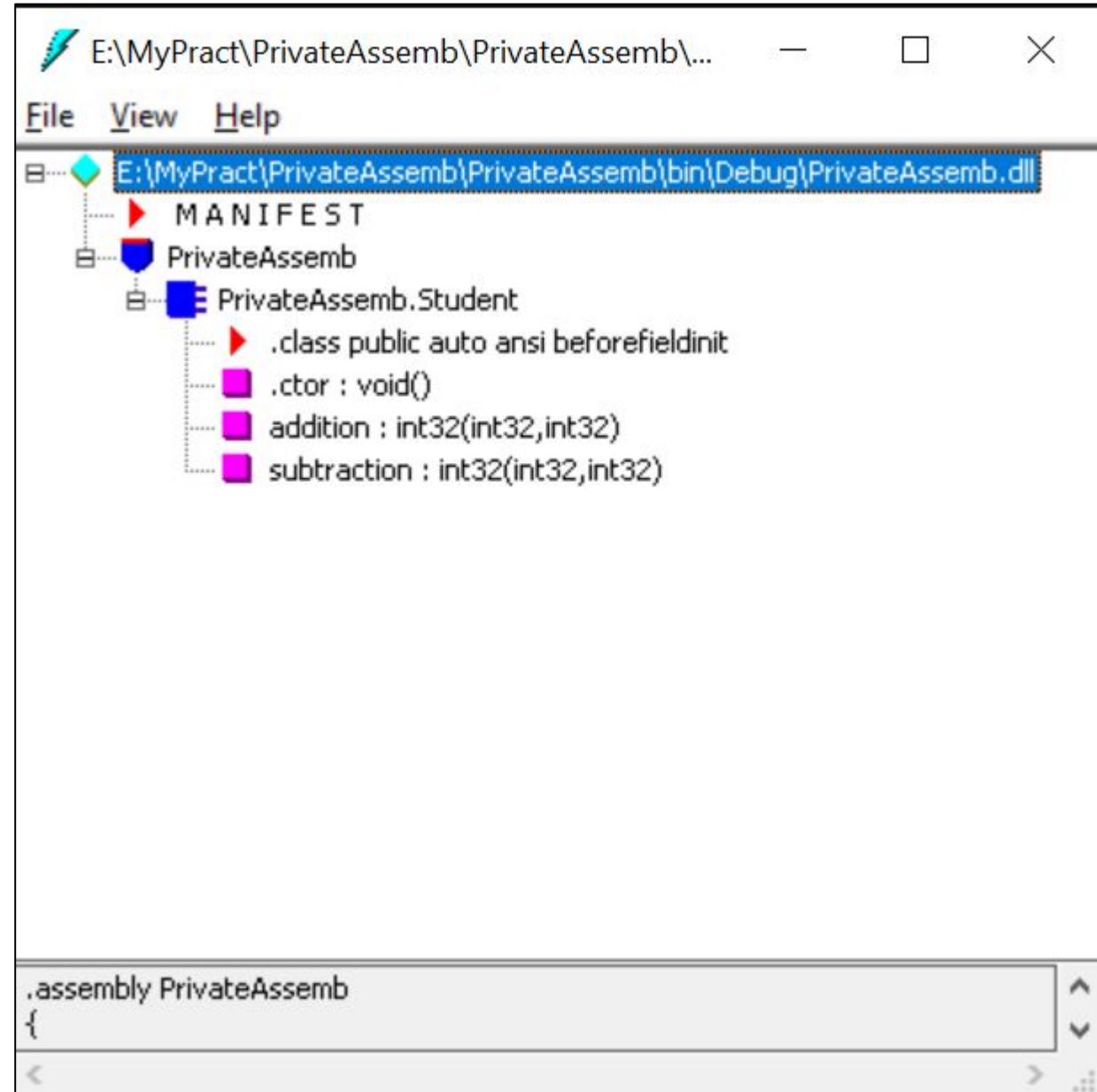
2. Then a ildasm window will open. In it select the File menu and click on open menu option.



# ILDASM Utility

3. Then go to the location of the assembly and select the assembly to open it.

4. A window will be opened which shows the information of the assembly.



# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-3**

## **Introduction to C# Basics**

# Contents

- Console Applications and Class Libraries (Demo)
- C# Basics
- Data types and CTS equivalents
- Access Specifiers
- Project References, using Classes (Demo)
- Methods
  - Method overloading
  - Optional parameters
  - Named parameters and positional parameters
  - Using params
- Properties
- Constructors and Destructors
- Object Initializers

# C# Basics

## What is C#?

- C# (pronounced "C sharp") is a **programming language** that is designed for building a variety of applications that run on the .NET Framework.
- C# is **simple, powerful, type-safe**(No uninitialized variables, unsafe casts), and **object-oriented**.
- The first “**Component oriented**” language in the C/C++ family
- **Everything is an object**

# C# Basics

## C# - Programming Structure

- Namespace declaration
- A class or structs or interfaces or enums, or delegates
- Members : Constants, fields, methods, properties, indexers, events, operators, constructors, destructors
- Statements & Expressions
- Comments

# C# Basics

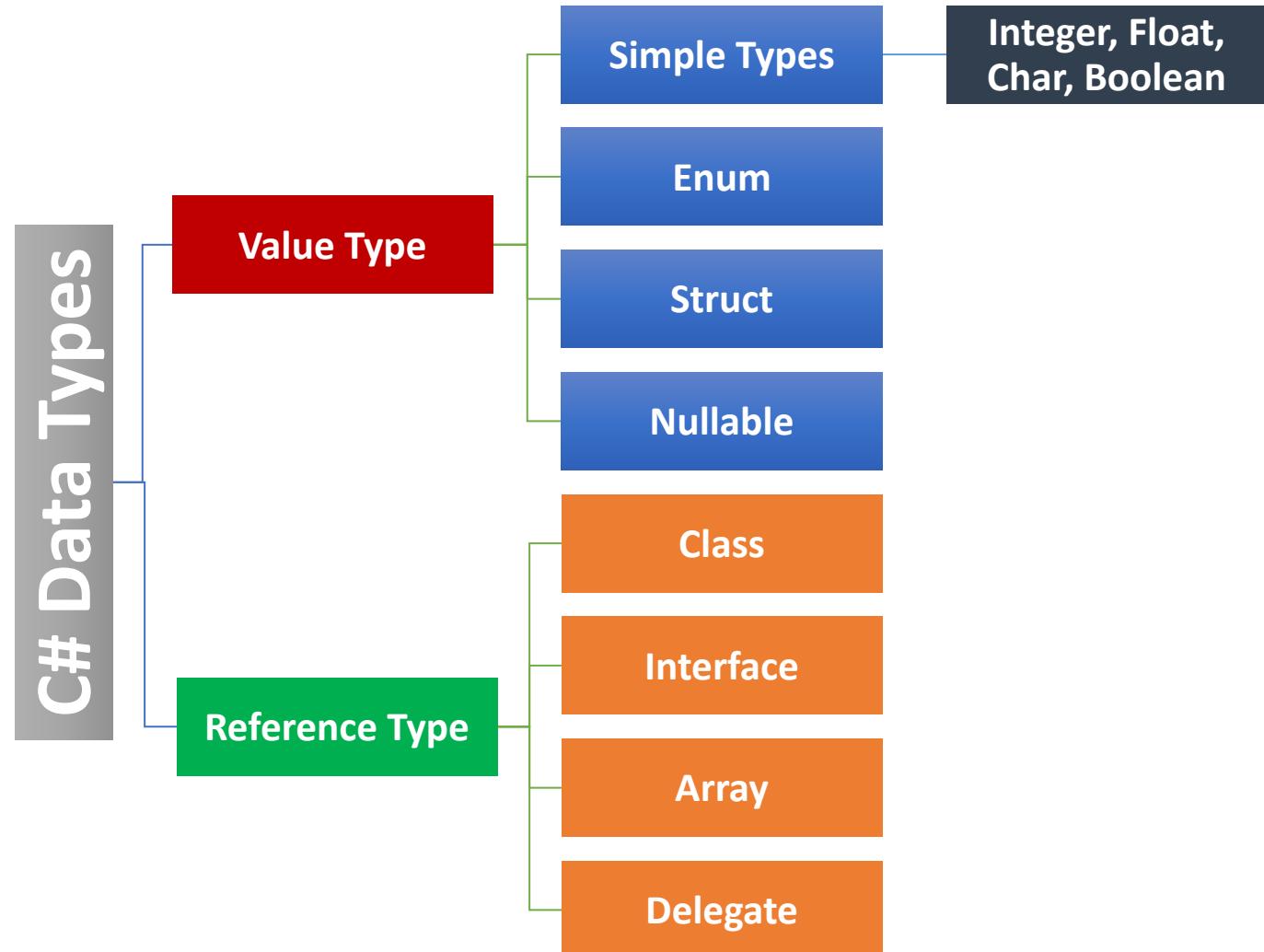
## C# - Sample Program

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

            /* my first program in C# */

            Console.WriteLine("Hello world");
            Console.Read();
        }
    }
}
```

# Data Types and CTS Equivalent



# Data Types and CTS Equivalent

## Predefined Data Types

Type	Description	Range	.NET Type
byte	8-bit unsigned integer	0 to 255	System.Byte
sbyte	8-bit signed integer	-128 to 127	System.SByte
short	16-bit signed integer	-32,768 to 32,767	System.Int16
ushort	16-bit unsigned integer	0 to 65,535	System.UInt16
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647	System.Int32
uint	32-bit unsigned integer	0 to 4,294,967,295	System.UInt32

# Data Types and CTS Equivalent

## Predefined Data Types

Type	Description	Range	.NET Type
long	64-bit signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	System.Int64
ulong	64-bit unsigned integer	0 to 18,446,744,073,709,551,615	System.UInt64
float	32-bit Single-precision floating point type	-3.402823e38 to 3.402823e38	System.Single
double	64-bit double-precision floating point type	-1.79769313486232e308 to 1.79769313486232e308	System.Double
char	16-bit single Unicode character	Any valid character, e.g. a, b, *	System.Char

# Data Types and CTS Equivalent

## Predefined Data Types

Type	Description	Range	.NET Type
bool	8-bit logical true/false value	True or False	System.Boolean
object	Base type of all other types.		System.Object
string	A sequence of Unicode characters		System.String
DateTime	Represents date and time	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999	System.DateTime

# Data Types and CTS Equivalent

- **Boxing** : Value type is converted to object type, it is called **boxing**.  
Allocates box, copies value into it.

```
object obj;  
obj = 100; // this is boxing
```

- **Unboxing** : when an Object type is converted to a value type, it is called **unboxing**. Checks type of box, copies value out

```
int i = 123;  
object obj = i; //boxing  
int j = (int)obj; //unboxing
```

# Access Specifiers

- Encapsulation, in object oriented programming methodology, **prevents access** to implementation details.
- **Abstraction** allows making **relevant information visible** and **encapsulation** enables a programmer to **implement the desired level of abstraction**.
- C# supports the following access specifiers:
  - **Public**
  - **Private**
  - **Protected**
  - **Internal**
  - **Protected internal**

# Access Specifiers

## Internal :

- allows a class to expose its member variables and member functions to other functions and objects in the current assembly.
- In other words, internal members can be accessed from any class or method defined within the application in which the member is defined.

## Protected Internal:

The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

# Methods

- **Methods** are members of a type that **execute a block of statements**.
- Methods can **return a single value** or **return nothing**.
  - A method that performs some actions but does not return a value indicates this with the **void type** before the name of the method.
  - A method that performs some actions and returns a value indicates this with the **type of the return value** before the name of the method.

# Methods

## Elements of Method:

- **Access Specifier** – This determines the visibility of a variable or a method from another class.
- **Return type** – A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
- **Method name** – Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter list** – Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method body** – This contains the set of instructions needed to complete the required activity.

# Methods

## Example of Method:

```
public int calculateFactorial(int num)
{
    int factorial = 1;
    for (int i = 1; i <=num; i++)
    {
        factorial=factorial*i;
    }
    return factorial;
}
```

# Methods

## Method Overloading

- Instead of having two different method names, we could give both **methods the same name**. This is allowed because the methods each have a **different signature**.
- A **method signature** is a list of parameter types that can be passed when calling the method (as well as the type of the return value).

# Methods

## Method Overloading

```
public int addNumbers(int x,int y)
{
    return (x+y);
}

1 reference
public int addNumbers(int x, int y,int z)
{
    return (x + y + z);
}
```

# Methods

## Method Overloading

```
static void Main(string [] args)
{
    Program obj = new Program();
    int result1 = obj.addNumbers(12, 21); //calling function with 2 parameters
    int result2= obj.addNumbers(11,22,33); //callling function with 3 parameters
    Console.WriteLine("Result-1: " + result1);
    Console.WriteLine("Result-2: " + result2);
}
```

# Methods

**Optional Parameters** are not compulsory parameters, they are optional. It helps to exclude arguments for some parameters. Or we can say in optional parameters, it is not necessary to pass all the parameters in the method

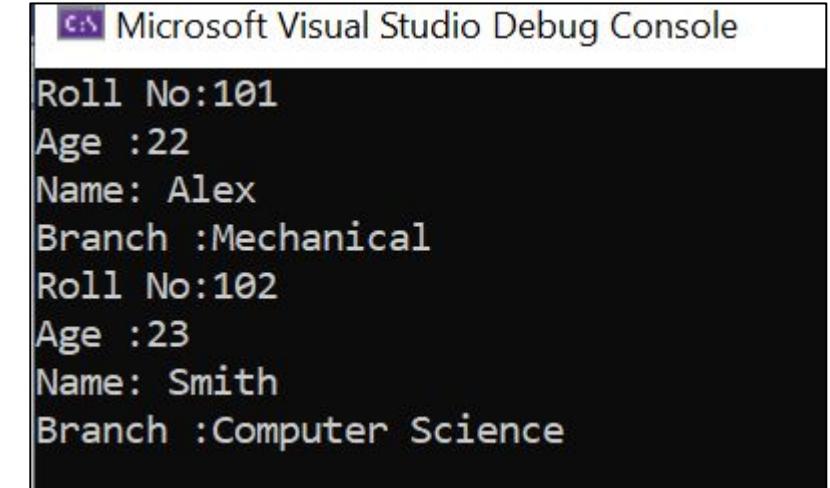
# Methods

## Optional Parameters

```
class Program
{ // This method contains three regular
// parameters, i.e. roll,age and name
// And one optional parameters branch.

    2 references
    public void displayStudentDetails(int roll,int age,string name,string branch="Mechanical")
    {
        Console.WriteLine("Roll No:" + roll);
        Console.WriteLine("Age :" + age);
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Branch :" + branch);
    }

    0 references
    static void Main(string [] args)
    {
        Program obj = new Program();
        obj.displayStudentDetails(101, 22, "Alex");
        obj.displayStudentDetails(102, 23, "Smith", "Computer Science");
    }
}
```



```
Microsoft Visual Studio Debug Console
Roll No:101
Age :22
Name: Alex
Branch :Mechanical
Roll No:102
Age :23
Name: Smith
Branch :Computer Science
```

# Methods

## Named and positional parameters

- Generally, while calling the method we need to pass all the method arguments in the **same sequence of parameters** in the method definition.
- If we use named parameters, **we don't need to worry about the order or sequence of parameters** while calling the method.
- In named parameters, the **parameter values will map to the right parameter based on the parameter name**. So, while calling the methods, we can **send the parameters in any sequence or order**

# Methods

## Named and positional parameters

```
class Program
{
    1 reference
    public void displayStudentDetails(int roll,int age,string name,string branch)
    {
        Console.WriteLine("Roll No:" + roll);
        Console.WriteLine("Age :" + age);
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Branch :" + branch);
    }

    0 references
    static void Main(string [] args)
    {
        Program obj = new Program();
        obj.displayStudentDetails(age:23,name:"Ken",branch:"Civil",roll:101);
    }
}
```

Sending the  
parameters in any  
sequence or order

# Methods

**params** : It is used as a parameter which can take the **variable number of arguments**.

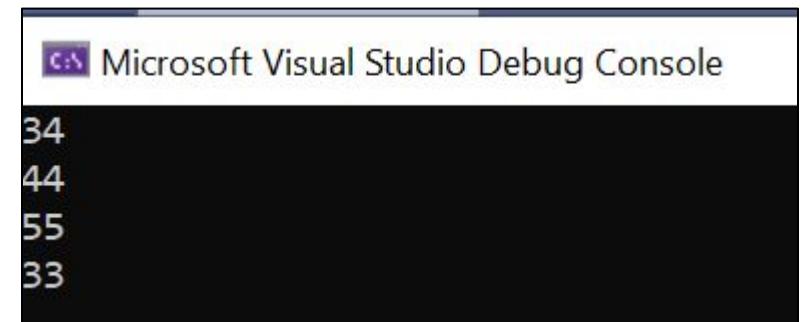
- It is useful when programmer **don't have any prior** knowledge about the **number of parameters** to be used.
- Only **one params keyword** is allowed and no additional params will be allowed in function declaration after a params keyword.
- The **length of params will be zero if no arguments** will be passed.

# Methods

**params :**

```
class Program
{
    1 reference
    public void displayStudentMarks(params int [] marks)
    {
        for(int i = 0; i < marks.Length; i++)
            Console.WriteLine(marks[i]);
    }

    0 references
    static void Main(string [] args)
    {
        Program obj = new Program();
        obj.displayStudentMarks(34, 44, 55, 33);
    }
}
```



# Properties

- Property in C# is a member of a class that provides a flexible mechanism for classes **to expose private fields**.
- Internally, C# properties are special methods called **accessors**. A C# property have two accessors,
  - **get** property accessor : *returns a property value*
  - **set** property accessor : *assigns new value*
- The **value** keyword represents the **value of a property**.
- Properties can be read-write, read-only, or write-only. The **read-write** property implements **both get and set accessor**. A **write-only** property implements a **set accessor**, but no get accessor. A **read-only** property implements a **get accessor**, but no set accessor.

# Constructors and Destructors

- **Constructor** is a special method which is **invoked automatically** at the **time of object creation**. It is used to initialize the data members of new object generally. The constructor in C# has the **same name** as **class or struct**.
- **Destructors** in C# are methods inside the class used to destroy instances of that class when they are no longer needed. The Destructor is **called implicitly** by the .NET Framework's Garbage collector and therefore programmer has no control as when to invoke the destructor.

# Constructors and Destructors

- There can be five types of constructors in C#.
  - **Default Constructor** – It has no arguments and It is invoked at the time of creating object.
  - **Parameterized Constructor** - A constructor which has parameters is called parameterized constructor.
  - **Copy Constructor** - used to copy one object's data into another object
  - **Static Constructor** –There is no matter how many numbers instances (objects) of the class are created, static constructor is going to be invoked only once and that is when the class is load for the first time. The static constructor is used to initialize the static fields of the class.
  - **Private Constructor** - When a class contains a private constructor then we cannot create an object for the class outside of the class. So, private constructors are used to creating an object for the class within the same class

# Object Initializer

In object initializer, you can **initialize the value to the fields or properties** of a class at the time of creating object **without calling constructor**.

```
class Program
{
    0 references
    static void Main(string [] args)
    {
        Emp emp = new Emp()
        {
            EmpID = 10,
            EmpName = "Alex"
        };
        Console.WriteLine(emp.EmpID);
        Console.WriteLine(emp.EmpName);
    }
}
```

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price

# **Session-4**

**Static keyword and  
Inheritance**

# Contents

- Static members of the class
  - Fields/Variables
  - Methods
  - Properties
  - Constructors
- Static Class
- Inheritance
  - Access Specifiers
  - Constructors in Hierarchy
  - Overloading in derived class
  - Override
  - Sealed class and methods
  - Abstract class and methods

# Static Members

## Static variables

- Single copy(instance) of the variable is created and shared among multiple objects.
- Static variables are accessed with the name of the class, they do not require any object for access.

```
namespace FirstConsoleApp
{
    0 references
    class Program
    {
        static int x = 1; //static variable
        0 references
        static void Main(string [] args)
        {
            Console.WriteLine(x);
        }
    }
}
```

# Static Members

## Static Methods

- Static methods are accessed with the name of the class.
- A static method can access static and non-static fields, static fields are directly accessed by the static method without class name whereas non-static fields require objects.

```
namespace FirstConsoleApp
{
    class Program
    {
        static int x = 1; //static variable
        int y=2;//non-static variable
        static void count()//static method
        {
            Console.WriteLine(x++);
        }
        static void Main(string [] args)
        {
            Program.count();//calling static method
            Program obj = new Program();
            Console.WriteLine(obj.y);//Accessing non-static variable
        }
    }
}
```

# Static Members

## Static Properties

- Similar to static methods
- Static properties use the same get and set tokens as instance properties

```
namespace FirstConsoleApp
{
    class Program
    {
        static int empID;

        public static int EmpID { get => empID; set => empID = value; }

        static void Main(string [] args)
        {
            Program.EmpID = 101; //Accessing static properties
            Console.WriteLine(Program.EmpID);
        }
    }
}
```

# Static Members

## Static Constructor

- There is no matter how many numbers instances (objects) of the class are created, static constructor is going to be invoked only once and that is when the class is load for the first time.
- The static constructor is used to initialize the static fields of the class.

The screenshot shows the code for a C# program named `FirstConsoleApp`. The `Program` class contains a static constructor that outputs "Static Constructor" and a default constructor that outputs "Default Constructor". In the `Main` method, two objects are created, both outputting "Default Constructor".

```
namespace FirstConsoleApp
{
    class Program
    {
        static Program()
        {
            Console.WriteLine("Static Constructor");
        }

        Program()
        {
            Console.WriteLine("Default Constructor");
        }

        static void Main(string [] args)
        {
            Program obj=new Program();
            Program obj1=new Program();
        }
    }
}
```

Microsoft Visual Studio Debug Console

cs	Static Constructor
	Default Constructor
	Default Constructor

# Static Class

- A static class can only contain static data members, static methods, and a static constructor.
- It is not allowed to create objects of the static class.
- Static classes are **sealed**, means ***you cannot inherit a static class from another class.***

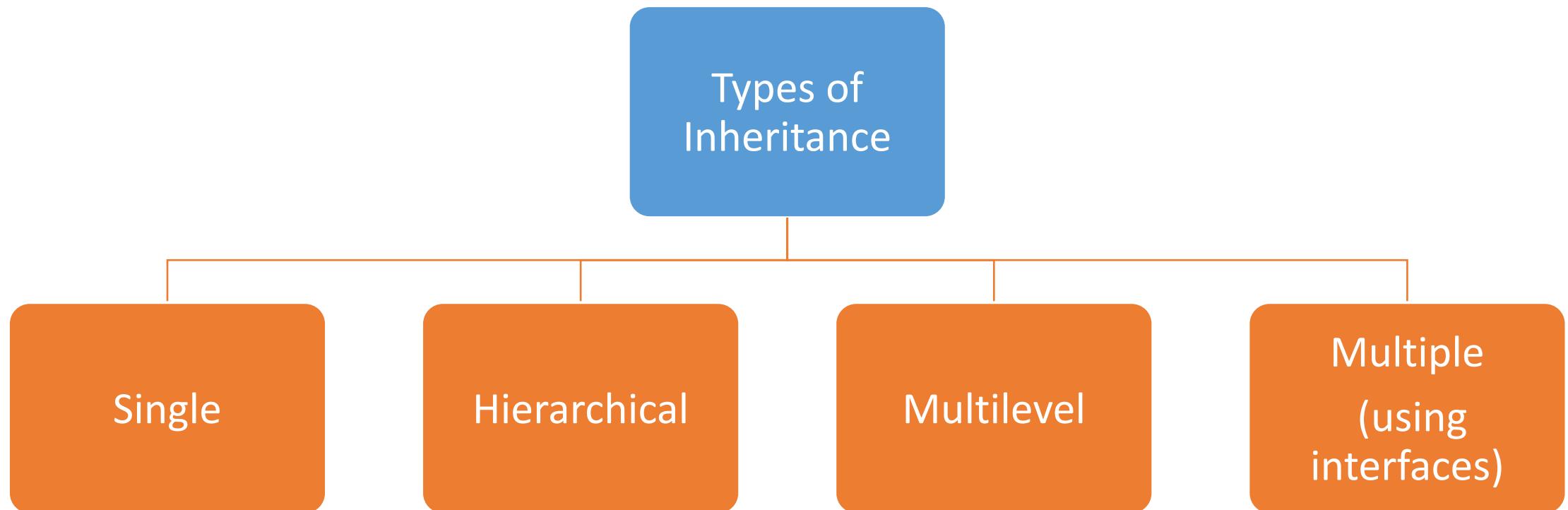
```
3 references
static class Program
{
    0 references
    static Program()
    {
        Console.WriteLine("Static Constructor");
    }
    1 reference
    public static void addNumbers(int x,int y)
    {
        Console.WriteLine("Addition : "+(x + y));
    }
}
0 references
class Course:Program
{
    0 references
    static void Main(string[] args)
    {
        Program.addNumbers(12, 21);
    }
}
```

 *cannot derived from static class*

# Inheritance

- It is the mechanism in C# by which one class (child or derived) is allowed to **inherit the features**(fields and methods) of another class (parent or base).
- **Inheritance** is one of the primary concepts of object-oriented programming (OOP).
- Inheritance supports the concept of “**reusability**”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

# Inheritance



# Inheritance

## Access Specifiers

	public	protected	internal	protected internal	private	private protected
<i>Entire program</i>	Yes	No	No	No	No	No
<i>Containing class</i>	Yes	Yes	Yes	Yes	Yes	Yes
<i>Current assembly</i>	Yes	No	Yes	Yes	No	No
<i>Derived types</i>	Yes	Yes	No	Yes	No	No
<i>Derived types within current assembly</i>	Yes	Yes	Yes	Yes	No	Yes

# Inheritance

## Constructors in Hierarchy :

**Two different cases arise as follows:**

**Case 1:** Only derived class contains a constructor. So the objects of the derived class are instantiated by that constructor and the objects of the base class are instantiated automatically by the default constructor.

**Case 2:** In this case, both the base class and derived class has their own constructors, so the process is complicated because the constructors of both classes must be executed. To overcome this situation C# provide a keyword known as a **base** keyword. With the help of base keyword, the derived class can call the constructor which is defined in its base class.

# Inheritance

## Overloading

- When the compiler goes looking for instance method overloads, it considers the compile-time class of the "target" of the call, and looks at methods declared there.
- If it can't find anything suitable, it then looks at the parent class then the grandparent class, etc.
- This means that if there are two methods at different levels of the hierarchy, the "deeper" one will be chosen first

# Inheritance

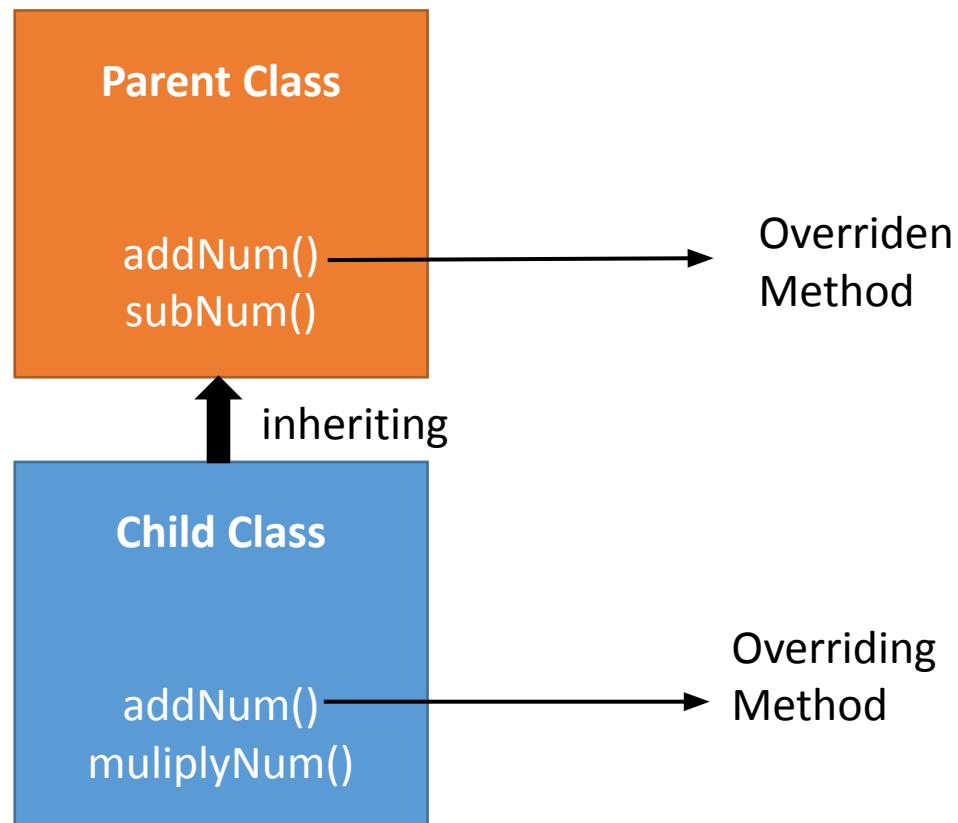
## Overriding:

- Overriding is a feature that allows a **subclass** or child class **to provide a specific implementation of a method** that is **already provided** by one of its **super-classes** or parent classes.
- When a method in a subclass has the **same name**, **same parameters** or signature and **same return type**(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.
- Method overriding is one of the ways by which C# achieve **Run Time Polymorphism(Dynamic Polymorphism)**.

# Inheritance

## Overriding:

- The overridden base method must be virtual, abstract, or override.



# Inheritance

## Overriding:

In C# we can use 2 types of keywords for Method Overriding:

- **virtual keyword:** This modifier or keyword **use within base class method.** It is used to modify a method in *base class* for *overridden* that particular method in the derived class.
- **override:** This modifier or keyword **use with derived class method.** It is used to modify a *virtual* or *abstract* method into *derived class* which presents in base class.

# Inheritance

## Sealed Class

- Sealed classes are used to restrict the users from inheriting the class.  
A class can be sealed by using the ***sealed*** keyword.
- No class can be derived from a sealed class.

## Sealed Method

- A *method can also be sealed*, and in that case, the method cannot be overridden.
- If you want to declare a method as sealed, then it has to be declared as **virtual** in its base class.

# Inheritance

## Abstract Class

- An Abstract class is never intended to be instantiated directly.
- This class must contain at least one *abstract method*, which is marked by the keyword or modifier *abstract* in the class definition.
- The Abstract classes are typically used to define a base class in the *class hierarchy*

## Abstract Method

- A method which is declared abstract, has no “body” and declared inside the abstract class only.
- An abstract method must be implemented in all non-abstract classes using the *override* keyword

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-5**

## **Interfaces and Operator Overloading**

# Contents

- Interfaces
  - Implementing interfaces
  - Explicitly implementing interfaces
  - Inheritance in interfaces
  - Default interfaces methods
  - IDisposable and IComparable
- Operator Overloading

# Interface

- It is like abstract class because **all the methods** which are **declared** inside the interface are **abstract methods**. It cannot have method body but in C# 8.0 this feature has also been implemented using **Default interface methods**.
- They cannot be instantiated.
- It is used to **achieve multiple inheritance** which can't be achieved by class. It is used to **achieve fully abstraction** because it cannot have method body.
- Its **implementation** must be provided by **class** or **struct**. The class or struct which implements the interface, **must provide** the **implementation** of **all the methods** declared **inside** the interface.

# Interface

- Interface methods are **public** and **abstract** by default. You **cannot explicitly** use public and abstract keywords for an interface method.

```
namespace FirstConsoleApp
{
    2 references
    internal interface ICalculate
    {
        2 references
        double calculateArea(float r);
        2 references
        double calculateArea(int l, int b);
    }
}
```

# Implementing Interface

```
namespace FirstConsoleApp
{
    internal class Shape : ICalculate
    {
        public double calculateArea(float r)
        {
            return (3.14f * r * r);
        }

        public double calculateArea(int l, int b)
        {
            return (l*b);
        }
    }
}
```

```
namespace FirstConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            ICalculate obj = new Shape();
            Console.WriteLine(obj.calculateArea(2.3f));
            Console.WriteLine( obj.calculateArea(3, 4));
        }
    }
}
```

# Explicitly Implementing Interface

- Using *<InterfaceName>.<MemberName>*
- Explicit implementation is useful when **class** is implementing multiple interfaces
- It is also useful if **interfaces** have the **same method name** coincidentally.
- Do not use **public** modifier with an explicit implementation. It will give a compile-time error.

# Explicitly Implementing Interface

```
namespace FirstConsoleApp
{
    1 reference
    internal class Shape : ICalShapePerimer
    {
        2 references
        void ICalShapePerimer.calculatePerimeter(int l, int w)
        {
            Console.WriteLine("Perimeter of Rec:" + (2 * (l + w)));
        }
        2 references
        void ICalShapePerimer.calculateCircumference(int r)
        {
            Console.WriteLine("Circumference of Circle:" + (2 * 3.14f * r));
        }
    }
}
```

```
namespace FirstConsoleApp
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            ICalShapePerimer p = new Shape();
            p.calculatePerimeter(2, 3);
            p.calculateCircumference(2);
        }
    }
}
```

# Inheritance in Interface

- C# allows the user to **inherit one interface** into another interface.
- When a class implements the inherited interface then it must provide the **implementation of all the members** that are defined within the **interface inheritance chain** (i.e including the base interface methods) Otherwise, the compiler throws an error.
- If both derived interface and base interface declares the same member then the **base interface member** name is **hidden** by the **derived interface member** name.

# Inheritance in Interface

```
namespace FirstConsoleApp
{
    1 reference
    interface A
    {
        2 references
        void showA();
    }
    1 reference
    interface B : A
    {
        2 references
        void showB();
    }
}
```

```
2 references
class Program:B
{
    2 references
    public void showA()
    {
        Console.WriteLine("Interface A Method");
    }
    2 references
    public void showB()
    {
        Console.WriteLine("Interface B Method");
    }
    0 references
    static void Main(string[] args)
    {
        Program obj= new Program();
        obj.showA();
        obj.showB();
    }
}
```

# Default Interface Methods

- C# 8.0 is allowed to **add members** as well as **their implementation** to the interface.
- Now you are allowed to **add a method with their implementation** to the **interface** without breaking the existing implementation of the interface, such type of methods is known as **default interface methods(also known as the virtual extension methods)**.
- You are allowed to implement indexer, property, or event accessor in the interface.
- You are allowed to use access modifiers like private, protected, internal, public, virtual, abstract, override, sealed, static, extern with default methods, properties, etc. in the interface.

# Default Interface Methods

- You are allowed to create static fields, methods, properties and events in the interface.
- The explicit access modifiers with default access are **public**.
- If an interface contains **default method** and **inherited** by some specified class, then the **class does not know anything about the existence of the default methods** of that interface and also does not contain the implementation of the default method.
- You are allowed to use the same name methods in the interface, but they must have different parameter lists

# Default Interface Methods

```
namespace FirstConsoleApp
{
    2 references
    interface A
    {
        2 references
        void showA();
        1 reference
        public void addNum(int num)//Default interface method
        {
            Console.WriteLine(num);
        }
    }
}
```

```
1           3 references
class Program:A
{
    2 references
    public void showA()
    {
        Console.WriteLine("Interface A Method");
    }
}
0 references
static void Main(string[] args)
{
    Program obj= new Program();
    obj.showA();

    A obj2 = new Program();
    obj2.addNum(22);
}
```

# IDisposable Interfaces

- When the garbage collector runs it can **clean up your managed resources**.  
The garbage collector **does not know how to free unmanaged resources** (such as file handles, network connections and database connections).
- The term "**unmanaged resource**" is usually used to describe something ***not directly under the control of the garbage collector***
- The following are two mechanisms to automate the freeing of unmanaged resources:
  1. Declaring a **destructor (or Finalizer)** as a member of your class.
  2. Implementing the **System.IDisposable** interface in your class.

# IDisposable Interfaces

- **IDisposable** is an interface that contains **only a single method** i.e. **Dispose()**, for **releasing unmanaged resources**.
- **IDisposable** is defined in the **System** namespace.
- When your application or class library encapsulates unmanaged resources such as **files, streams, database connections**, etc, they should implement the **IDisposable** interface
- Inside the **Dispose()** method we can perform the **resources clean-up**
- If you are using a class that implements the **IDisposable** interface, you should call its **Dispose()** method when you are finished using the class by **try/finally block** or **using statement**.
- To prevent the dispose method from running twice (in case the object already has been disposed) , we add **GC.SuppressFinalize(this)**

# IComparable Interface

- Use the IComparable Interface in C# to sort elements. It is also used to compare the current instance with another object of same type.
- It provides you with a method of comparing two objects of a particular type. Remember, while implementing the IComparable interface, **CompareTo()** method should also be implemented.

# IComparable Interface

```
namespace TestInterfaces
{
    6 references
    class Program:IComparable<Program>
    {
        int salary;
        4 references
        public int Salary { get => salary; set => salary = value; }

        0 references
        static void Main(string[] args)
        {
            Program program=new Program();
            program.Salary = 3400;

            Program program2 = new Program();
            program2.Salary = 3400;

            int retval=program.CompareTo(program2);
            Console.WriteLine(retval);
        }
    }
}

// if first obj val is greater than other then 1
// if first obj val is less than other then -1
// if first obj val is equal to other then 0
1 reference
public int CompareTo(Program obj)
{
    return this.Salary.CompareTo(obj.Salary);
}
```

# Operator Overloading

- Ability to use the same operator to do various operations by **providing special meaning to C# operators.**
- It provides **additional capabilities to C# operators** when they are applied to user-defined data types.
- It enables to make user-defined implementations of various operations where one or both of the operands are of a user-defined class.
- Only the **predefined set of C# operators** can be **overloaded**
- An operator can be overloaded by defining a function to it. The function of the operator is declared by using the **operator keyword**.

# Operator Overloading

## *Syntax:*

```
AccessSpecifier Class_Name operator Operator_symbol (parameters)  
{  
    //Code  
}
```

*Example :*

```
public static Calculator operator + (Calculator obj1,Calculator obj2)  
{  
    Calculator obj3 = new Calculator();  
    obj3.number = obj2.number + obj1.number;  
    return obj3;  
}
```

# Operator Overloading

## **Overloading ability of the various operators available in C#**

- unary operators take one operand and can be overloaded.
- Binary operators take two operands and can be overloaded.
- Comparison operators can be overloaded.
- Conditional logical operators cannot be overloaded directly
- Assignment operators cannot be overloaded.

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-6**

## **Reference and Value Types**

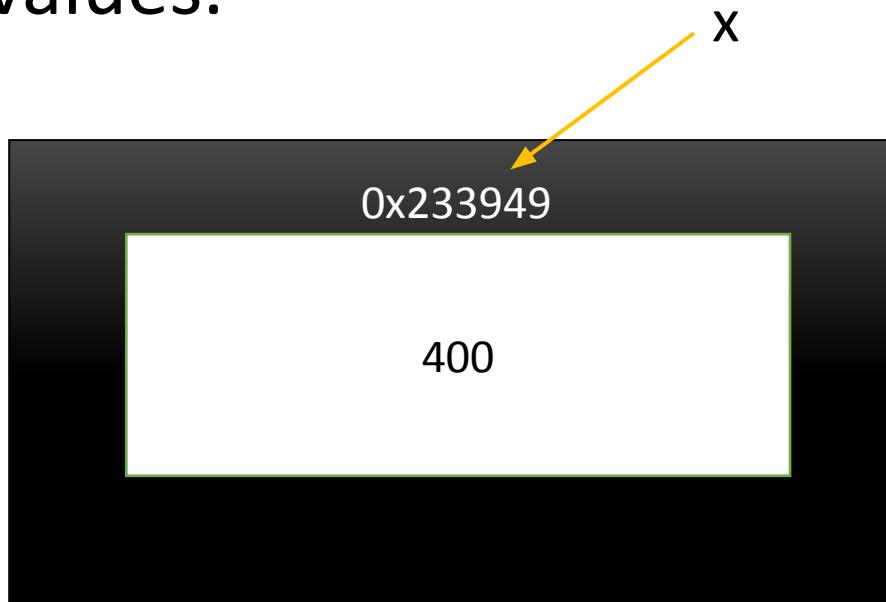
# Contents

- Value Types
  - enum
  - struct
- Reference Types
- Nullable Types
- ref and out
- Arrays
- Indexers

# Value Types

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

```
int x = 400;
```



Memory Allocation of Value Type

# Value Types

## **enum (enumeration type) :**

- It is mainly used to assign the names or string values to integral constants, that make a program easy to read and maintain
- is defined using **enum** keyword directly inside namespace, class or structure
- Enums are lists of constants. When you need a predefined list of values which do represent some kind of numeric or textual data, you should use an enum.
- The default type is **int**, and the approved types are **byte**, **sbyte**, **short**, **ushort**, **uint**, **long**, and **ulong**

Example :

```
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

# Value Types

## struct

It helps you to make a **single variable hold** related data of **various data types**.

The **struct** keyword is used for creating a structure.

```
struct Book
{
    public int bookID;
    public string title;
    public float price;
};

class Program
{
    0 references
    static void Main(string[] args)
    {
        Book obj; //Declaring obj of type Book

        obj.bookID = 101;
        obj.title = "Operation Management";
        obj.price = 340.50f;

        Console.WriteLine(obj.title);
        Console.WriteLine(obj.price);
        Console.WriteLine(obj.bookID);
    }
}
```

# Value Types

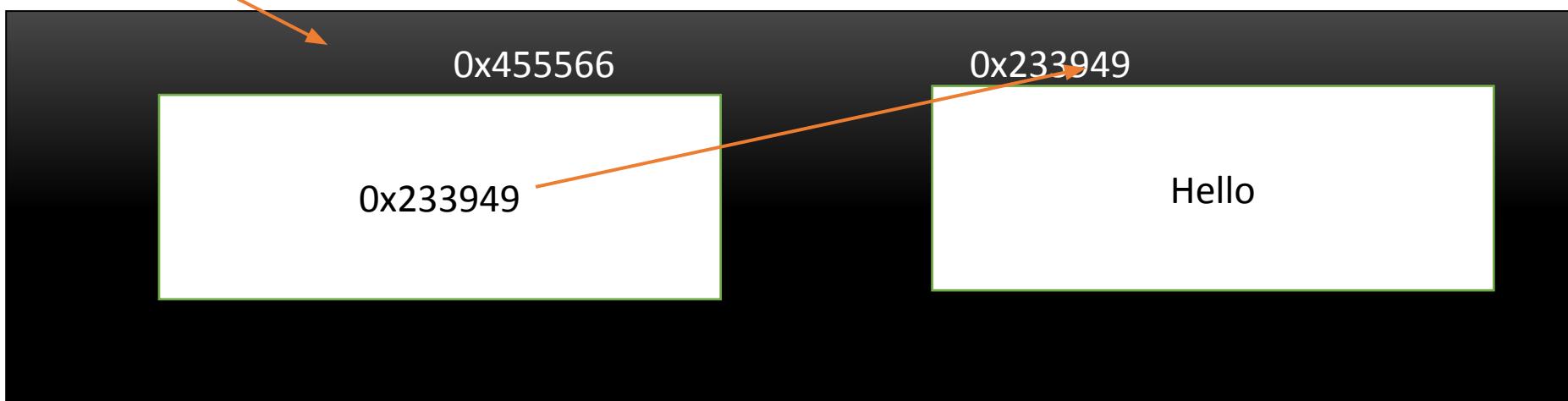
## **struct**

- Structures **cannot inherit other structures or classes.**
- Structures **can have defined constructors, but not destructors.**  
The default constructor is automatically defined and cannot be changed. It can contain parameterized constructor or static constructor.
- A structure can **implement** one or more **interfaces**.
- Structure members **cannot** be specified as **abstract, virtual, or protected**
- Structures can have **methods, fields and properties**

# Reference Types

- Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored.

```
string str = "Hello";
```

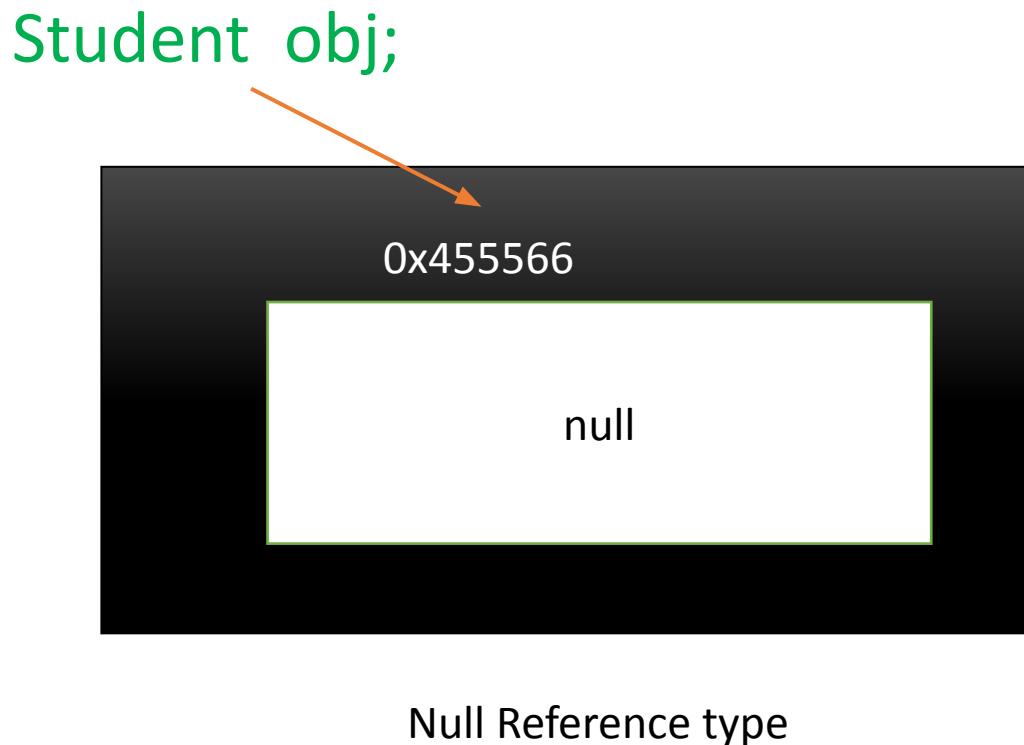


Reference type variables  
contains address where  
the value is stored

Actual Value

# Reference Types

- Null Reference type



# Nullable Types

- The Nullable type allows you to **assign a null value** to a **variable**.
- Nullable types can **only work with Value Type**, not with Reference Type.

Syntax : Nullable <data\_type> variable\_name = null;

Or

data\_type? variable\_name=null;

you can also use a shortcut which includes ? operator with the data type:

# Nullable Types

How to access the value of Nullable type variables?

You have to use **GetValueOrDefault()** method to get an original assigned value if it is not null. You will get the **default value zero** if it is null.

```
namespace Session6Demo
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            int? x = null;
            Nullable<int> y = 12;
            Console.WriteLine(x.GetValueOrDefault());
            Console.WriteLine(y.GetValueOrDefault());
        }
    }
}
```

# Nullable Types

Use the '??' (Null Coalescing Operator) to assign a nullable type to a non-nullable type.

```
namespace Session6Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            int? x = null;
            int y = x ?? 7; //if x is null then assign 7 to y.
            Console.WriteLine(y);
        }
    }
}
```

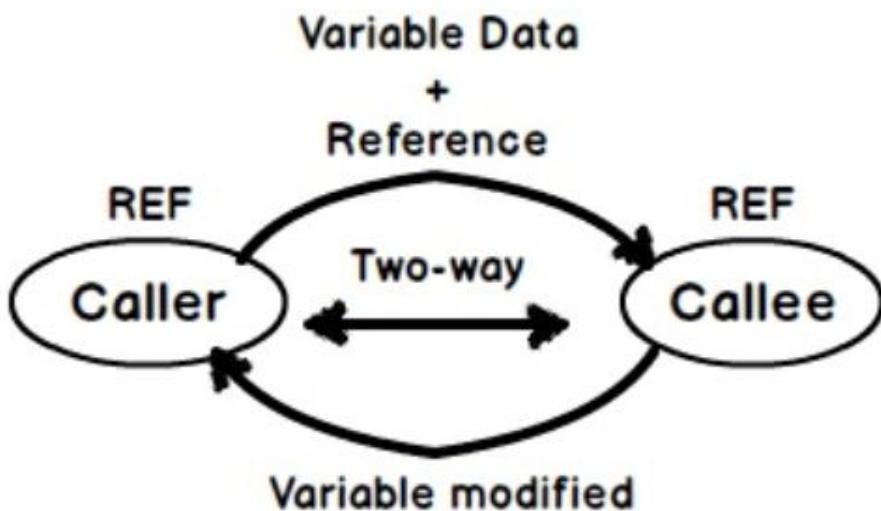
# Nullable Types

## Advantage of Nullable Types:

- The main use of nullable type is in database applications. Suppose, a table a row fetched null values in application, then you can use nullable type to get the null values.
- Nullable types can only be used with value types.
- The **HasValue** property returns true if the variable contains a value, or false if it is null.

# ref

The *ref* keyword passes arguments to **calling method by reference**. But before calling the method, the arguments **must be initialized** with some values.



```
namespace Session6Demo
{
    0 references
    class Program
    {
        1 reference
        static void reffunction(ref int i)
        {
            i = 50;
        }
        0 references
        static void Main(string [] args)
        {
            int r = 3; // Compiler error if no value assigned
            reffunction(ref r);
            Console.WriteLine(r);
        }
    }
}
```

# out

We need **not to initialize** the **parameter** before calling the method.

But C# compiler puts a constraint on the called method, that **parameter must be initialize within the method.**

```
namespace Session6Demo
{
    class Program
    {
        static void outfunction(out int i)
        {
            i = 50; //Compiler error if no value assigned
        }

        static void Main(string [] args)
        {
            int r; // No value assigned
            outfunction(out r);
            Console.WriteLine(r);
        }
    }
}
```

# Arrays

- Array is a collection of variables of the same type stored at contiguous memory locations.
- Declaring array: `datatype[] Arrayname;` eg. `int[] marks;`
- Initializing array : `int[] marks=new int[5];`
- Assigning values to an array:
  - Using index number : `marks[0]=56; marks[1]=66;`
  - At the time of declaration : `int[] marks={ 44, 55, 66};`
  - `int[] marks=new int[3] { 44, 55, 66};`
  - `int[] marks=new int[] { 44, 55, 66};`

# Arrays

## Accessing Array Elements :

```
namespace Session6Demo
{
    class Program
    {
        static void Main(string [] args)
        {
            int[] marks = new int[3];
            marks[0] = 66; //assigning value at index 0
            marks[1] = 77;//assigning value at index 1
            marks[2] = 44;//assigning value at index 2
            Console.WriteLine(marks[0] +" "+ marks[1]+ " " +marks[2]);
        }
    }
}
```

# Arrays

## Accessing Array Elements (using loops)

```
namespace Session6Demo
{
    class Program
    {
        static void Main(string [] args)
        {
            int[] marks = new int[3] { 33, 44, 55 };
            for(int i=0;i<marks.Length;i++)
            {
                Console.WriteLine(marks[i]);
            }
        }
    }
}
```

# Arrays

## Multidimensional Array

- The multi-dimensional array contains **more than one row** to store the values.
- To storing and accessing the values of the array, one required the **nested loop**

```
namespace Session6Demo
{
    class Program
    {
        static void Main(string [] args)
        {
            int[,] marks=new int[4,2];//4 rows and 2 columns
            marks[0,0] = 11; marks[0,1] = 22;
            marks[1,0] = 33; marks[1,1] = 44;
            marks[2,0] = 55; marks[2,1] = 66;
            marks[3,0] = 77; marks[3,1] = 88;
            for(int i = 0; i < 4;i++)
            {
                for(int j = 0; j < 2; j++)
                {
                    Console.WriteLine(marks[i,j] + " ");
                }
                Console.WriteLine();
            }
        }
    }
}
```

# Arrays

## Jagged Array

An array whose elements are arrays is known as Jagged arrays it means “**array of arrays**”. The jagged array elements may be of different dimensions and sizes.

```
namespace Session6Demo
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            int[][] arr= new int[2][]; //Jagged Array for 2 arrays

            arr[0] = new int[5] { 23, 32, 44, 33, 22 };
            arr[1] = new int[4] { 11, 2, 23, 21 };

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

# Indexer

- An indexer is a **special type of property** that **allows** a **class** or a **structure** to be **accessed** like an **array** for its internal collection.
- An indexer can be defined the same way as property with ‘this’ keyword and square brackets [ ].
- Modifiers can be private, public, protected or internal

## Syntax:

```
<modifier> <return type> this [argument list]
{
    get
    {
        // your get block code
    }
    set
    {
        // your set block code
    }
}
```

# Indexer

```
class Program
{
    private string [] names=new string[4];
    5 references
    public string this[int index] //creating indexer
    {
        get { return names[index]; }
        set { names[index] = value; }
    }
    0 references
    static void Main(string[] args)
    {
        Program obj=new Program();
        obj[0] = "Vikrant";
        obj[1] = "Alex";
        obj[2] = "Ken";
        obj[3] = "Smith";

        for(int i = 0; i < 4; i++)
            Console.WriteLine(obj[i]);
    }
}
```

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-7**

## **Generics and Collections**

# Contents

- Generics
- Generic classes
- Generic methods
- Generics constraints
- Collections ( generic and non-generic)
- Iterating collections using foreach
- Collections example based on ICollection, IList, IDictionary

# Generics

- Enables your **types** to be more safely **reusable** and **more efficient**.
- It means, generics **allow** you to write a class or method that **can work with any data type**.
- Generic has a **performance advantage** because it removes the possibilities of **boxing** and **unboxing**

# Generic Class

- A generic class **increases the reusability**. The more type parameters mean more reusable it becomes.
- A generic class can be a **base class** to other generic or non-generic classes or abstract classes.
- A generic class can be **derived** from other generic or non-generic interfaces, classes, or abstract classes.

```
namespace Session7Demo
{
    public class Program<T> //Generic class
    {
        public T data;
    }
    class TestProgram
    {
        static void Main(string[] args)
        {
            Program<int> p= new Program<int>();
            p.data = 32;
            Console.WriteLine(p.data);

            Program<string> p1 = new Program<string>();
            p1.data = "Apple";
            Console.WriteLine(p1.data);
        }
    }
}
```

# Generics class

```
namespace Session7Demo
{
    public class Program<T> where T : IComparable<T>
    {
        public T data;
        public string process(T input)
        {
            if (data.CompareTo(input) == 0)
                return "Same";
            else
                return "Not Same";
        }
    }
}
```

```
class TestProgram
{
    static void Main(string[] args)
    {
        Program<int> p = new Program<int>();
        p.data = 32;
        Console.WriteLine(p.process(32));

        Program<string> p1 = new Program<string>();
        p1.data = "Apple";
        Console.WriteLine(p1.process("Apple"));
    }
}
```

```
Microsoft Visual Studio Debug Console
Same
Same
```

# Non-Generics class

```
namespace Session7Demo
{
    public class Program
    {
        public object data;
        public string process(object input)
        {
            if (data==input)
                return "Same";
            else
                return "Not Same";
        }
    }
}
```

```
class TestProgram
{
    static void Main(string[] args)
    {
        Program p= new Program();
        p.data = 32;
        Console.WriteLine(p.process(32));

        Program p1 = new Program();
        p1.data = "Apple";
        Console.WriteLine(p1.process("Apple"));
    }
}
```

Microsoft Visual Studio Debug Console  
Not Same  
Same

# Generic Methods

- A method declared with the **type parameters** for its **return type** or **parameters** is called a generic method.

```
namespace Session7Demo
{
    public class Program<T>
    {
        public void showValues(T value) //Type parameter
        {
            Console.WriteLine(value);
        }
    }

    class TestProgram
    {
        static void Main(string[] args)
        {
            Program<int> p= new Program<int>();
            p.showValues(10);

            Program<string> p1 = new Program<string>();
            p1.showValues("India");
        }
    }
}
```

# Generic Constraints

If we want to restrict a generic class to accept only the particular type of placeholder, then we need to use **Constraints**

```
namespace Session7Demo
{
    // T: class constraint means Program class will accept only reference types
    public class Program<T> where T : class
    {
        public void showValues(T value) //Type parameter
        {
            Console.WriteLine(value);
        }
    }
    class TestProgram
    {
        static void Main(string[] args)
        {
            Program<int> p= new Program<int>(); //Error as its a value type
            p.showValues(10);

            Program<string> p1 = new Program<string>();
            p1.showValues("India");
        }
    }
}
```

# Generic Constraints

Constraint	Description
where T: struct	The type argument must be a value type.
where T: class	The type argument must be a reference type.
where T: <base class name>	The type of argument must be or derive from the specified base class.
where T: <interface name>	The type argument must be or implement the specified interface.

# Generic methods with Constraints

```
namespace Session7Demo
{
    4 references
    public class Program<T>
    {
        //X:struct constraint means only value type is accepted
        2 references
        public void showValues<X>(T value ) where X:struct
        {
            Console.WriteLine(value);
        }
    }
    0 references
    class TestProgram
    {
        0 references
        static void Main(string[] args)
        {
            Program<int> p= new Program<int>();
            p.showValues<int>(12);

            Program<string> p1 = new Program<string>();
            p1.showValues<string>("India");    //Error - method constraint is value type
        }
    }
}
```

# Collections (generic and non-generic)

- C# includes specialized classes that **store** series of **values or objects** are called collections.
- Collection classes serve various purposes, such as **allocating memory dynamically** to elements and **accessing a list of items** on the basis of an **index** etc.
- There are two types of collections available in C#: **non-generic collections** and **generic collections**.
- **System.Collections** - namespace contains the **non-generic** collection types
- **System.Collections.Generic** - namespace includes **generic** collection types.

# Generic Collections

- C# includes the following generic collection classes in the namespace **System.Collections.Generic**

Generic Collections	Description
List<T>	Contains elements of specified type. It grows automatically as you add elements in it.
Dictionary< TKey, TValue >	Contains key-value pairs.
SortedList< TKey, TValue >	Stores key and value pairs. It automatically adds the elements in ascending order of key by default.
Queue<T>	Stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection.
Stack<T>	Stores the values as LIFO (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values.
HashSet<T>	Contains non-duplicate elements. It eliminates duplicate elements.

# Generic Collections

**List<T>:** Its a **collection** of **strongly typed objects** that can be accessed by index and having methods for sorting, searching, and modifying list.

It is the generic version of the ArrayList



```
namespace Session7Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int>list = new List<int>();
            list.Add(17);
            list.Add(23);
            list.Add(31);
            foreach(int i in list)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

The image shows a code editor window with the following C# code. The code defines a namespace named 'Session7Demo' containing a class 'Program'. The 'Main' method creates a generic list of integers ('List<int>'), adds three integers (17, 23, 31) to it, and then iterates over the list using a 'foreach' loop to print each integer to the console. The code is color-coded, with 'List' and 'foreach' in purple, 'int' in blue, and 'Console.WriteLine' in teal.

**Note:** **foreach** is useful for traversing each items in an array or a collection of items and displayed one by one.

# Generic Collections

## Dictionary<Tkey,Tvalue>:

- Stores key-value pairs in no particular order.
- Implements IDictionary<TKey, TValue> interface.
- Keys must be unique and cannot be null.
- Values can be null or duplicate
- Elements are stored as KeyValuePair<TKey, TValue> objects.

```
namespace Session7Demo
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Dictionary<string, string> items = new Dictionary<string, string>();
            items.Add("India", "Rupees");
            items.Add("USA", "Dollars");
            items.Add("Japan", null);
            foreach(KeyValuePair <string, string> i in items)
            {
                Console.WriteLine(i.Key + " " + i.Value);
            }
        }
    }
}
```

# Non-generic Collections

C# includes the following non-generic collection classes in the namespace **System.Collections**

Non-generic Collections	Usage
ArrayList	Stores objects of any type like an array. However, there is no need to specify the size of the ArrayList like with an array as it grows automatically.
SortedList	Stores key and value pairs. It automatically arranges elements in ascending order of key by default. C# includes both, generic and non-generic SortedList collection.
Stack	Stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. C# includes both, generic and non-generic Stack.
Queue	Stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. C# includes generic and non-generic Queue.
Hashtable	Stores key and value pairs. It retrieves the values by comparing the hash value of the keys.

# Non-Generic Collections

## ArrayList:

- Its a non-generic **collection of objects** whose size increases dynamically.
- It is the same as Array except that its **size increases dynamically**.
- can be used to **add unknown data** where you **don't know the types** and the **size** of the data.

```
using System.Collections;
namespace Session7Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList set = new ArrayList();
            set.Add("India");
            set.Add(32);
            set.Add("USA");
            set.Add(null);
            foreach (object s in set)
            {
                Console.WriteLine(s);
            }
        }
    }
}
```

# Non-Generic Collections

## Hashtable

- is a non-generic collection that stores key-value pairs, similar to generic **Dictionary<TKey, TValue>** collection
- Implements **IDictionary** interface.
- Keys must be unique and cannot be null.
- Values can be null or duplicate.
- Elements are stored as **DictionaryEntry** objects.

```
using System.Collections;
namespace Session7Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable set = new Hashtable();
            set.Add(1, "India");
            set.Add(2, 32);
            set.Add(3, "USA");
            set.Add(4, null);
            foreach (DictionaryEntry s in set)
            {
                Console.WriteLine(s.Key + " " + s.Value);
            }
        }
    }
}
```

# Collections example using ICollection

```
namespace Session7Demo
{
    class Emp
    {
        public int empID;
        public string empName;
        public Emp(int id, string ename)
        {
            this.empID = id;
            this.empName = ename;
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        //Emp data type in place of T in ICollection
        ICollection<Emp> list = new List<Emp>();
        list.Add(new Emp(19, "Alex"));
        list.Add(new Emp(11, "Ram"));
        list.Add(new Emp(14, "John"));
        foreach(Emp i in list)
        {
            Console.WriteLine(i.empID+ " " + i.empName);
        }
    }
}
```

# Collections example using IList

```
namespace Session7Demo
{
    class Emp
    {
        public int empID;
        public string empName;
        public Emp(int id, string ename)
        {
            this.empID = id;
            this.empName = ename;
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        IList<Emp> list = new List<Emp>();
        list.Add(new Emp(19, "Alex"));
        list.Add(new Emp(11, "Ram"));
        list.Add(new Emp(14, "John"));
        foreach(Emp i in list)
        {
            Console.WriteLine(i.empID + " " + i.empName);
        }
    }
}
```

# Collections example using IDictionary

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Create a new dictionary of strings, with string keys,
        // and access it through the IDictionary generic interface.
        IDictionary< string, string> list = new Dictionary<string, string>();
        list.Add("Alex", "Manager");
        list.Add("John", "Clerk");
        list.Add("Merry", "Operator");

        // When you use foreach to enumerate dictionary elements,
        // the elements are retrieved as KeyValuePair objects.
        foreach (KeyValuePair<string, string> i in list)
        {
            Console.WriteLine(i.Key      +"  "+i.Value);
        }
    }
}
```

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-8**

## **Delegates**

# Contents

- Delegates
  - Calling methods using delegates
  - Uses of delegates
  - Multicast delegates
  - Action, func, Predicate delegates
- Anonymous methods
- Lambdas

# Delegates

- Contains the **memory address** of methods that **matches the same signature as the delegate** so that it can be called safely with the correct parameter types
- The delegate is a **reference type** data type
- A delegate can be declared outside of the class or inside the class. Practically, it **should be declared out of the class.**
- Steps while working with delegates:
  - Declare a delegate
  - Set a target method
  - Invoke a delegate

# Delegates

```
1 namespace Session8Demo
2 {
3     public delegate int MyDelegate(int x,int y); //Declare a Delegate
4     2 references
5     class TestProgram
6     {
7         1 reference
8         public int calculateFunction(int a,int b)//Target method
9         {
10            return (a + b);
11        }
12        0 references
13        static void Main(string[] args)
14        {
15            TestProgram obj=new TestProgram();
16            //Create Delegate Instance that points to the Method
17            MyDelegate d = new MyDelegate(obj.calculateFunction);
18            //Invoke the Delegate, which calls the method
19            Console.WriteLine(d(12, 22));
20        }
21    }
22 }
```

# Uses of Delegates

- Delegates are **reference type** but instead of referencing objects it **reference methods**.
- Delegates have **no method body**.
- Delegates are **type-safe, object-oriented** and **secure**.
- A Delegate is a **function pointer** that allows you to reference a method.
- A Function that is added to delegates must have **same return type** and **same signature** as delegate.

# Multicast Delegates

- A delegate that **points multiple methods** is called a multicast delegate
- It helps the user to point more than one method **in a single call**.
- Delegates are combined and when you call a delegate then a complete list of methods is called.
- All methods are called in **First in First Out(FIFO)** order.
- ‘+’ or ‘+=’ Operator is used to add the methods to delegates.
- ‘-’ or ‘-=’ Operator is used to remove the methods from the delegates list.

**Note:** Remember, multicasting of delegate should have a return type of void otherwise it will return the value only from the last method added in the multicast.

# Multicast Delegates

```
namespace Session8Demo
{
    public delegate void MyDelegate(int x,int y); //Declare a Delegate
    2 references
    class TestProgram
    {
        1 reference
        public void addFunction(int a,int b)//Target method1
        {
            Console.WriteLine("Addition function" + (a+b));
        }
        1 reference
        public void subFunction(int a, int b)//Target method2
        {
            Console.WriteLine("Subtract function" +(a-b));
        }
        1 reference
        public void mulFunction(int a, int b)//Target method3
        {
            Console.WriteLine("Multiplication function:" + (a * b));
        }
    }
}
```

```
0 references
static void Main(string[] args)
{
    TestProgram obj=new TestProgram();
    // call 1st method
    MyDelegate d1 = obj.addFunction;

    // call 2nd method
    d1 += obj.subFunction; //Multicasting
    // call 3rd method
    d1 += obj.mulFunction; //Multicasting
    // pass the values in three methods
    d1.Invoke(2, 3);
}
```

# Func Delegate

- Func is a **generic delegate** included in the **System** namespace.
- It has **zero or more *input*** parameters and **one *out*** parameter. The **last parameter** is considered as an ***out*** parameter.
- A Func delegate type can include 0 to 16 input parameters of different types. However, it must include an out parameter for the result.

# Func Delegate

```
namespace Session8Demo
{
    class TestProgram
    {
        static int addNum(int x, int y)
        {
            return x + y;
        }

        static string display(string msg)
        {
            return msg;
        }

        static float calculateTax()
        {
            return (0.5f * 1300);
        }
    }
}
```

```
0 references
static void Main(string[] args)
{
    Func<int, int,int> add = addNum;//2 input and 1 out parameter
    int result = add(10, 10);
    Console.WriteLine(result);

    Func<string,string> func = display;// 1 input and 1 out parameter
    string message = func("Hello world");
    Console.WriteLine(message);

    Func<float> obj = calculateTax; //No input and 1 out parameter
    float res = obj();
    Console.WriteLine(res);
}
```

# Action Delegate

- Action is a delegate type defined in the **System** namespace.
- An Action type delegate is the same as Func delegate except that the Action delegate **doesn't return a value**. In other words, an Action delegate can be **used with a method that has a void return type**.
- An Action delegate can take up to 16 input parameters of different types.

```
namespace Session8Demo
{
    class TestProgram
    {
        static void addNum(int x, int y)
        {
            Console.WriteLine(x + y);
        }

        static void display(string msg)
        {
            Console.WriteLine(msg);
        }

        static void Main(string[] args)
        {
            Action<int,int> obj1 = addNum;
            obj1(10, 12);
            Action<string> obj2 = display;
            obj2("Hello World");
        }
    }
}
```

# Predicate Delegate

- It represents a method containing a set of criteria and checks whether the passed parameter meets those criteria.
- A predicate delegate methods must take one input parameter and return a boolean - true or false.
- Predicate delegate is defined in the **System** namespace.

# Predicate Delegate

```
namespace Session8Demo
{
    0 references
    class TestProgram
    {
        1 reference
        static bool checkValues(int x)
        {
            bool flag=false;
            if (x > 0)
                flag= true;
            else
                flag= false;
            return flag;
        }
        0 references
        static void Main(string[] args)
        {
            Predicate<int> obj1 = checkValues;
            Console.WriteLine(obj1(10));
        }
    }
}
```

# Anonymous Method

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

```
namespace Session8Demo
{
    public delegate void MyDelegate(int val); //Declare delegate
    0 references
    class TestProgram
    {
        0 references
        static void Main(string[] args)
        {
            //Anonymous method
            MyDelegate d = delegate (int val) {
                Console.WriteLine(val);
            };

            d(10);
        }
    }
}
```

# Anonymous Method as parameter

```
namespace Session8Demo
{
    public delegate void MyDelegate(int val); //Declare delegate
    0 references
    class TestProgram
    {
        1 reference
        static void printValues(MyDelegate obj,int val)
        {
            obj(val);
        }
        0 references
        static void Main(string[] args)
        {
            //Passing Anonymous method values
            printValues(delegate (int val) { Console.WriteLine(val); }, 400);
        }
    }
}
```

# Lambda

- A lambda expression in C# **describes a pattern**.
- Lambda Expressions has the token `=>` in an expression context. This is read as “**goes to**” operator and **used** when a **lambda expression is declared**.
- **Lambda expression** is a better way to **represent an anonymous method**.
- Both anonymous methods and Lambda expressions allow you define the **method implementation inline**, however, an **anonymous method explicitly** requires you to **define the parameter types** and the **return type** for a method
- Syntax :

*Parameter => expression*      eg. `sum=> sum+3;`  
*Parameter-list => expression*    eg. `(a, b) => a * b;`

# Lambda

```
namespace Session8Demo
{
    public delegate void MyDelegate(int val); //Declare Delegate
    0 references
    class TestProgram
    {
        0 references
        static void Main(string[] args)
        {
            //Lambda expressions
            MyDelegate d1 = x =>
            {
                Console.WriteLine(x);
            };
            d1(8);
        }
    }
}
```

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-9**

## **Exception Handling**

# Contents

- Exceptions
- Checked and Unchecked
- Exception Hierarchy and classes
- Exception handling
  - Try-catch-finally
  - Do's and Don'ts of Exception Handling
- User-defined Exception classes
- Events
  - Declaring and raising the events

# Exceptions

- An exception is an **abnormal condition** that arises in a code sequence **at run time**.
- In other words, an exception is a **run-time error**
- When an exceptional condition arises, an **object** representing that exception is **created and thrown** in the **method** that caused the error

# Checked and Unchecked

- To handle integral type exceptions in C#, the checked and unchecked keywords are used.
- They thus define the checked context where arithmetic overflow raises an exception and the unchecked context where the arithmetic overflow is ignored and the result is truncate

```
namespace Session9Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            int n = int.MaxValue;
            Console.WriteLine(n); //output : 2147483647
            Console.WriteLine(n + 10); //output: -2147483639
        }
    }
}
```

# Checked and Unchecked

## Checked

To check the overflow explicitly, the checked keyword is used in C#. It also checks the conversion of the integral type values at compile time

```
namespace Session9Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            checked
            {
                int n = int.MaxValue;
                Console.WriteLine(n + 10);
            }
        }
    }
}
```

```
Unhandled Exception:
System.OverflowException: Arithmetic operation resulted in an overflow.
```

# Checked and Unchecked

## Unchecked

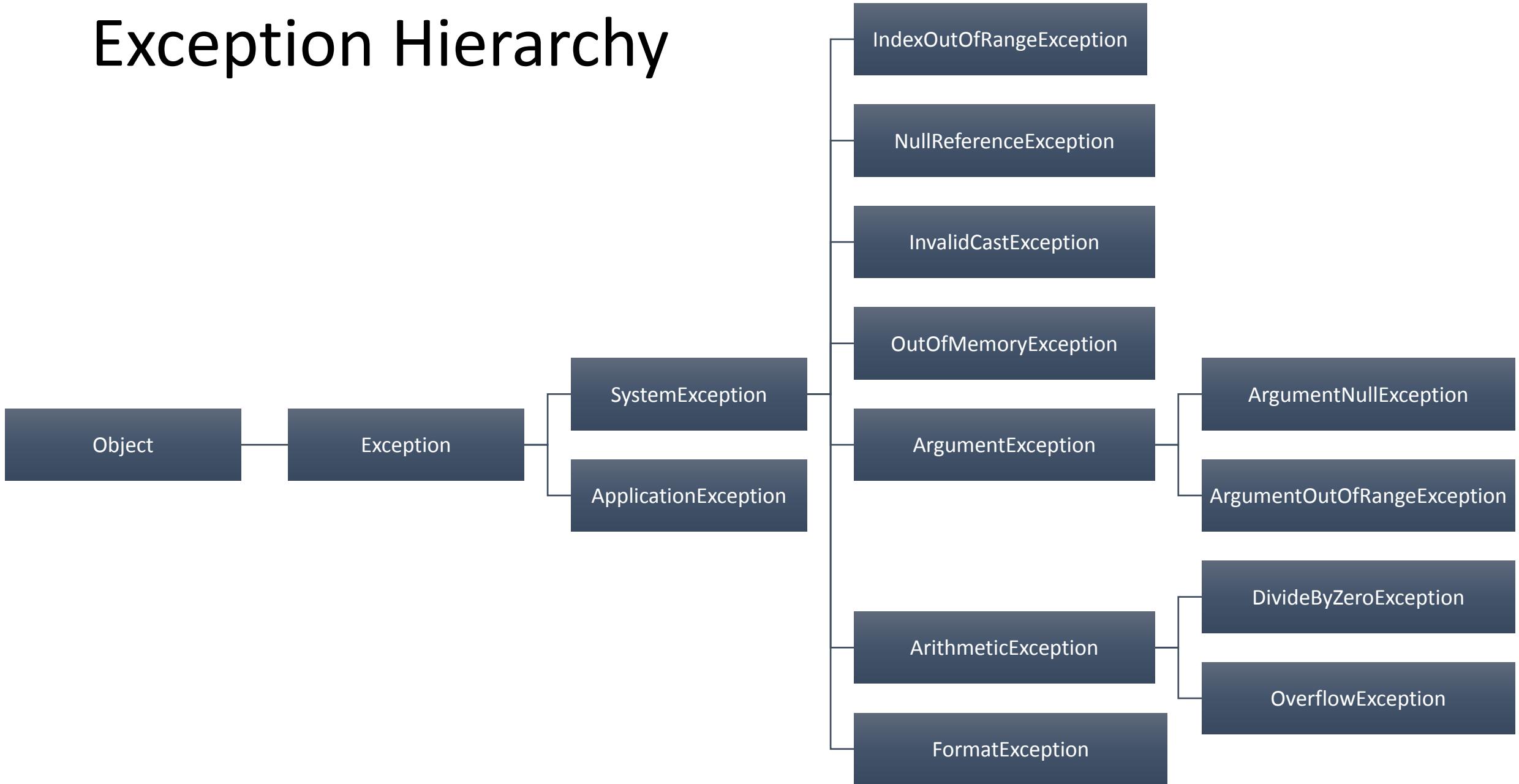
The integral-type arithmetic exceptions are ignored by the C# Unchecked keyword.

It may thus produce a truncated or wrong result, as it does not check explicitly.

```
namespace Session9Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            unchecked
            {
                int n = int.MaxValue;
                Console.WriteLine(n + 10);
            }
        }
    }
}
```

Output: -2147483639

# Exception Hierarchy



# Exception classes

Exception Class	Description
<b>ArgumentException</b>	Raised when a non-null argument that is passed to a method is invalid.
<b>ArgumentNullException</b>	Raised when null argument is passed to a method.
<b>ArgumentOutOfRangeException</b>	Raised when the value of an argument is outside the range of valid values.
<b>DivideByZeroException</b>	Raised when an integer value is divide by zero.
<b>FileNotFoundException</b>	Raised when a physical file does not exist at the specified location.
<b>FormatException</b>	Raised when a value is not in an appropriate format to be converted from a string by a conversion method such as Parse.
<b>IndexOutOfRangeException</b>	Raised when an array index is outside the lower or upper bounds of an array or collection.

# Exception classes

Exception Class	Description
<b>NullReferenceException</b>	Raised when program access members of null object.
<b>OverflowException</b>	Raised when an arithmetic, casting, or conversion operation results in an overflow.
<b>OutOfMemoryException</b>	Raised when a program does not get enough memory to execute the code.
<b>StackOverflowException</b>	Raised when a stack in memory overflows.
<b>InvalidCastException</b>	Handles errors generated during typecasting.

# Exception handling (key definitions)

try	Used to define a try block. This block holds the code that may throw an exception.
catch	Used to define a catch block. This block catches the exception thrown by the try block.
finally	The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
throw	Used to throw an exception manually.

# Exception handling

Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords.

A try/catch block is placed around the code that might generate an exception.

```
try
{
    // statements causing exception
}

catch (ExceptionName e1)
{
    // error handling code
}

catch (ExceptionName e2)
{
    // error handling code
}

catch (ExceptionName eN)
{
    // error handling code
}

finally
{
    // statements to be executed
}
```

# Exception Handling

```
namespace Session9Demo
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            try
            {
                int x = 45; int y = 0;
                int z = x / y;
            }
            catch (ArithmaticException e1)
            {
                Console.WriteLine("Error: " + e1.Message);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Microsoft Visual Studio Debug Console  
Error: Attempted to divide by zero.  
Code ends...

By : Dr. Vikrant

12

# Exception Handling

```
namespace Session9Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int[] arr = new int[3];
                arr[0] = 33; arr[1] = 53; arr[2] = 99; arr[3] = 11;
                foreach (int i in arr)
                {
                    Console.WriteLine(i);
                }
            }
        }
    }
}
```

```

}
catch (IndexOutOfRangeException e1)
{
    Console.WriteLine("Error: " + e1.Message);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
finally
{
    Console.WriteLine("Code ends...");
}
```

 Microsoft Visual Studio Debug Console  
Error: Index was outside the bounds of the array.  
Code ends...

# Do's and Don't of Exception Handling

## Do's

- Use try/catch/finally blocks to recover from errors or release resources
- Throw exceptions instead of returning an error code
- Use the predefined .NET exception types
- End exception class names with the word “Exception”
- Include three constructors in custom exception classes
  - [Exception\(\)](#), which uses default values.
  - [Exception\(String\)](#), which accepts a string message.
  - [Exception\(String, Exception\)](#), which accepts a string message and an inner exception.

# Do's and Don't of Exception Handling

## Don't

- Don't throw an exception when a simple if statement can be used to check for errors. For example, a simple if statement to check whether a connection is closed is much better than throwing an exception for the same.
- Don't catch **Exception**. Always use the most specific exception for the code you are writing. Remember good code is not code that doesn't throw exceptions. Good code throws exceptions as needed and handles only the exceptions it knows how to handle.
- Don't swallow an exception by putting an empty catch block

# User-Defined Exception Classes

- Define your own exception. User-defined exception classes are derived from the Exception class.

```
namespace Session9Demo
{
    3 references
    class MyException : Exception
    {
        1 reference
        public MyException(string message)
        {
            Console.WriteLine(message);
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            int x = 0;
        }
    }
}
```

```
try
{
    if(x==0)
        throw new MyException("This is incorrect...");

} catch(MyException ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
```

# Events

- **Events** are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur
- The class that **sends or raises** an event is called a **Publisher** and class that **receives or handle** the event is called "**Subscriber**".
- A **publisher** is an object that contains the definition of the event and the delegate. A publisher class object invokes the event and it is notified to other objects.
- A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

# Events

## Declaring and Raise Events

To declare an event inside a class, first of all, you must declare a delegate type for the event as:

```
public delegate string MyDelegate(string str);
```

then, declare the event using the **event** keyword –

```
event MyDelegate MyEvent;
```

then, raise the event using the **invoke**

```
MyEvent.invoke("India");
```

# Events

```
namespace Session9Demo
{
    class PublisherClass //Publisher Class
    {
        // Declare Delegate
        public delegate void MyDelegate(string msg);
        // Declaring Event
        public event MyDelegate myevent;
        public void displayMessage()
        {
            myevent.Invoke("Welcome to India");// Raise event
        }
    }
}
```

```
class Program //Subscriber Class
{
    //Event handler
    //Delegate call this method when event raised
    static void message(string str)
    {
        Console.WriteLine(str);
    }
    static void Main(string[] args)
    {
        PublisherClass obj = new PublisherClass();

        // Event gets binded with delegate
        obj.myevent += new PublisherClass.MyDelegate(message);
        obj.displayMessage();
    }
}
```

# Exercise

Write a C# code that simply adds two numbers. Only condition is if the sum of number is odd it fires an event that print a message using delegates.

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-10**

**Partial Classes and LINQ**

# Contents

- Anonymous types
- Extension methods
- Partial classes
- Partial methods
- LINQ
- Writing LINQ queries
- LINQ to objects
- Deferred execution
- PLINQ

# Anonymous Types

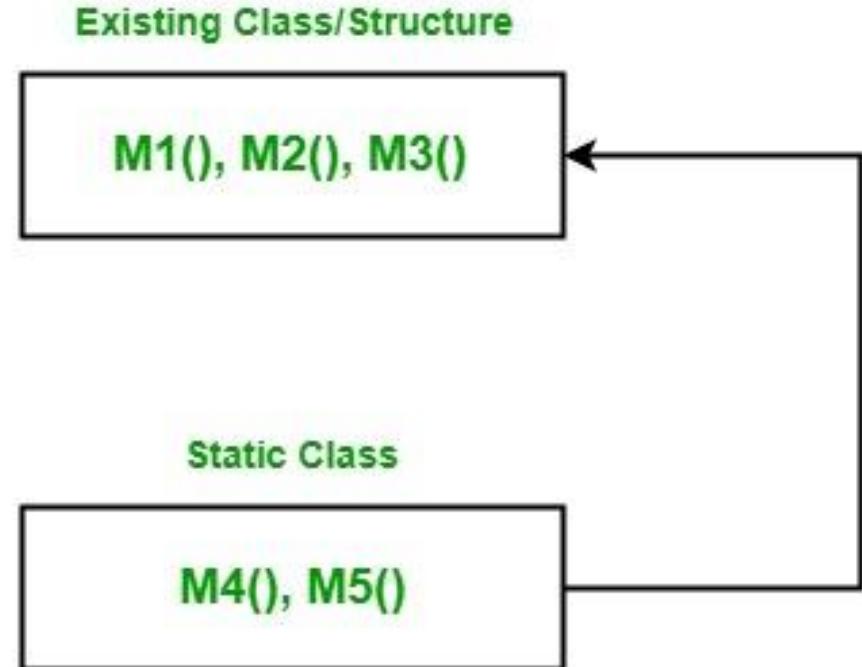
- Anonymous type is a **type without any name** that can contain public **read-only properties** only. It **cannot contain** other members, such as **fields, methods, events**, etc.
- You create an anonymous type using the ***new*** operator. The implicitly typed variable- **var** is used to **hold the reference** of anonymous types.
- The properties of anonymous types are **read-only** and **cannot be initialized** with a **null, anonymous function, or a pointer type**.

# Anonymous Types

```
namespace Session10Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            var student = new { rollNo = 100, name = "Alex", address = "Pune" };
            Console.WriteLine(student.rollNo);
            Console.WriteLine(student.name);
            Console.WriteLine(student.address);
            student.rollNo = 33; //Error : Its a read-only and cannot be changed
        }
    }
}
```

# Extension Methods

- **Extension method** concept allows you to add new methods in the existing class or in the structure without modifying the source code of the original type
- You need to **create** a **new class** which is **static** and contain lets say two methods which you want to add in the existing class, now **bind this class** with the **existing class**.
- After binding you will see the existing class can access the two new added methods



# Extension Methods

```
namespace Session10Demo
{
    public class ExistingClass
    {
        public void M1() { Console.WriteLine("M1 method"); }
        public void M2() { Console.WriteLine("M2 method"); }
        public void M3() { Console.WriteLine("M3 method"); }

    }
    public static class StaticClass
    {
        public static void M4(this ExistingClass g) { Console.WriteLine("M4 method"); }
        public static void M5(this ExistingClass g, string str) { Console.WriteLine(str); }
    }
}
```

```
class TestProgram
{
    static void Main(string[] args)
    {
        ExistingClass obj=new ExistingClass();
        obj.M1();
        obj.M2();
        obj.M3();
        obj.M4();
        obj.M5("M5 Method");
    }
}
```

# Partial Class

- It is possible to split the implementation of a class, a struct, a method, or an interface in multiple (.cs) files using the partial keyword.
- The compiler will combine all the implementation from multiple (.cs) files when the program is compiled

# Partial Class

## Rules for Partial Classes

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

# Partial Class

```
namespace Session10Demo
{
    4 references
    public partial class Student
    {
        public int studentID;
        public string studentName;
        1 reference
        public Student(int id, string name)
        {
            this.studentID = id;
            this.studentName = name;
        }
    }
    4 references
    public partial class Student
    {
        1 reference
        public void displayStudents()
        {
            Console.WriteLine("ID:" + studentID);
            Console.WriteLine("Name:" + studentName);
        }
    }
}
```

```
0 references
class TestProgram
{
    0 references
    static void Main(string[] args)
    {
        Student obj = new Student(101,"Alex");
        obj.displayStudents();
    }
}
```

# Partial Method

C# contains a special method is known as a partial method, which contains declaration part in one partial class and definition part in another partial class

# Partial Method

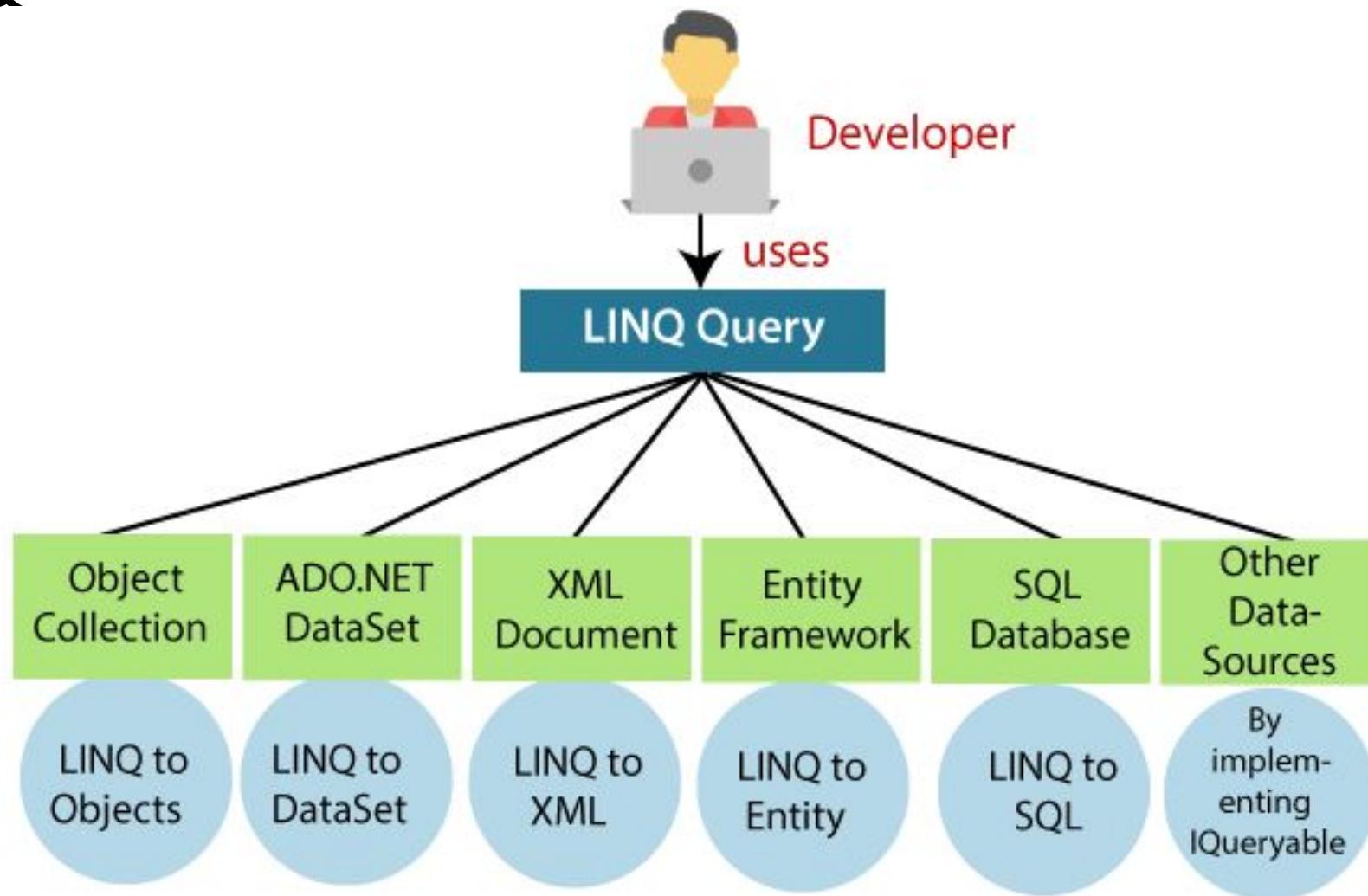
```
namespace Session10Demo
{
    4 references
    public partial class Student
    {
        public int studentID;
        public string studentName;
        1 reference
        public Student(int id, string name)
        {
            this.studentID = id;
            this.studentName = name;
        }
        2 references
        public partial void displayStudents();
    }
    4 references
    public partial class Student
    {
        2 references
        public partial void displayStudents()
        {
            Console.WriteLine("ID:" + studentID);
            Console.WriteLine("Name:" + studentName);
        }
    }
}
```

```
        ]
        ]
    }
    0 references
    class TestProgram
    {
        0 references
        static void Main(string[] args)
        {
            Student obj = new Student(101,"Alex");
            obj.displayStudents();
        }
    }
}
```

# LINQ

- LINQ is a **Language Integrated Query**.
- LINQ provides the new way **to manipulate the data**, whether it is to or from **the database** or with an **XML file** or with a **simple list of dynamic data**.
- LINQ is a uniform query system in C# to retrieve the data from different sources of data and formats such as **XML, generics, collections, ADO.Net DataSet, Web Service, MS SQL Server, and other databases server**.
- LINQ provides the rich, **standardized query syntax** in a .NET programming language such as C# and VB.NET, which allows the developers to interact with any data sources.

# LINQ



# LINQ

## Advantages of LINQ

- We do not need to learn new query language syntaxes for different sources of data because it provides the standard query syntax for the various data sources.
- In LINQ, we have to write less code in comparison to the traditional approach.
- LINQ provides the compile-time error checking as well as intelligence support in Visual Studio. This powerful feature helps us to avoid run-time errors.
- LINQ provides a lot of built-in methods that we can be used to perform the different operations such as filtering, ordering, grouping, etc. which makes our work easy.
- The query of LINQ can be reused.

# LINQ

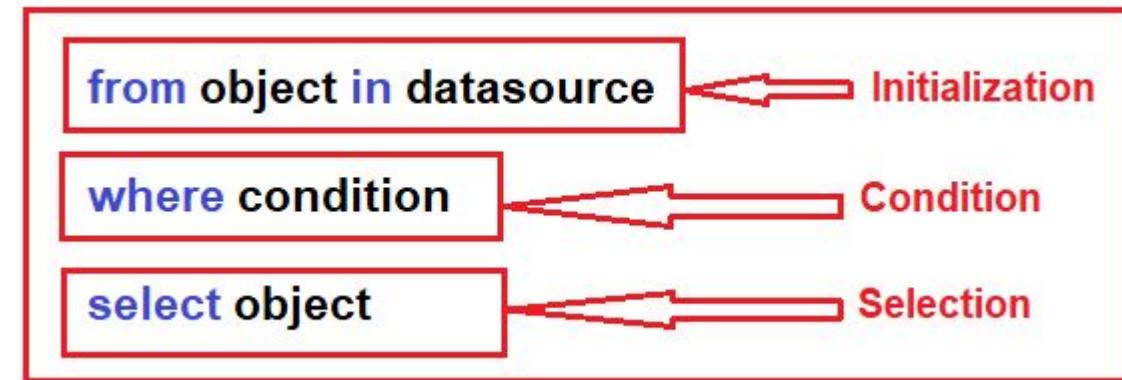
## Writing LINQ Query:

- LINQ queries **return results as objects**. It enables you to uses object-oriented approach on the result set and not to worry about transforming different formats of results into objects.
- Each Query is a combination of three things; they are:
  1. Initialization(to work with a particular data source)
  2. Condition(where, filter, sorting condition)
  3. Selection (single selection, group selection or joining)
- There are two basic ways to write a LINQ query to **IEnumerable** collection or IQueryable data sources.
  1. Using Query Syntax
  2. Using Method Syntax

# LINQ

## Writing LINQ Query:

### 1. Using Query Syntax



### Example :

```
Result variable      Range variable      Sequence  
var result = from s in strList           where s.Contains("Tutorials")  
Standard Query Operators    select s;           Sequence  
                                         (IEnumerable or  
                                         IQueryable collection)  
                                         Conditional expression
```

# LINQ

## Writing LINQ Query:

### 1. Using Query Syntax

#### Example -1

```
namespace Session10Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> list = new List<string>();
            list.Add("Java");
            list.Add(".NET");
            list.Add("Python");
            list.Add("SQL");
            // LINQ Query Syntax
            var result =from items in list
                        where items.Contains("Java")
                        select items;

            foreach (var item in result)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

# LINQ

## Writing LINQ Query:

### 1. Using Query Syntax

#### Example -2

```
namespace Session10Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> list = new List<int>();
            list.Add(34);
            list.Add(44);
            list.Add(75);
            list.Add(20);
            // LINQ Query Syntax
            var result =from items in list
                        where items >= 44
                        select items;

            foreach (var item in result)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

B No issues found

# LINQ

## Points to Remember (FOR QUERY SYNTAX)

- As name suggest, **Query Syntax** is same like SQL (Structure Query Language) syntax.
- Query Syntax starts with *from* clause and can be end with *Select* or *GroupBy* clause.
- Use various other operators like filtering, joining, grouping, sorting operators to construct the desired result.
- Implicitly typed variable - var can be used to hold the result of the LINQ query.

# LINQ

## Writing LINQ Query:

### 2. Using Method Syntax

- Method syntax **uses extension methods** included in the **Enumerable or Queryable static class**, similar to how you would call the extension method of any class.
- The extension method **Where()** is defined in the Enumerable class.

Example :

```
var result = strList.Where(s => s.Contains("Tutorials"));
```

A diagram illustrating the components of the LINQ query. A blue arrow points from the word 'Where' in the code to the text 'Extension method'. Another blue bracket spans the entire lambda expression 's => s.Contains("Tutorials")' with the text 'Lambda expression' written below it.

# LINQ

## Writing LINQ Query: 2. Using Method Syntax

### Example -1

```
namespace Session10Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> stringList = new List<string>()
            {
                "C# Tutorials",
                "VB.NET Tutorials",
                "Learn C++",
                "MVC Tutorials" ,
                "Java"
            };
            // LINQ Method Syntax
            var result = stringList.Where(s => s.Contains("Tutorials"));
            foreach (var item in result)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

# LINQ to Object

- **LINQ to Objects** offers usage of any LINQ query supporting `IEnumerable<T>` for accessing in-memory data collections **without any need of LINQ provider (API)** as in case of LINQ to SQL or LINQ to XML.
- Queries in LINQ to Objects return variables of type usually `IEnumerable<T>` only
- There are also many advantages of LINQ to Objects over traditional foreach loops like **more readability, powerful filtering, capability of grouping, enhanced ordering** with minimal application coding.

# LINQ to Object

```
namespace Session10Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] str= { "India", "Japan", "USA", "UK" };

            var result = from items in str
                         select items;

            foreach (var item in result)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

# Deferred Execution

- In Deferred Execution, the query is not executed when declared. It is **executed when the query object is iterated over a loop.**
- Deferred execution is applicable on any in-memory collection as well as remote LINQ providers like LINQ-to-SQL, LINQ-to-Entities, LINQ-to-XML, etc.
- Deferred Execution **returns the Latest Data**

# Deferred Execution

```
namespace Session10Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> list = new List<int>();
            list.Add(34);
            list.Add(44);
            list.Add(75);
            list.Add(20);
            // LINQ Query Syntax
            var result = from items in list
                        where items >= 44
                        select items;

            foreach (var item in result)
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

In this example, you can see the query is materialized and executed when you iterate using the foreach loop. This is called deferred execution

Query does not executes here

Query executes here

# PLINQ (Parallel LINQ)

- Parallel LINQ (PLINQ) is a parallel implementation of LINQ to Objects.
- PLINQ supports Parallel programming and is closely related to the Task Parallel Library.
- In very simple words, PLINQ enables a query to automatically take advantage of multiple processors.
- PLINQ can significantly increase the speed of LINQ to Objects queries by using all available cores on the host computer more efficiently.

Sequential Linq execution

```
var results = from c in customers  
              where c.FirstName.StartsWith("San")  
              select c;
```

Parallel Linq execution

```
var results = from c in customers.AsParallel()  
              where c.FirstName.StartsWith("San")  
              select c;
```

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-11**

**Reflection and File IO**

# Contents

- Creating Shared Assembly
- Creating Custom Attributes
- Reflection
- Dynamic Assembly Loading using Reflection
- Files IO and Streams
  - Reading and Writing files
  - Working with Drives, Directories and Files

# Creating Shared Assembly

- A **shared assembly** is an assembly that **resides in a centralized location known as the GAC** (Global Assembly Cache) and that provides resources to multiple applications.
- If an assembly is shared **then single copy** can be used by **multiple applications**.
- The GAC folder is under the Windows folder.

# Steps to create and use Shared Assembly

**Step 1 :** Open VS.NET and Create a new Class Library

**Step 2:** Generating Cryptographic Key Pair using the tool SN.Exe

- Make sure that you start Administrator “Developer Command Prompt for VS 2022”
- Write the following command on command prompt

**C:\> sn -k "C:\mynewkey.snk"**

**Step 3:** Sign the component with the key and build the class library project (Go to properties of the project in solution explorer -> select signing -> Check the checkbox of Sign the assembly and browse for the key).

**Step 4:** Host the signed assembly in Global Assembly Cache

**C:\>gacutil -i "E:\MyClassLibrary\bin\Debug\MyClassLibrary.dll"**

# Steps to create Shared Assembly

**Step 5 :** Test the assembly by creating the client application. Just add the project reference and browse for the shared assembly recently created.

**Step 6 :** Execute the client program

# Creating Custom Attributes

- Custom attributes are essentially traditional classes that derive directly or indirectly from `System.Attribute`
- Attributes are **metadata extensions** that give **additional information to the compiler** about the **elements** in the program code at **runtime**.
- Attributes are used **to impose conditions or to increase the efficiency of a piece of code**
- The primary steps to properly design custom attribute classes :
  - a. Applying the `AttributeUsageAttribute`
  - b. Declaring the attribute class
  - c. Declaring constructors
  - d. Declaring properties

# Custom Attributes

## Applying the AttributeUsageAttribute :

A custom attribute declaration begins with the `System.AttributeUsageAttribute`, which **defines some of the key characteristics** of your **attribute class**. For example, you can specify whether your attribute can be inherited by other classes or specify which elements the attribute can be applied to.

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
```

The `AttributeUsageAttribute` has three members that are important for the creation of custom attributes: **AttributeTargets**, **Inherited**, and **AllowMultiple**.

# Custom Attributes

Applying the **AttributeUsageAttribute** :

**AttributeTargets.All** is specified, indicating that **this attribute can be applied to all program elements.**

**AttributeTargets.Class**, indicating that **your attribute can be applied only to a class**

**AttributeTargets.Method**, indicating that **your attribute can be applied only to a method.**

# Custom Attributes

## Applying the AttributeUsageAttribute :

**AttributeUsageAttribute.Inherited** property indicates whether your attribute can be inherited by classes that are derived from the classes to which your attribute is applied.

```
// This defaults to Inherited = true.  
public class MyAttribute : Attribute  
{  
    //...  
}  
  
[AttributeUsage(AttributeTargets.Method, Inherited = false)]  
public class YourAttribute : Attribute  
{  
    //...  
}
```

```
//The two attributes are then applied to a method in the  
base class  
public class MyClass  
{  
    [MyAttribute]  
    [YourAttribute]  
    public virtual void MyMethod()  
    {  
        //...  
    }  
}
```

# Custom Attributes

Applying the `AttributeUsageAttribute` :

Finally, the class `YourClass` is inherited from the base class `MyClass`. The method `MyMethod` shows `MyAttribute` but not `YourAttribute`

```
public class YourClass : MyClass
{
    // MyMethod will have MyAttribute but not YourAttribute.
    public override void MyMethod()
    {
        //...
    }
}
```

# Custom Attributes

## Applying the AttributeUsageAttribute :

**AttributeUsageAttribute.AllowMultiple** property indicates whether multiple instances of your attribute can exist on an element. If set to **true**, multiple instances are allowed; if set to **false**( default), only one instance is allowed.

```
//This defaults to AllowMultiple = false.  
public class MyAttribute : Attribute  
{  
}  
  
[AttributeUsage(AttributeTargets.Method, AllowMultiple =  
true)]  
public class YourAttribute : Attribute  
{  
}
```

```
public class MyClass  
{  
    // This produces an error.  
    // Duplicates are not allowed.  
    [MyAttribute]  
    [MyAttribute]  
    public void MyMethod()  
    {  
        //...  
    }  
  
    // This is valid.  
    [YourAttribute]  
    [YourAttribute]  
    public void YourMethod()  
    {  
        //...  
    }  
}
```

# Custom Attributes

Declaring the Attribute class :

[AttributeUsage(AttributeTargets.Method)]

```
public class MyAttribute : Attribute  
{  
    // ...  
}
```

# Custom Attributes

Declaring Constructor:

```
public MyAttribute(bool myvalue)
{
    this.myvalue = myvalue;
}
```

# Custom Attributes

## Declaring Properties

```
public bool MyProperty  
{  
    get {return this.myvalue;}  
    set {this.myvalue = value;}  
}
```

# Reflection

- **Reflection** objects are used for **obtaining type information** at **runtime**.  
The classes that give access to the metadata of a running program are in the **System.Reflection** namespace.
- Reflection has the following applications :
  - It allows view attribute information at runtime.
  - It allows examining various types in an assembly and instantiate these types.
  - It allows late binding to methods and properties
  - It allows creating new types at runtime and then performs some tasks using those types.

# Reflection

- The **MethodInfo** object of the **System.Reflection** class needs to be initialized for discovering the attributes associated with a class.

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

# Reflection

```
namespace Session11Demo
{
    [AttributeUsage(AttributeTargets.All, AllowMultiple = false, Inherited = true)]
    public class FirstAttribute : Attribute
    {
        private string name=null;

        1 reference
        public FirstAttribute(string names)
        {
            Name = names;
        }

        1 reference
        public string Name { get => name; set => name = value; }
    }
}
```

```
[AttributeUsage(AttributeTargets.All)]
2 references
public class SecondAttribute : Attribute
{
    private string address=null;

    1 reference
    public SecondAttribute(string addr)
    {
        Address = addr;
    }

    1 reference
    public string Address { get => address; set => address = value; }

    [FirstAttribute("Vikrant")]
    [SecondAttribute("Pune")]
    1 reference
    public class MyClass
    {

    }

    0 references
    public class YourClass
    {
        0 references
        static void Main(string[] args)
        {
            System.Reflection.MemberInfo memberInfo = typeof(MyClass);

            var obj = memberInfo.GetCustomAttributes(true);

            foreach (var i in obj)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

# Dynamic Assembly Loading using Reflection

the act of loading external assemblies on demand is known as Dynamic Loading. Using the Assembly class, we can dynamically load both private and shared assemblies from the local location to a remote location as well as, explore its properties.

```
using System.Reflection;
namespace Session11Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            Assembly assembly = Assembly.LoadFile(@"E:\MSVS2022CDAC\SharedAssemblyDemo.dll");

            Type[] t = assembly.GetTypes();

            foreach (var i in t)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

# Files IO and Streams

## Files

- A **file** is a collection of data stored in a disk with a specific name and a directory path.
- **System.IO** namespace has various classes that are used for performing numerous operations with files, such as creating and deleting files, reading from or writing to a file, closing a file etc.

# Files IO and Streams

## Basic file operations :

- **Reading** – data is read from a file.
- **Writing** – data is written to a file. By default, all existing contents are removed from the file, and new content is written.
- **Appending** – writing information to a file. The only difference is that the existing data in a file is not overwritten. The new data to be written is added at the end of the file.

# Files

## { Reading text files}

```
using System.IO;
namespace Session11Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            string filepath = @"E:\MSVS2022CDAC\Session7Demo\Session7Demo\abc.txt";
            string[] lines;

            //Check if file exist or not
            if (File.Exists(filepath))
            {
                //Read All lines of text file
                lines = File.ReadAllLines(filepath);
                foreach (string line in lines)
                {
                    Console.WriteLine(line);
                }
            }
            else
            {
                Console.WriteLine("File does not exist...");
            }
        }
    }
}
```

# Files

## { Reading text files}

```
using System.IO;
namespace Session11Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            string filepath = @"E:\MSVS2022CDAC\Session7Demo\Session7Demo\abc.txt";
            string lines;
            if (File.Exists(filepath))
            {
                //Read whole lines in text file at once and store in string variable
                lines = File.ReadAllText(filepath);
                Console.WriteLine(lines);
            }
            else
            {
                Console.WriteLine("File does not exist...");
            }
        }
    }
}
```

# Exercise

1. Write a C# code to copy the content of one file to another (hint : use File.Copy method )
  
2. Write a C# code to delete the existing file. (hint: use File.Delete method)

# Files

{ Writing into text files}

```
using System.IO;
namespace Session11Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            string filepath = @"E:\MSVS2022CDAC\Session7Demo\Session7Demo\abc.txt";
            string lines="Welcome to class";
            if (File.Exists(filepath))
            {
                //Write all content of string into file at once
                File.WriteAllText(filepath,lines);
                Console.WriteLine("Writing completed");
            }
            else
            {
                Console.WriteLine("File does not exist...");
            }
        }
    }
}
```

# Files

## { Writing into text files}

```
using System.IO;
namespace Session11Demo
{
    0 references
    public class Program
    {
        0 references
        static void Main(string[] args)
        {
            string filepath = @"E:\MSVS2022CDAC\Session7Demo\Session7Demo\abc.txt";
            string[] lines = { "India", "USA", "UK", "Finland" };
            if (File.Exists(filepath))
            {
                //Each entry in the string array will be a new line in the new file.
                File.WriteAllLines(filepath, lines);
                Console.WriteLine("Writing completed");
            }
            else
            {
                Console.WriteLine("File does not exist...");
            }
        }
    }
}
```

# Exercise

1. Try to append the contents in the existing file (hint : use File.AppendAllText or File.AppendAllLines)

# Stream

- The **stream** is basically the **sequence of bytes** passing through the communication path.
- There are two main streams: the **input stream** and the **output stream**. The **input stream** is used for reading data from file (read operation) and the **output stream** is used for writing into the file (write operation).

# Stream

## StreamWriter Class

The StreamWriter class implements **TextWriter** for writing character to stream in a particular format.

Methods	Description
<b>Close()</b>	Closes the current StreamWriter object and stream associate with it.
<b>Flush()</b>	Clears all the data from the buffer and write it in the stream associate with it.
<b>Write()</b>	Write data to the stream. It has different overloads for different data types to write in stream.
<b>WriteLine()</b>	It is same as Write() but it adds the newline character at the

# Stream

{ using StreamWriter}

```
using System.IO;
namespace Session11Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            StreamWriter sw = new StreamWriter("E://cdac.txt");
            // To write on the console screen
            Console.WriteLine("Enter the text to write in the File");
            // To read the input from the user
            string str = Console.ReadLine();
            // To write a line in buffer
            sw.WriteLine(str);
            // To write in output stream
            sw.Flush();
            // To close the stream
            sw.Close();
        }
    }
}
```

# Stream

## StreamReader Class

The StreamReader class implements TextReader for reading character from the stream in a particular format

Methods	Description
Close()	Closes the current StreamReader object and stream associate with it.
Peek()	Returns the next available character but does not consume it.
Read()	Reads the next character in input stream and increment characters position by one in the stream
ReadLine()	Reads a line from the input stream and return the data in form of string
Seek()	It is use to read/write at the specific location from a file

# Stream

{ using StreamReader}

```
using System.IO;
namespace Session11Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            // Taking a new input stream and opened it
            StreamReader sr = new StreamReader("E://cdac.txt");
            // specify the position to start reading input stream
            sr.BaseStream.Seek(0, SeekOrigin.Begin);
            string str = sr.ReadLine();
            // To read the whole file line by line
            while (str != null)
            {
                Console.WriteLine(str); str = sr.ReadLine();
            }
            // to close the stream
            sr.Close();
        }
    }
}
```

# Working with Directories and Drives

## Directory class

**Directory** class in the .NET Framework class library provides static methods for creating, copying, moving, and deleting directories and subdirectories. Before you can use the **Directory** class, you must import the **System.IO** namespace.

```
using System.IO;
namespace Session11Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            string rootdir = @"E:\CDACFolder";
            // If directory does not exist, create it.
            if (!Directory.Exists(rootdir))
            {
                Directory.CreateDirectory(rootdir);
            }
        }
    }
}
```

# Exercise

1. Check the directory exists or not and then delete it. (use delete method)
2. Move an existing directory to a new specified directory with full path. (use move method)

# Working with Directories and Drives

GetLogicalDrives  
method **returns all**  
**the logical drives** on a  
system.

```
using System.IO;
namespace Session11Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            string[] drives = Directory.GetLogicalDrives();
            foreach (string drive in drives)
            {
                System.Console.WriteLine(drive);
            }
        }
    }
}
```

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-12**

## **Threading and Task**

# Contents

- Threading
  - ThreadStart, Parameterized ThreadStart
  - Synchronizing critical data using lock and Monitor
- Working with Tasks
- Calling functions with and without return values
  - Using async and await

# Threads

**Threads** are programs line of execution or smallest unit of processing that can be scheduled by an operating system

## Process v/s Thread

- A **process** represents an application whereas a **thread** represents a module of the application.
- **Process** is **heavyweight** component whereas **thread** is **lightweight**. A **thread** can be termed as lightweight sub-process because it is executed inside a process.
- Whenever you create a **process**, a separate memory area is **occupied**. But **threads** share a **common memory area**.

# Thread life cycle

- The life cycle of a thread is started when instance of **System.Threading.Thread class** is created. When the task execution of the thread is completed, its life cycle is ended.
- Following states in the life cycle of a Thread:
  - **Unstarted** - When the instance of Thread class is created
  - **Runnable (Ready to run)** - When start() method on the thread is called
  - **Running** - Only one thread within a process can be executed at a time
  - **Not Runnable** - if sleep() or wait() method is called on the thread, or input/output operation is blocked
  - **Dead (Terminated)** - After completing the task, thread enters into dead or terminated state.

# Thread class

- Thread class provides **properties** and **methods** to **create** and **control** threads.
- It is found in **System.Threading** namespace.

Property	Description
CurrentThread	returns the instance of currently running thread.
IsAlive	checks whether the current thread is alive or not. It is used to find the execution status of the thread.
IsBackground	is used to get or set value whether current thread is in background or not.
Name	is used to get or set the name of the current thread.
Priority	is used to get or set the priority of the current thread.
ThreadState	is used to return a value representing the thread state.

# Thread class

Methods	Description
Abort()	is used to terminate the thread. It raises ThreadAbortException.
Join()	is used to block all the calling threads until this thread terminates.
Resume()	is used to resume the suspended thread. It is obsolete.
Sleep(Int32)	is used to suspend the current thread for the specified milliseconds.
Start()	changes the current state of the thread to Runnable.
Suspend()	suspends the current thread if it is not suspended. It is obsolete.
Yield()	is used to yield the execution of current thread to another thread.

# Types of Thread

## Foreground Thread

- A thread which **keeps on running to complete its work** even if the **Main thread leaves its process**, this type of thread is known as foreground thread.
- Foreground thread **does not care** whether the **main thread is alive or not**, it completes only when it finishes its assigned work Or in other words, the life of the foreground thread does not depend upon the main thread.

# Types of Thread

## Foreground Thread

```
using System.Threading;
namespace Session12Demo
{
    0 references
    public class Program
    {
        1 reference
        static void getThreadInformation()
        {
            for (int i=1; i <=10; i++)
            {
                Console.WriteLine("getThreadInformation is in progress !!");
                Thread.Sleep(1000);
            }
            Console.WriteLine("getThreadInformation ends!!!");
        }
        0 references
        static void Main(string[] args)
        {
            Thread t1=new Thread(getThreadInformation);
            t1.Start();
            Console.WriteLine("Main Thread Ends!!!");
        }
    }
}
```

# Types of Thread

## Background Thread

- A **thread which leaves its process when the Main method leaves its process**, these types of the thread are known as the background threads.
- The **life** of the background thread **depends upon the life of the main thread**. If the main thread finishes its process, then background thread also ends its process.

# Types of Thread

## Background Thread

```
using System.Threading;
namespace Session12Demo
{
    public class Program
    {
        static void getThreadInformation()
        {
            Console.WriteLine("In progress thread is:" + Thread.CurrentThread.Name);
            Thread.Sleep(2000);
            Console.WriteLine("Completed thread is:" + Thread.CurrentThread.Name);
        }

        static void Main(string[] args)
        {
            Thread t1=new Thread(getThreadInformation);
            t1.Name = "MyThread1";
            t1.Start();
            Console.WriteLine("Main Thread Ends!!!");
        }
    }
}
```

# ParameterizedThreadStart

- **Thread(ParameterizedThreadStart) Constructor** is used to initialize a new instance of the Thread class.
- It defined a delegate which allows an object to pass to the thread when the thread starts.
- This constructor gives *ArgumentNullException* if the parameter of this constructor is null.

# ParameterizedThreadStart

```
using System.Threading;
namespace Session12Demo
{
    0 references
    public class Program
    {
        1 reference
        static void getThreadInformation()
        {
            for (int i = 1; i <=10; i++)
            {
                Console.WriteLine("In progress thread is:" + Thread.CurrentThread.Name);
                Thread.Sleep(1000);
            }
        }
        0 references
        static void Main(string[] args)
        {
            // Thread(ParameterizedThreadStart) constructor with static method
            Thread t1 =new Thread(getThreadInformation);
            t1.Name = "MyThread1";
            t1.Start();
        }
    }
}
```

# ThreadStart

- *ThreadStart* is a delegate which represents a method to be invoked when this thread begins executing.
- **Thread(ThreadStart) Constructor** is used to initialize a new instance of a Thread class. This constructor will give *ArgumentNullException* if the value of the parameter is null.
- Syntax :

```
public Thread(ThreadStart start);
```

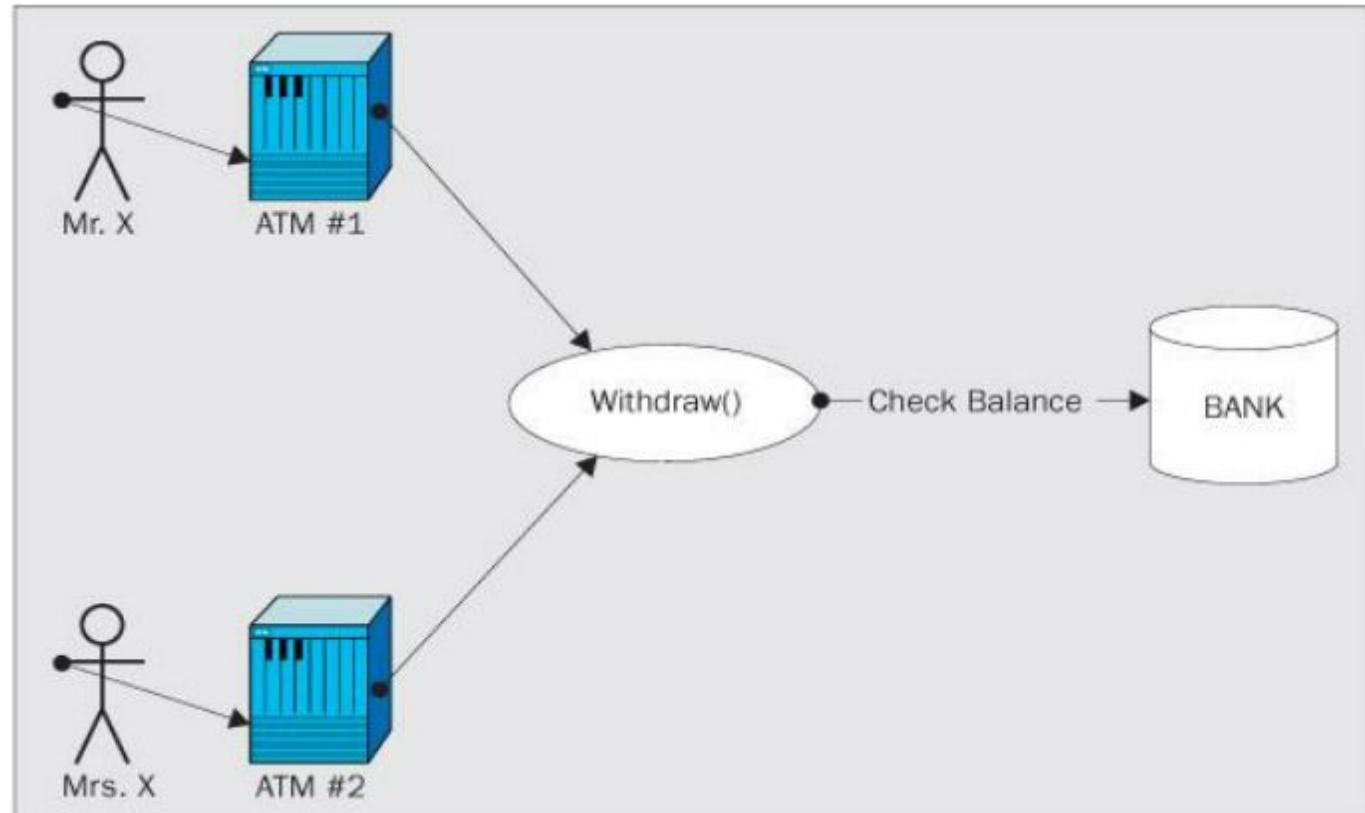
# ThreadStart

```
namespace Session12Demo
{
    public class Program
    {
        static void getThreadInformation()
        {
            for (int i = 1; i <=10; i++)
            {
                Console.WriteLine("In progress thread is:" + Thread.CurrentThread.Name);
                Thread.Sleep(1000);
            }
        }

        static void Main(string[] args)
        {
            // Thread(ThreadStart) constructor with static method
            Thread t1 = new Thread(new ThreadStart(getThreadInformation));
            t1.Name = "MyThread1";
            t1.Start();
        }
    }
}
```

# Thread Synchronization

- Synchronization is a technique that allows **only one thread** to access the resource for the particular time. **No other thread** can interrupt until the assigned thread finishes its task.
- Advantages:
  - Consistency Maintain
  - No Thread Interference



Money withdraw from the same bank account at the same time

# Thread Synchronization

## Synchronizing critical data using lock :

- **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

# Synchronizing critical data using lock

```
using System.Threading;
namespace Session12Demo
{
    2 references
    public class Program
    {
        2 references
        public void getThreadInformation()
        {
            lock(this)
            {
                for (int i = 1; i <=10; i++)
                {
                    Thread.Sleep(1000);
                    Console.WriteLine("i=" + i + " " + " Thread Name: " + Thread.CurrentThread.Name);
                }
            }
        }

        static void Main(string[] args)
        {
            Program obj=new Program();
            Thread t1 = new Thread(new ThreadStart(obj.getThreadInformation));
            Thread t2 = new Thread(new ThreadStart(obj.getThreadInformation));
            t1.Name = "MyThread1"; t2.Name = "MyThread2";
            t1.Start(); t2.Start();
        }
    }
}
```

# Monitor class

- Provides a mechanism that **synchronizes access to objects**.
- It can be done by **acquiring a significant lock** so that **only one thread can enter** in a given piece of code **at one time**.
- Monitor is not different from lock but the monitor class **provides more control over the synchronization** of various threads trying to access the same block of code.
- The Monitor class has the following methods for the synchronize access to a region of code by taking and releasing a lock:
  - Monitor.Enter
  - Monitor.TryEnter
  - Monitor.Exit.

# Monitor class

```
using System.Threading;
namespace Session12Demo
{
    public class Program
    {
        public static object locker = new object();
        public static void PrintNum()
        {
            Monitor.Enter(locker);
            try
            {
                for (int i = 1; i <= 10; i++)
                {
                    Thread.Sleep(1000);
                    Console.WriteLine(i.ToString());
                }
            }
            finally
            {
                Monitor.Exit(locker);
            }
        }
    }
}
```

```
static void Main(string[] args)
{
    Thread t1=new Thread(new ThreadStart(PrintNum));
    Thread t2 = new Thread(new ThreadStart(PrintNum));
    t1.Start();
    t2.Start();
}
```

# Working with Tasks

- the task is basically used to implement Asynchronous Programming i.e. executing operations asynchronously
- Task-related classes are in **System.Threading.Tasks** namespace.
- In a performance point of view, the **Task.Run** or **Task.Factory.StartNew** methods are preferable to create and schedule the computational tasks.

# Working with Tasks

```
using System.Threading;
namespace Session12Demo
{
    0 references
    public class Program
    {
        3 references
        public static void PrintNum()
        {
            Console.WriteLine("Child Thread :" + Thread.CurrentThread.ManagedThreadId + " started..");
            for (int i = 1; i <= 10; i++)
            {
                Console.WriteLine(i);
            }
            Console.WriteLine("Child Thread :" + Thread.CurrentThread.ManagedThreadId + " finished..");
        }
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Main Threads Started");
            // Create a Task
            Task t1 = new Task(PrintNum);
            t1.Start();

            // Create a started task using Factory
            Task t2 = Task.Factory.StartNew(PrintNum);

            // Creating a started task using Task.Run
            Task t3 = Task.Run(() => { PrintNum(); });

            Console.WriteLine("Main Threads Completed");
        }
    }
}
```

# Task return type

Using `async` and `await`

- **Async methods** that **don't contain a return statement** or that **contain a return statement that doesn't return an operand** usually have a **return type of Task**.
- If you use a **Task return type** for an **async method**, a **calling method** can use an **await operator** to **suspend the caller's completion until the called async method has finished**.

# Task return type

## Using async and await

```
using System.Threading;
namespace Session12Demo
{
    0 references
    public class Program
    {
        1 reference
        public static async Task DisplayCurrentInfoAsync()
        {
            try
            {
                Console.WriteLine("I am in display information.");
                Console.WriteLine("Sorry for the delay...");
            }catch (Exception ex)
            {
                Console.WriteLine(ex.ToString());
            }
            await Task.Delay(1000);
        }
    }
}
```

```
static void Main(string[] args)
{
    Task t1=Task.Run(() => {DisplayCurrentInfoAsync();})
}
```

# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development  
Fourth Edition by Mark J. Price
2. <https://www.geeksforgeeks.org>

# **Session-13**

**Introduction to ASP.NET**

# Contents

- What is ASP.NET?
- ASP.NET and earlier Web Development platforms
- Seven important facts about ASP.NET
- Web Architecture
- ASP.NET page life cycle
- Validation controls

# What is ASP.NET?

ASP.NET is a server-side **technology** for building powerful, dynamic Web applications and is part of the .NET Framework

# ASP.NET and earlier web development platforms

- ✓ ASP.NET features a **completely object-oriented programming model**, which includes an event-driven, control-based architecture that encourages code encapsulation and code reuse.
- ✓ ASP.NET gives you the **ability to code in any supported .NET language** (including Visual Basic, C#, J#, and many other languages that have third-party compilers)

# ASP.NET and earlier web development platforms

- ✓ ASP.NET is also a platform for **building web services**, which are reusable units of code that other applications can call across platform and computer boundaries.
- ✓ ASP.NET is dedicated **to high performance**. ASP.NET pages and components are compiled on demand instead of being interpreted every time they're used. ASP.NET also includes a finetuned data access model and flexible data caching to further boost performance.

# Seven Important Facts About ASP.NET

Fact 1: ASP.NET Is Integrated with the .NET Framework

Fact 2: ASP.NET Is Compiled, Not Interpreted

Fact 3: ASP.NET is Multilanguage

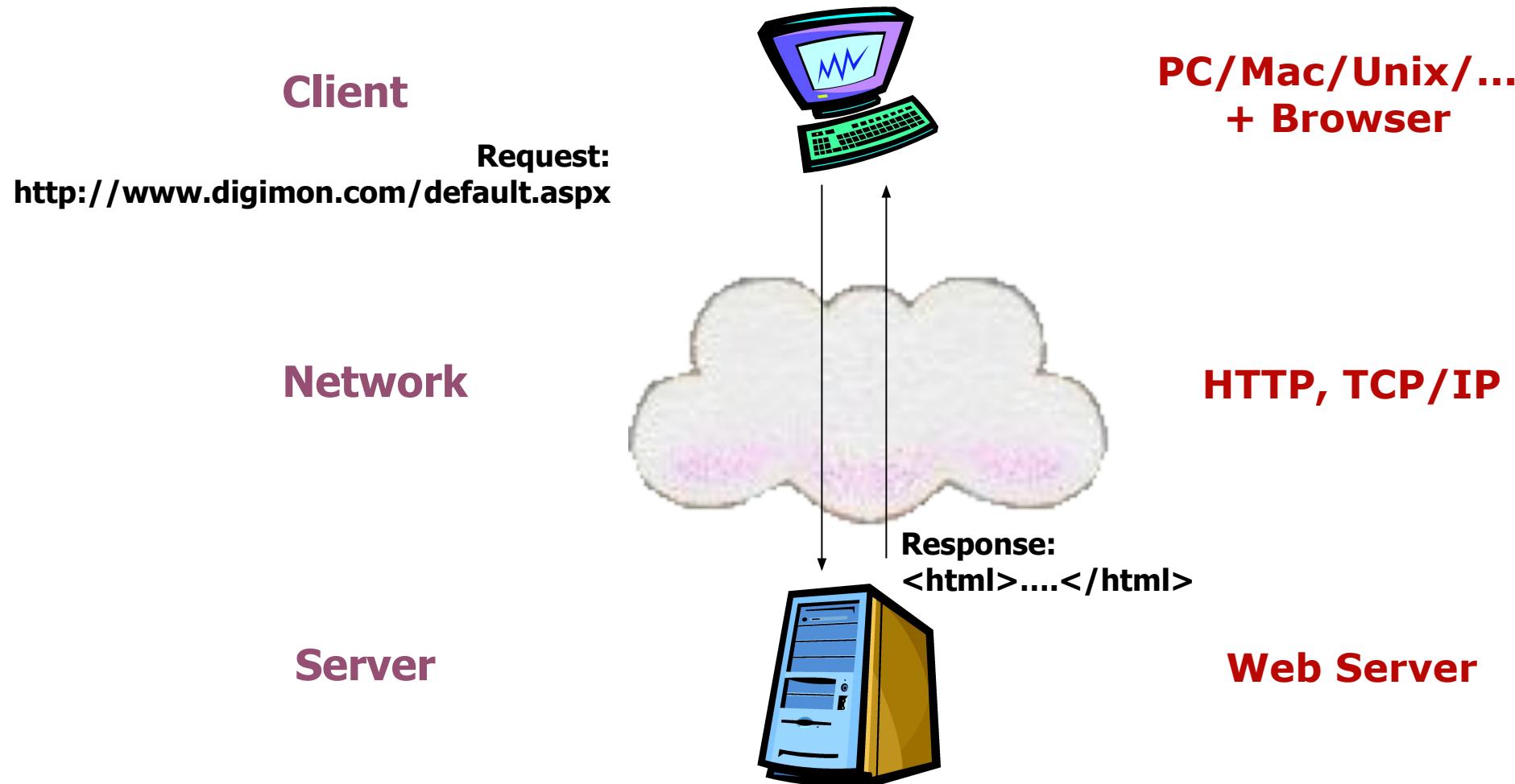
Fact 4: ASP.NET runs inside the CLR

Fact 5: ASP.NET is Object Oriented

Fact 6: ASP.NET is Multidevice and Multibrowser

Fact 7: ASP.NET is Easy to Deploy and configure

# Web Architecture



# ASP.NET Page Life Cycle

Srno	Stages	Description
1.	<b>Page request</b>	When the page is requested by a user, ASP.NET determines whether the page needs to be parsed and compiled (therefore beginning the life of a page)
2.	<b>Start</b>	Page properties such as <b>Request</b> and <b>Response</b> are set. At this stage, the page also determines whether the request is a postback or a new request and sets the <b>IsPostBack</b> property
3.	<b>Initialization</b>	During page initialization, controls on the page are available and each control's <b>UniqueID</b> property is set. A master page and themes are also applied to the page if applicable.
4	<b>Load</b>	During load, if the current request is a postback, control properties are loaded with information recovered from <b>view state</b> and <b>control state</b> .

# ASP.NET Page Life Cycle

Srno	Stages	Description
5.	<b>Postback event handling</b>	If the request is a postback, control event handlers are called. After that, the <b>Validate</b> method of all validator controls is called, which sets the <b>IsValid</b> property of individual validator controls and of the page
6.	<b>Rendering</b>	Before rendering, view state is saved for the page and all controls. During the rendering stage, the page calls the <b>Render</b> method for each control, providing a <b>text writer</b> that writes its output to the <b>OutputStream</b> object of the page's <b>Response</b> property.
7.	<b>Unload</b>	The Unload event is raised after the page has been fully rendered, sent to the client, and is ready to be discarded. At this point, page properties such as Response and Request are unloaded and cleanup is performed.

# Validation Controls

- A Validation server control is used to validate the data of an input control.
- If the data does not pass validation, it will display an error message to the user.
- Syntax:  
`<asp:control_name id="some_id" runat="server" />`

# Validation Controls

Validation Server Control	Description
<a href="#"><u>RequiredFieldValidator</u></a>	Makes an input control a required field
<a href="#"><u>RangeValidator</u></a>	Checks that the user enters a value that falls between two values
<a href="#"><u>CompareValidator</u></a>	Compares the value of one input control to the value of another input control or to a fixed value
<a href="#"><u>RegularExpressionValidator</u></a>	Ensures that the value of an input control matches a specified pattern
<a href="#"><u>CustomValidator</u></a>	Allows you to write a method to handle the validation of the value entered
<a href="#"><u>ValidationSummary</u></a>	Displays a report of all validation errors occurred in a Web page

# Validation Controls

**RequiredFieldValidator:** ensures that the required **field is not empty**. It is generally tied to a text box to force input into the text box.

Syntax:

```
<asp:RequiredFieldValidator  
    ID="rfvusername"  
    runat="server"  
    ControlToValidate =“txtUsername”  
    ErrorMessage="*”>  
</asp:RequiredFieldValidator>
```

# Validation Controls

- The **RangeValidator** control verifies that the input value falls within a predetermined range.
- It has three specific properties:

Properties	Description
Type	it defines the type of the data; the available values are: Currency, Date, Double, Integer and String
MinimumValue	it specifies the minimum value of the range
MaximumValue	it specifies the maximum value of the range

```
<asp:RangeValidator ID="rvclass"
runat="server"
ControlToValidate="txtclass"
ErrorMessage="Enter your class (6 -
12)"
MaximumValue="12"
MinimumValue="6"
Type="Integer">
</asp:RangeValidator>
```

# Validation Controls

- The **CompareValidator** control compares a value in one control with a fixed value, or, a value in another control.
- It has specific properties:

Properties	Description
Type	it specifies the data type
ControlToCompare	it specifies the value of the input control to compare with
ValueToCompare	it specifies the constant value to compare with
Operator	it specifies the comparison operator, the available values are: Equal, NotEqual, GreaterThan, GreaterThanEqual, LessThan, LessThanEqual and DataTypeCheck

# Validation Controls

## CompareValidator control :

Syntax:

```
<asp:CompareValidator  
    ID="CompareValidator1"  
    runat="server"  
    ErrorMessage="Password Must Be Same"  
    ControlToValidate="txtConfirmPwd"  
    ControlToCompare="txtChoosePwd">  
</asp:CompareValidator>
```

# Validation Controls

- The **RegularExpressionValidator** allows validating the input text by matching against a **pattern against a regular expression**. The regular expression is set in the ValidationExpression property.
- Syntax:

```
<asp:RegularExpressionValidator  
    ID="RegularExpressionValidator1"  
    runat="server"  
    ErrorMessage="Please Enter Valid Email"  
    ControlToValidate="txtEmail"  
    ValidationExpression="\w+([-.\'])\w+@\w+([- .])\w+*\.\w+([-.\'])\w+*"  
</asp:RegularExpressionValidator>
```

# Validation Controls

- The CustomValidator control allows writing application specific **custom validation routines** for both the **client side** and the **server side** validation.
- The client side validation is accomplished through the ***ClientValidationFunction*** property. The client side validation routine should be written in a scripting language, like **JavaScript** or **VBScript**, which the browser can understand.
- The server side validation routine must be called from the controls ***ServerValidate*** event handler. The server side validation routine should be written in any .Net language, like **C#** or **VB.Net**.

# Validation Controls

- Syntax:

```
<asp:CustomValidator  
    ID="CustomValidator1"  
    runat="server"  
    ErrorMessage="Password Should not be less than 5 character"  
    ControlToValidate="txtChoosePwd"  
    ClientValidationFunction="validateText">  
</asp:CustomValidator>
```

# Validation Controls

- The **ValidationSummary** control does not perform any validation but **shows a summary of all errors** in the page.
- The summary displays the values of the ErrorMessage property of all validation controls that failed validation.
- Syntax:

```
<asp:ValidationSummary  
    ID="ValidationSummary1"  
    runat="server"  
    DisplayMode = "BulletList" />
```

# References

1. C .Net Web Developers Guide by Syngress
2. ASP.NET 4.5, Covers C# and VB Codes, Black Book , Kogent Learning Solutions Inc.

# **Session-14**

**Introduction to ASP.NET MVC**

# Contents

- ASP.NET MVC Architecture
- ASP.NET MVC folder structure
- Creating controllers
- Action Methods
- Different Types of Action Results

# ASP.NET MVC Architecture

- MVC stands for **Model, View, and Controller**.
- MVC is a **standard design pattern**
- The ASP.NET MVC framework is a **lightweight, highly testable presentation framework** that (as with Web Forms-based applications) is integrated with existing ASP.NET features, such as master pages
- The MVC framework is defined in the **System.Web.Mvc** namespace and is a fundamental, supported part of the **System.Web** namespace.

# ASP.NET MVC Architecture

## Models.

- Model objects are the parts of the application that **implement the logic** for the **applications** data domain.
- Often, model objects retrieve and store model state in a database.
- For example, a **Product** object might retrieve information from a database, **operate on it**, and then **write updated information back** to a Products table in SQL Server.

# ASP.NET MVC Architecture

## View

- View in MVC is a **user interface**.
- View **display model data** to the **user** and also enables them to modify them.
- View in ASP.NET MVC is HTML, CSS, and some special syntax (Razor syntax) that makes it easy to **communicate with the model** and the **controller**.

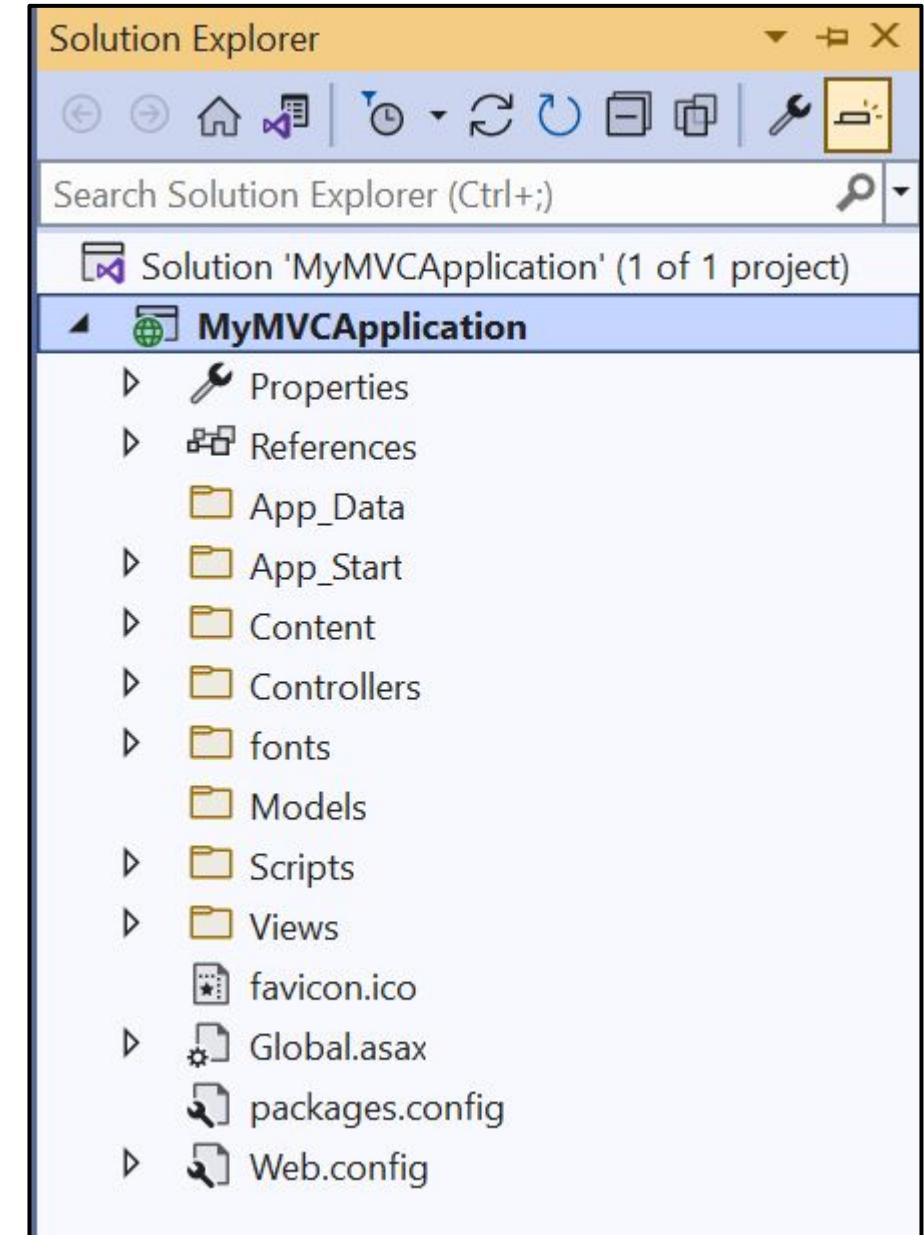
# ASP.NET MVC Architecture

## Controller:

- The controller **handles** the user **request**.
- Typically, the user **uses the view** and raises an **HTTP request**, which will be handled by the controller.
- The controller **processes the request** and **returns the appropriate view** as a **response**.
- Controller is the **request handler**

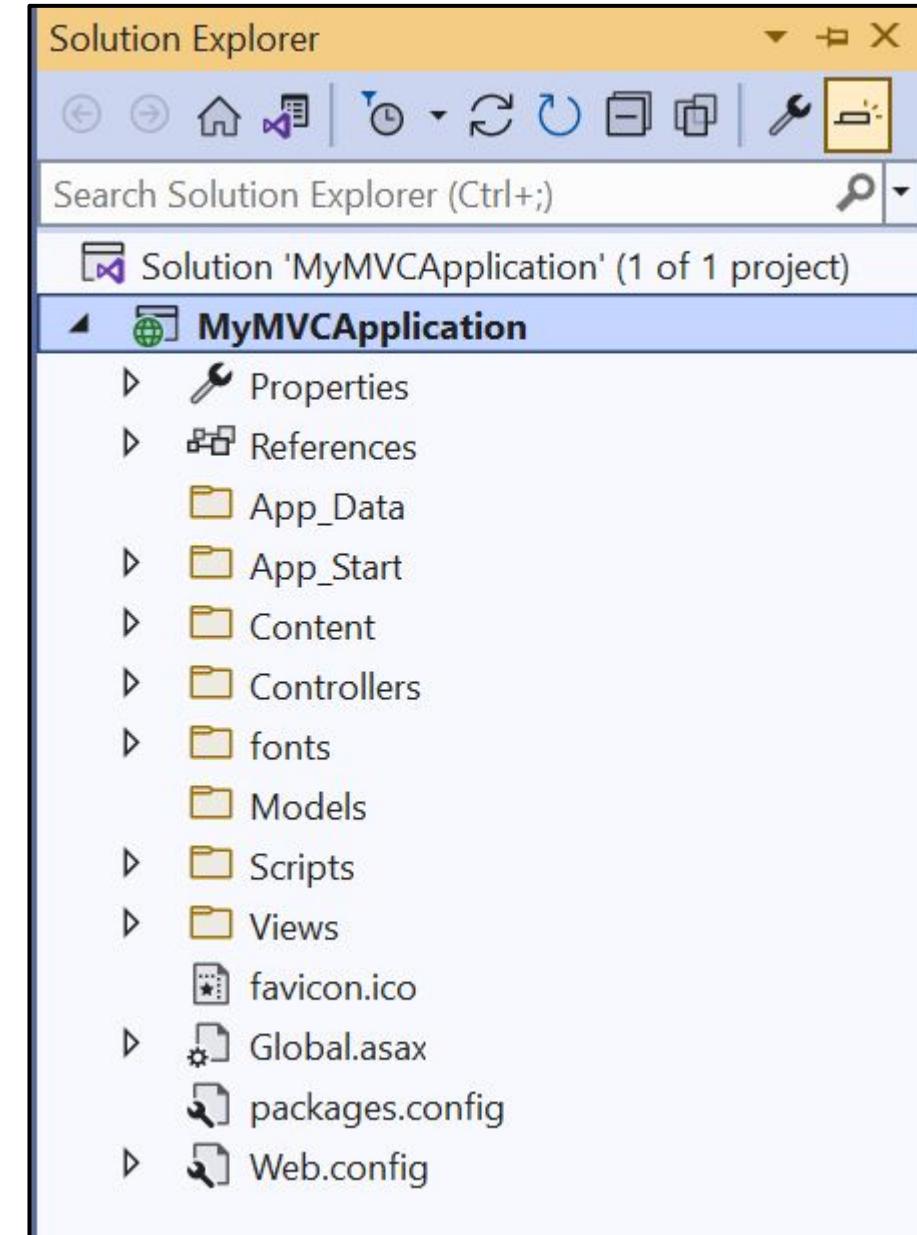
# ASP.NET MVC folder structure

- **App\_Data** : folder can contain application data files like LocalDB, .mdf files, XML files, and other data related files.
- **App\_Start** : folder can contain class files that will be executed when the application starts
- **Content** : folder contains static files like CSS files, images, and icons files. MVC application includes bootstrap.css, bootstrap.min.css, and Site.css by default.



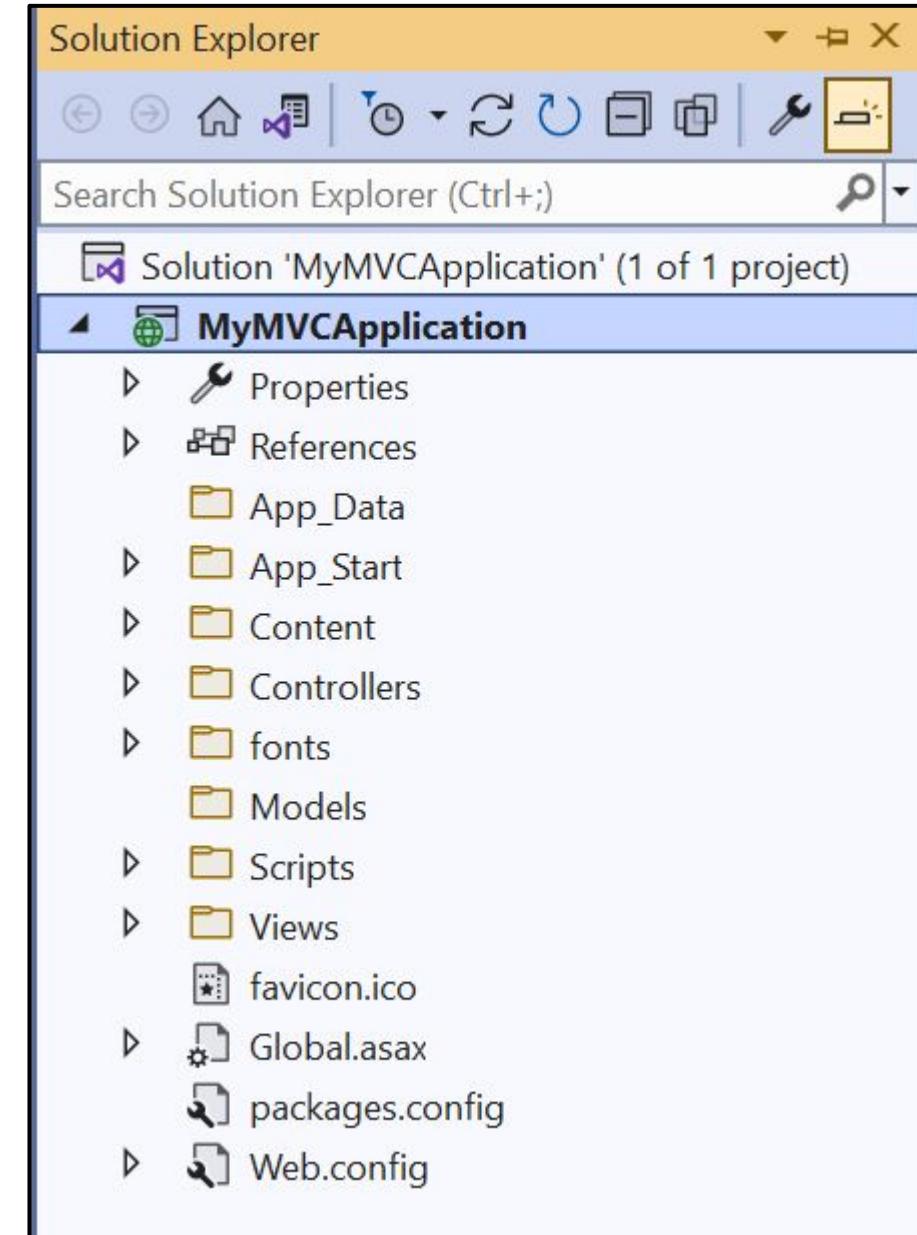
# ASP.NET MVC folder structure

- **Controllers:** folder contains class files for the controllers. A controller handles users' request and returns a response. MVC requires the name of all controller files to end with "Controller". Eg. *HomeController*
- **Fonts :** folder contains custom font files for your application.
- **Models :** folder contains model class files. Typically model class includes public properties, which will be used by the application to hold and manipulate application data.



# ASP.NET MVC folder structure

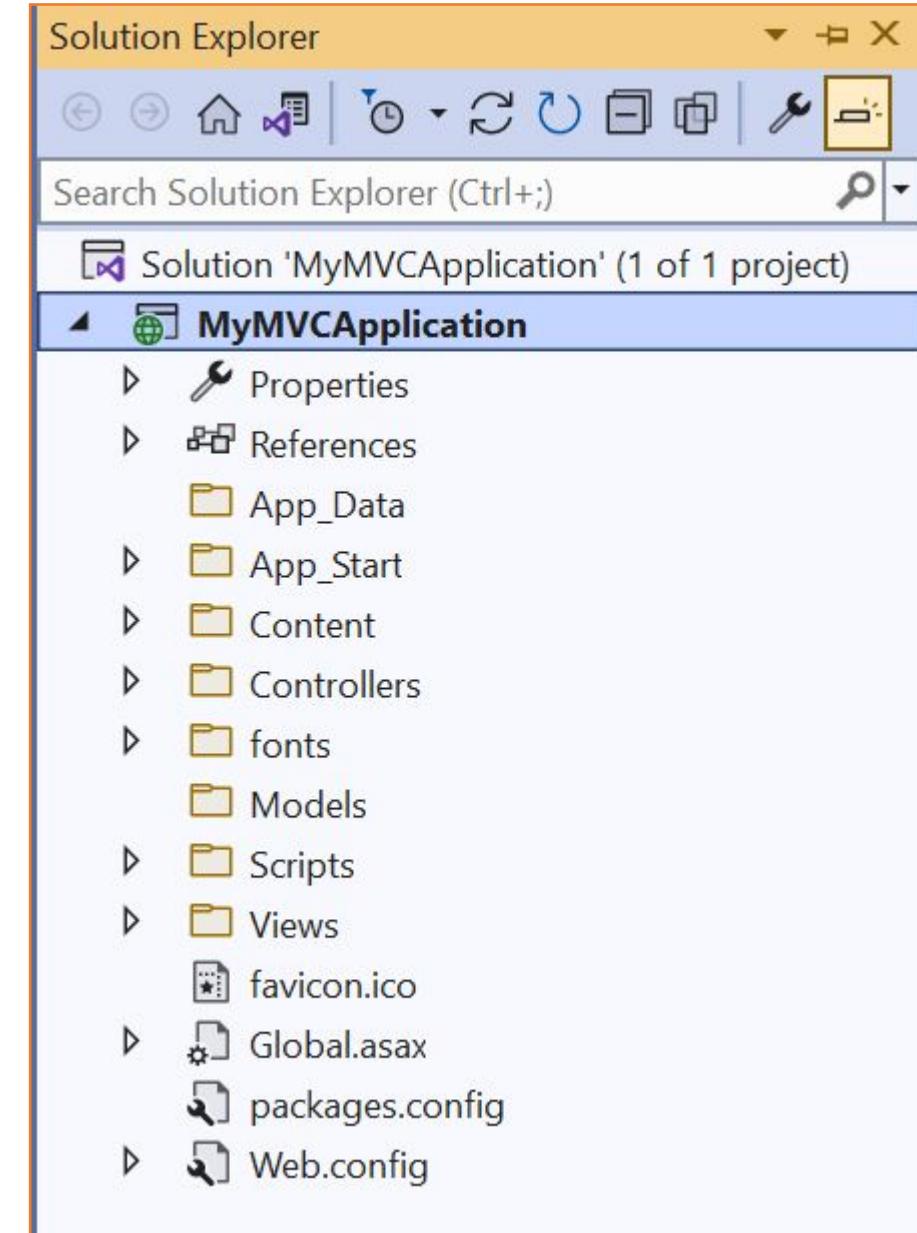
- **Scripts** folder contains JavaScript or VBScript files for the application.
- **Views** folder contains HTML files for the application. Typically view file is a .cshtml file where you write HTML and C# code. The Views folder includes a separate folder for each controller.
- For example, all the .cshtml files, which will be rendered by HomeController will be in View -> Home folder. The Shared folder under the View folder contains all the views shared among different controllers e.g., layout files.



# ASP.NET MVC folder structure

## Configuration files :

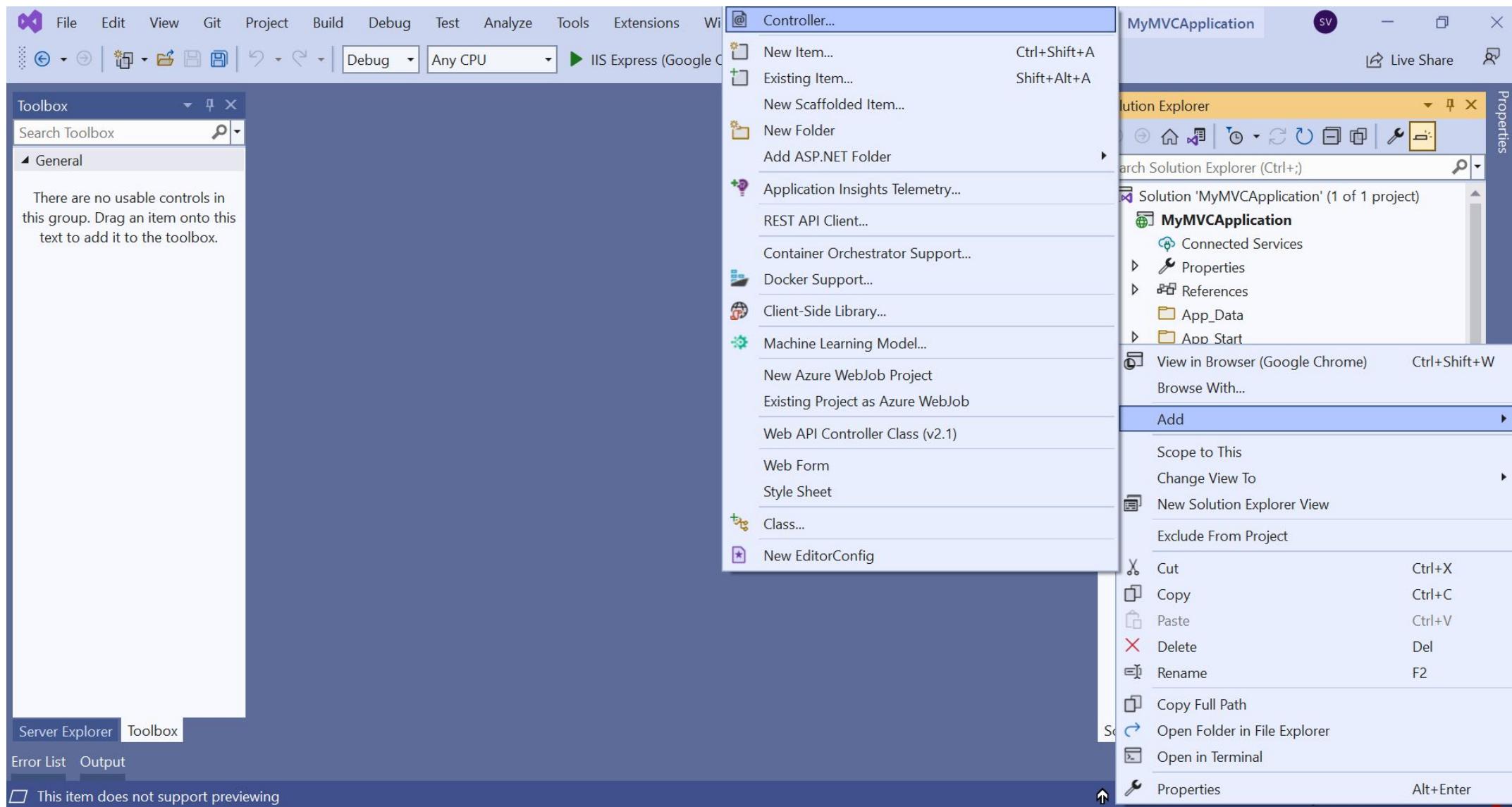
- **Global.asax** : file allows you to write code that runs in response to application-level events, such as *Application\_BeginRequest*, *application\_start*, *application\_error*, *session\_start*, *session\_end*, etc.
- **Packages.config** : file is managed by NuGet to track what packages and versions you have installed in the application.
- **Web.config** : file contains application-level configurations.



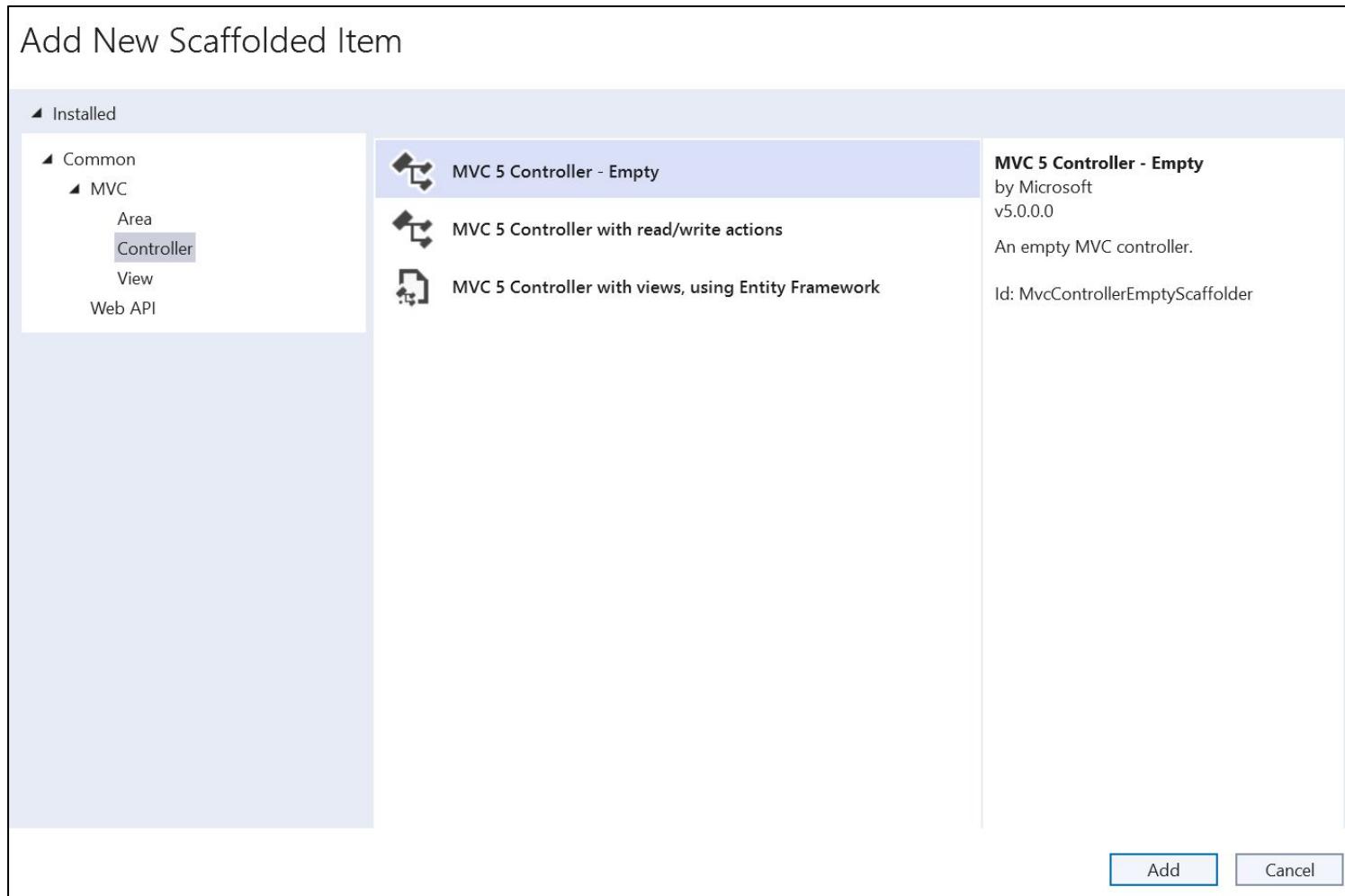
# Creating Controllers

- The Controller in MVC architecture **handles any incoming URL request**
- Controller is a class derived from the base class **System.Web.Mvc.Controller**
- Controller class contains public methods called **Action** methods.
- **Controller and its action method handles incoming browser requests,** retrieves necessary model data and returns appropriate responses.

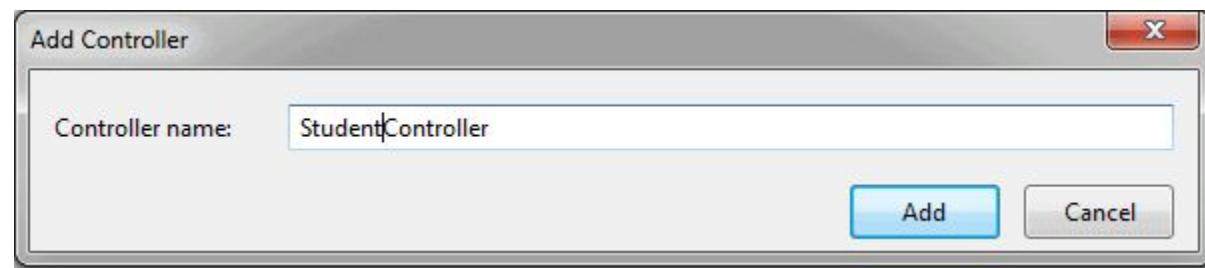
# Creating the Controller



# Creating Controller



# Creating Controller



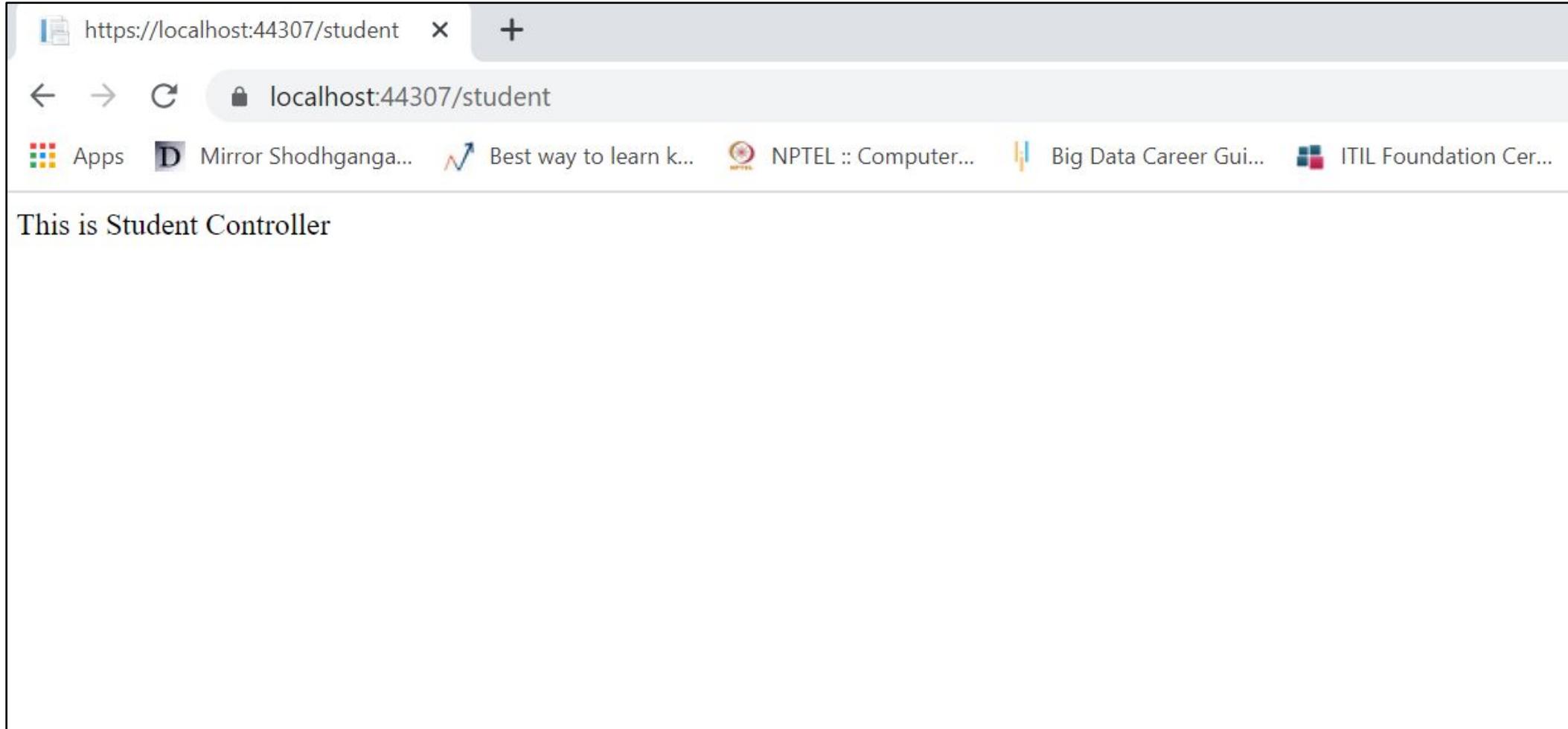
# Example : StudentController

The screenshot shows the Microsoft Visual Studio interface with the following details:

- Code Editor:** The main window displays the file `StudentController.cs`. The code defines a controller named `StudentController` that inherits from `Controller`. It contains a single action method `Index()` which returns the string "This is Student Controller".

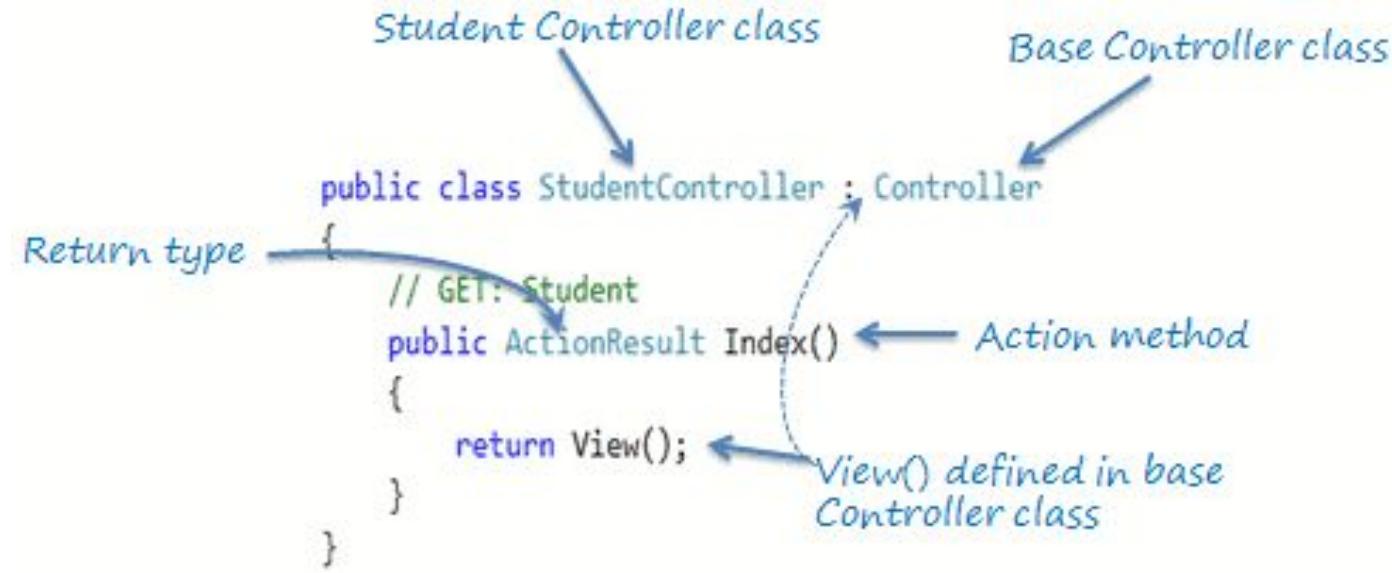
```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Web;
5  using System.Web.Mvc;
6
7  namespace MyMVCApplication.Controllers
8  {
9      public class StudentController : Controller
10     {
11         // GET: Student
12         public string Index()
13         {
14             return "This is Student Controller";
15         }
16     }
17 }
```
- Solution Explorer:** On the right, the Solution Explorer shows the project structure for "MyMVCApplication". It includes the following items:
  - Connected Services
  - Properties
  - References
    - App\_Data
    - App\_Start
    - Content
  - Controllers
    - HomeController.cs
    - StudentController.cs (highlighted)
  - Fonts
  - Models
  - Scripts
  - Views
    - Home
    - Shared
    - Student
      - \_ViewStart.cshtml
      - Web.config
      - favicon.ico
  - Global.asax
- Status Bar:** At the bottom, it shows "100 %", "No issues found", "Ln: 17 Ch: 2 SPC CRLF", and tabs for "Solution Explorer" and "Git Changes".

# Example : StudentController

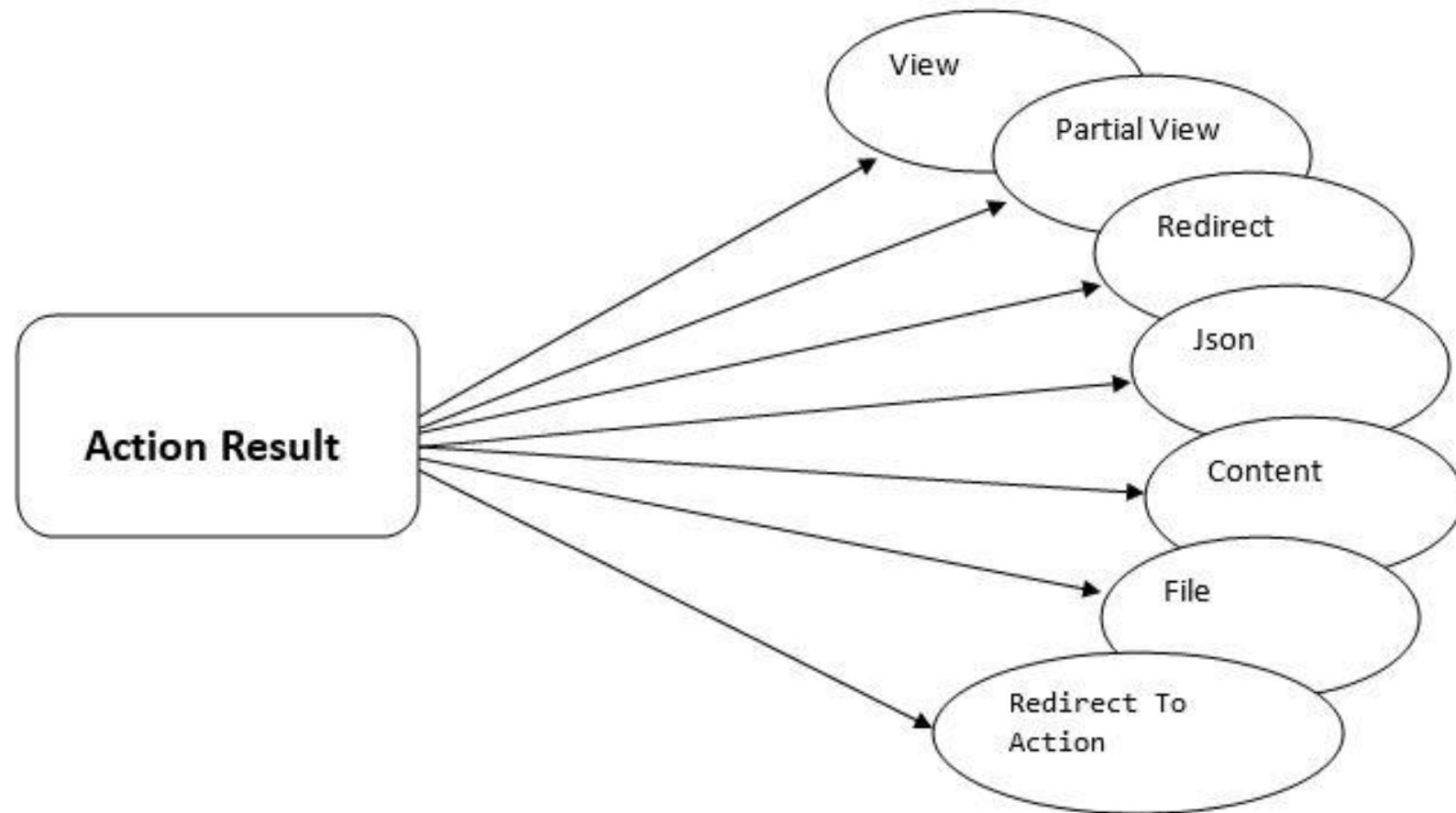


# Action Methods

- All the public methods of the Controller class are called Action methods.
- They are like any other normal methods with the following restrictions:
  - Action method must be public. It cannot be private or protected
  - Action method cannot be overloaded
  - Action method cannot be a static method.



# Different Types Of Action Results In ASP.NET MVC



# View Result

View result is a basic view result. It returns basic results to view page. View result can return data to view page through which class is defined in the model. View page is a simple HTML page.

```
public ViewResult Gallary()
{
    ViewBag.Message = "This is about gallary page";
    return View();
}
```

# PartialView Result

Partial View Result is returning the result to Partial view page. Partial view is one of the views that we can call inside Normal view page.

We should create a Partial view inside shared folder, otherwise we cannot access the Partial view.

```
public PartialViewResult Index()
{
    return PartialView("_PartialView");
}
```

# Redirect Result

- Redirect result is returning the result to specific URL. It is rendered to the page by URL.

```
public RedirectToResult Index()
{
    return Redirect("Home/Contact");
}
```

# Redirect to Action Result

Redirect to Action result is returning the result to a specified controller and action method. Controller name is optional in Redirect to Action method. If not mentioned, Controller name redirects to a mentioned action method in current Controller

```
public ActionResult Index()
{
    return RedirectToAction("GetStudents", "Student");
}
```

# Json Result

Json (JavaScript Object Notation) result is a significant Action Result in MVC. It will return simple text file format and key value pairs.

```
public JsonResult GetAllStudents()
{
    List<Student> students = new List<Student>()
    {
        new Student
        {
            StudentId = 1,
            StudentName = "Alex"
        },
        new Student
        {
            StudentId = 2,
            StudentName = "Smith"
        }
    };

    return Json(students, JsonRequestBehavior.AllowGet);
}
```

# File Result

File Result returns different file format view page when we implement file download concept in MVC using file result.

```
public ActionResult Index()
{
    return File("Web.Config", "text");
}
```

# Content Result

Content result returns different content's format to view. MVC returns different format using content return like HTML format, Java Script format and any other format.

```
public ActionResult Index()
{
    return Content("<script>alert('Welcome To All');</script>");
}
```

# References

1. C .Net Web Developers Guide by Syngress
2. ASP.NET 4.5, Covers C# and VB Codes, Black Book , Kogent Learning Solutions Inc.

# **Session-15**

## **Understanding Views and State management**

# Contents

- Razor View Engine
- HTML Helpers
  - Standard HTML Helper methods
  - Strongly-Typed HTML Helper
- Validations using Data Annotation Attributes
- State Management
  - Client side state management
  - Server side State management

# Razor View Engine

- Razor is one of the view engines supported in ASP.NET MVC. Razor allows you **to write a mix of HTML and server-side code using C# or Visual Basic**
- The razor view uses **@** character to include the server-side code. You can use C# or Visual Basic syntax to write server-side code inside the razor view.

# Razor View Engine

ASP.NET MVC supports the following types of razor view files:

File extension	Description
.cshtml	C# Razor view. Supports C# code with html tags.
.vbhtml	Visual Basic Razor view. Supports Visual Basic code with html tags.
.aspx	ASP.Net web form
.ascx	ASP.NET web control

# Razor View Engine

```
@model IEnumerable<MVC_BasicTutorials.Models.Student>
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Index</h2>
<p>@Html.ActionLink("Create New", "Create")</p>
<table class="table">
    <thead>
        <tr>
            <th>@Html.DisplayNameFor(model => model.StudentName)</th>
            <th>@Html.DisplayNameFor(model => model.Age)</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>@Html.DisplayFor(modelItem => item.StudentName)</td>
                <td>@Html.DisplayFor(modelItem => item.Age)</td>
                <td>@Html.ActionLink("Edit", "Edit", new { id=item.StudentId }) |</td>
            </tr>
        }
    </tbody>
</table>
```

*Razor Syntax*

*Html*

*Html helper*

# HTML Helpers

- **HtmlHelper** class renders HTML controls in **the razor view**.
- It binds the model object to HTML controls to display the value of model properties into those controls and also assigns the value of the controls to the model properties while submitting a web form.
- So we can use the **HtmlHelper** class in razor view instead of writing HTML tags manually.
- **@Html** is an object of the **HtmlHelper** class

```
@model IEnumerable<MVC_BasicTutorials.Models.Student>
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Index</h2>

<p> @Html.ActionLink("Create New", "Create") </p>
<table class="table">
    <tr>
        <th> @Html.DisplayNameFor(model => model.StudentName) </th>
        <th> @Html.DisplayNameFor(model => model.Age) </th>
    </tr>
```



# Form Design using HTML Helpers Methods

// Index.cshtml

```
@model Session15DemoMVC.Models.Student
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    @using (Html.BeginForm("Form1", "Student", FormMethod.Post))
    {
        @Html.LabelFor(m => m.StudentId); @Html.TextBoxFor(m => m.StudentId);
        @Html.LabelFor(m => m.StudentName); @Html.TextBoxFor(m => m.StudentName);
        @Html.LabelFor(m => m.Age); @Html.TextBoxFor(m => m.Age);
        <input type="submit" value="Submit Form" />
    }
</body>
<h4 style="color:purple">
    <b>ID:</b> @ViewBag.Id <br />
    <b>Name:</b> @ViewBag.Name <br />
    <b>Age:</b> @ViewBag.Age
</h4>
</html>
```

// StudentController

```
public ActionResult Form1( Student obj)
{
    ViewBag.Id = obj.StudentId;
    ViewBag.Name = obj.StudentName;
    ViewBag.Age=obj.Age;

    return View("Index");
}
```

# Standard HTML Helper methods

- **@Html.ActionLink()** - Used to create link on html page
- **@Html.TextBox()** - Used to create text box
- **@Html.CheckBox()** - Used to create check box
- **@Html.RadioButton()** - Used to create Radio Button
- **@Html.BeginForm()** - Used to start a form
- **@Html.EndForm()** - Used to end a form
- **@Html.DropDownList()** - Used to create drop down list
- **@Html.Hidden()** - Used to create hidden fields
- **@Html.Label()** - Used for creating HTML label is on the browser
- **@Html.TextArea()** - The TextArea Method renders textarea element on browser
- **@Html.Password()** - This method is responsible for creating password input field on browser
- **@Html.ListBox()** - The ListBox helper method creates html ListBox with scrollbar on browser

# Strongly-Typed HTML Helper

- `@Html.HiddenFor()`
- `@Html.LabelFor()`
- `@Html.TextBoxFor()`
- `@Html.RadioButtonFor()`
- `@Html.DropDownListFor()`
- `@Html.CheckBoxFor()`
- `@Html.TextAreaFor()`
- `@Html.PasswordFor()`
- `@Html.ListBoxFor()`

# Validations using Data Annotation Attributes

- ASP.NET MVC includes built-in attribute classes in the **System.ComponentModel.DataAnnotations** namespace.
- You can apply these attributes to the properties of the model class to display appropriate validation messages to the users.

Attribute	Usage
<b>Required</b>	Specifies that a property value is required.
<b>StringLength</b>	Specifies the minimum and maximum length of characters that are allowed in a string type property.
<b>Range</b>	Specifies the numeric range constraints for the value of a property.
<b>RegularExpression</b>	Specifies that a property value must match the specified regular expression.
<b>CustomValidation</b>	Specifies a custom validation method that is used to validate a property.

# Validations using Data Annotation Attributes

Attribute	Usage
<b>EmailAddress</b>	Validates an email address.
<b>FileExtension</b>	Validates file name extensions.
<b>MaxLength</b>	Specifies the maximum length of array or string data allowed in a property.
<b>MinLength</b>	Specifies the minimum length of array or string data allowed in a property.
<b>Phone</b>	Specifies that a property value is a well-formed phone number.

# Validations using Data Annotation Attributes

## Student.cs (inside model)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace Session15DemoMVC.Models
{
    public class Student
    {
        [Required]
        public int StudentId { get; set; }

        [StringLength(10)]
        [Required]
        public string StudentName { get; set; }

        [Range (18,45)]
        public int Age { get; set; }

        [EmailAddress]
        [Required]
        public string Email { get; set; }
    }
}
```

# Validations using Data Annotation Attributes

## Index.cshtml

```
@model Session15DemoMVC.Models.Student
@{
    ViewBag.Title = "Registration";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>

@using (Html.BeginForm("Form1", "Department", FormMethod.Post))
{
    @Html.LabelFor(m => m.StudentId);
    @Html.TextBoxFor(m => m.StudentId);
    @Html.ValidationMessageFor(m => m.StudentId, "", new { @class = "text-danger" })
    <br />

    @Html.LabelFor(m => m.StudentName);
    @Html.TextBoxFor(m => m.StudentName);
    @Html.ValidationMessageFor(m => m.StudentName, "", new { @class = "text-danger" })
    <br />
}

@Html.LabelFor(m => m.Age);
@Html.TextBoxFor(m => m.Age);
@Html.ValidationMessageFor(m => m.Age, "", new { @class = "text-danger" })

<br />

@Html.LabelFor(m => m.Email);
@Html.TextBoxFor(m => m.Email);
@Html.ValidationMessageFor(m => m.Email, "", new { @class = "text-danger" })
<br />
<input type="submit" value="Submit Form" />
}

</body>
</html>

<h4 style="color:purple">
<b>ID:</b> @ViewBag.Id <br />
<b>Name:</b> @ViewBag.Name <br />
<b>Age:</b> @ViewBag.Age<br />
<b>Email:</b> @ViewBag.Email
</h4>

</html>
```

# State Management

- State Management is the process where developers **can maintain status, user state and page information** on multiple requests for the same or different pages within the web application.
- Two types :
  - **Client-side state management** - The information is stored in the customer system while end-to-end interaction. Eg. ViewBag, TempData, Cookies, QueryStrings
  - **Server-side state management** - Store all the information in the server's memory. Eg. Session and Application

# Client-Side State Management

**View bag :** It is used to transfer the data from Controller to View

//HomeController

```
public ActionResult EmployeeDetails()
{
    Employee employee = new Employee();
    {
        employee.Empid = 101;
        employee.Ename = "Alex";
    }

    ViewBag.Message = "Employee Details";
    ViewBag.Emp= employee;
    return View();
}
```

//EmployeeDetails.cshtml

```
@{
    ViewBag.Title = "EmployeeDetails";
}

<h2>EmployeeDetails</h2>

<h3>@ViewBag.Message</h3>
<html>
<body>
    @{
        var emp = ViewBag.Emp;
        <table>
            <tr>
                <td>ID :</td>
                <td> @emp.Empid</td>
            </tr>
            <tr>
                <td>Name :</td>
                <td> @emp.Ename</td>
            </tr>
        </table>
    }
</body>
</html>
```

# Client-Side State Management

**TempData:** stores value in key/value pair. It is derived from TempDataDictionary. It is mainly used to transfer the data from one request to another request or we can say subsequent request. If the data for TempData has been read, then it will get cleaned. To persist the data, we use Keep()

# Client-Side State Management

## TempData:

```
public ActionResult Test TempData()
{
    Employee employee = new Employee();
    {
        employee.Empid = 101;
        employee.Ename = "Alex";
    }
    //Setting the TempData
    TempData["emp"] = employee;
    return RedirectToAction("Contact");
}

public ActionResult Contact()
{
    //Not reading TempData
    return View();
}

public ActionResult About()
{
    //Data will available here because we have not read data yet
    var tempEmpData = TempData["Emp"];

    return View();
}
```

//About.cshtml

```
@{
    ViewBag.Title = "About";
}
<h2>@ViewBag.Title.</h2>

<p>
    @{
        var data = (StateManagementDemo.Models.Employee)TempData["Emp"];
        TempData.Keep();

        @data.Empid;
        @data.Ename;
    }
</p>
```

# Client-Side State Management

## Cookies:

- A cookie is a **plain-text file stored by the client** (usually a browser), tied to a specific website.
- The client will then allow this specific website **to read the information stored** in this file on subsequent requests, basically allowing the server (or even the client itself) to store information for later use.

```
public ActionResult Cookietest()
{
    HttpCookie cookie = new HttpCookie("TestCookie");
    cookie.Value = "This is test cookie"; //Setting Cookie value
    this.ControllerContext.HttpContext.Response.Cookies.Add(cookie);

    if (this.ControllerContext.HttpContext.Request.Cookies.AllKeys.Contains("TestCookie"))
    {
        HttpCookie cookievar = this.ControllerContext.HttpContext.Request.Cookies["TestCookie"];
        ViewBag.CookieMessage = cookievar.Value;
    }
    return View();
}
```

# Client-Side State Management

## QueryString

- A query string is a string variable that is inserted at the end of the page URL. It can be used for sending data over several pages

```
//HomeController  
  
public ActionResult QueryStringTest()  
{  
    //Send Model object in QueryString to another Controller.  
    return RedirectToAction("Index", "Employee", new { Empid = 1, Ename = "Gary" });  
}
```

```
//EmployeeController  
public ActionResult Index()  
{  
    Employee obj = new Employee()  
    {  
        Empid = int.Parse(Request.QueryString["Empid"]),  
        Ename = Request.QueryString["Ename"]  
    };  
    return View(obj);  
}
```

## //QueryStringTest.cshtml of HomeController

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>QueryStringTest</title>
</head>
<body>
    @using (Html.BeginForm("Form1", "Home", FormMethod.Post))
    {
        <input type="submit" value="Send" />
    }
</body>
</html>
```

## //Index.cshtml of EmployeeController

```
@model StateManagementDemo.Models.Employee
 @{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <table cellpadding="0" cellspacing="0">
        <tr>
            <th colspan="2" align="center">Employee Details</th>
        </tr>
        <tr>
            <td>Employee ID:</td>
            <td>@Model.Empid</td>
        </tr>
        <tr>
            <td>Name:</td>
            <td>@Model.Ename</td>
        </tr>
    </table>
</body>
</html>
```

# Server-Side State Management

**Session :** Session state enables you to store and retrieve values for a user when the user navigates to other view in an ASP.NET MVC application

//EmployeeController

```
public ActionResult Form1(Employee obj)
{
    Session["empid"] = Convert.ToString(obj.EmpID);
    return View("Index");
}
```

//Index.cshtml

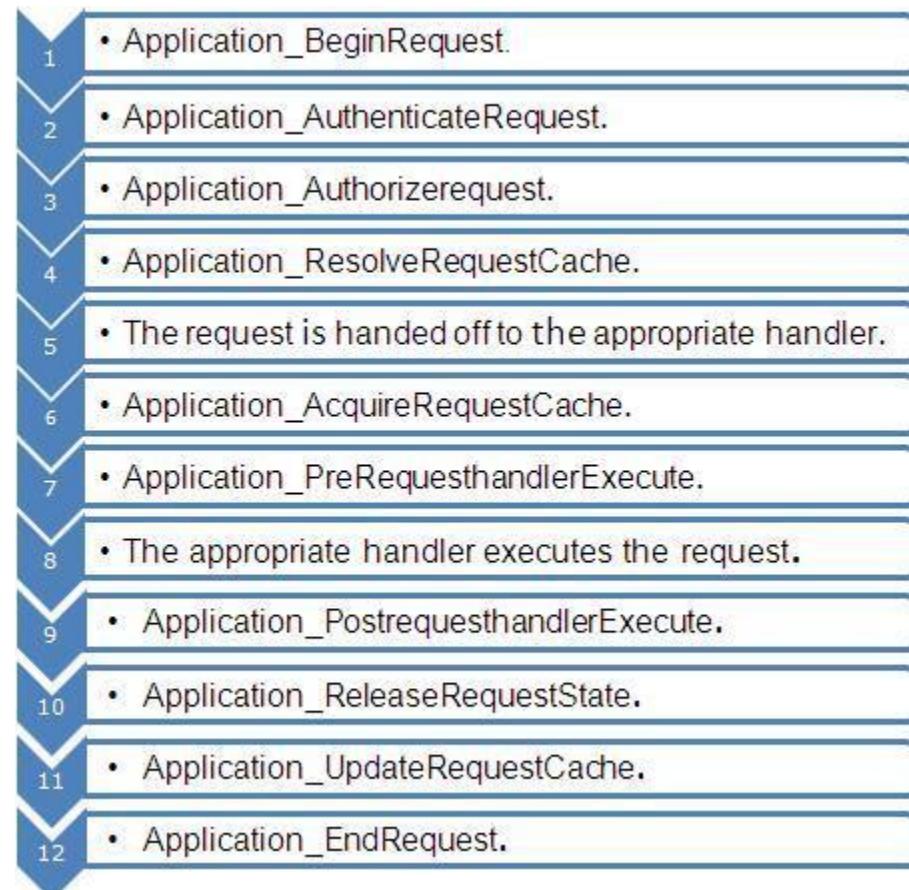
```
Session Name :
@if (Session["empid"] != null)
{
    @Html.Label(Convert.ToString(Session["empid"]))
}
```

# Server-Side State Management

- **Application State** : is stored in the **memory of the server** and is faster than storing and retrieving information in a database.
- Session state is specific for a single user session, but Application State is for all users and sessions.
- Technically the data is shared amongst users by a **HttpApplicationState** class and the data can be stored here in a key/value pair.
- It can also be accessed using the application property of the **HttpContext** class.
- **Global.asax** file is used for **handling application events** or methods. It always exists in the root level

# Server-Side State Management

Event execution by the `HTTPApplication` class for any specific requirement.



# References

1. C .Net Web Developers Guide by Syngress
2. ASP.NET 4.5, Covers C# and VB Codes, Black Book , Kogent Learning Solutions Inc.

# **Session-16**

**Data Management With  
ADO.NET, Routing and Request  
Life Cycle**

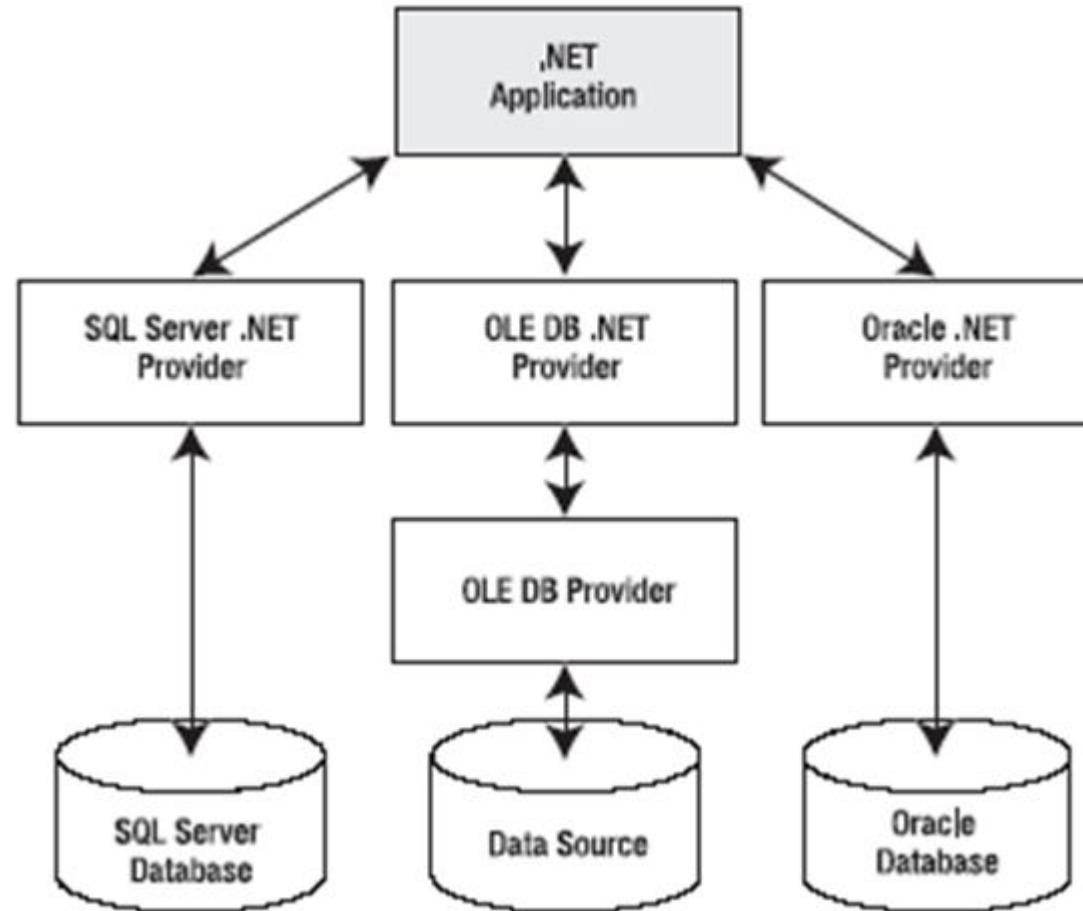
# Contents

- What is ADO.NET?
- ADO.NET Architecture
- ADO.NET Data Providers
- ADO.NET Namespaces
- ADO.NET objects
- Routing
- Configure Routes in MVC
- Multiple Routes
- Register Routes
- Request Life cycle

# What is ADO.NET?

- The .NET Framework includes its own data access technology, ADO.NET.
- ADO. NET consists of managed classes that allow .NET applications to **connect to data sources** (usually relational databases), execute commands, and manage disconnected data

# ADO.NET Architecture



# ADO.NET Data Providers

A **data provider** is a set of ADO.NET classes that allows you to :

- access a specific database
- execute SQL commands and retrieve data.

Essentially, a data provider is a bridge between your application and a data source.

# ADO.NET Namespaces

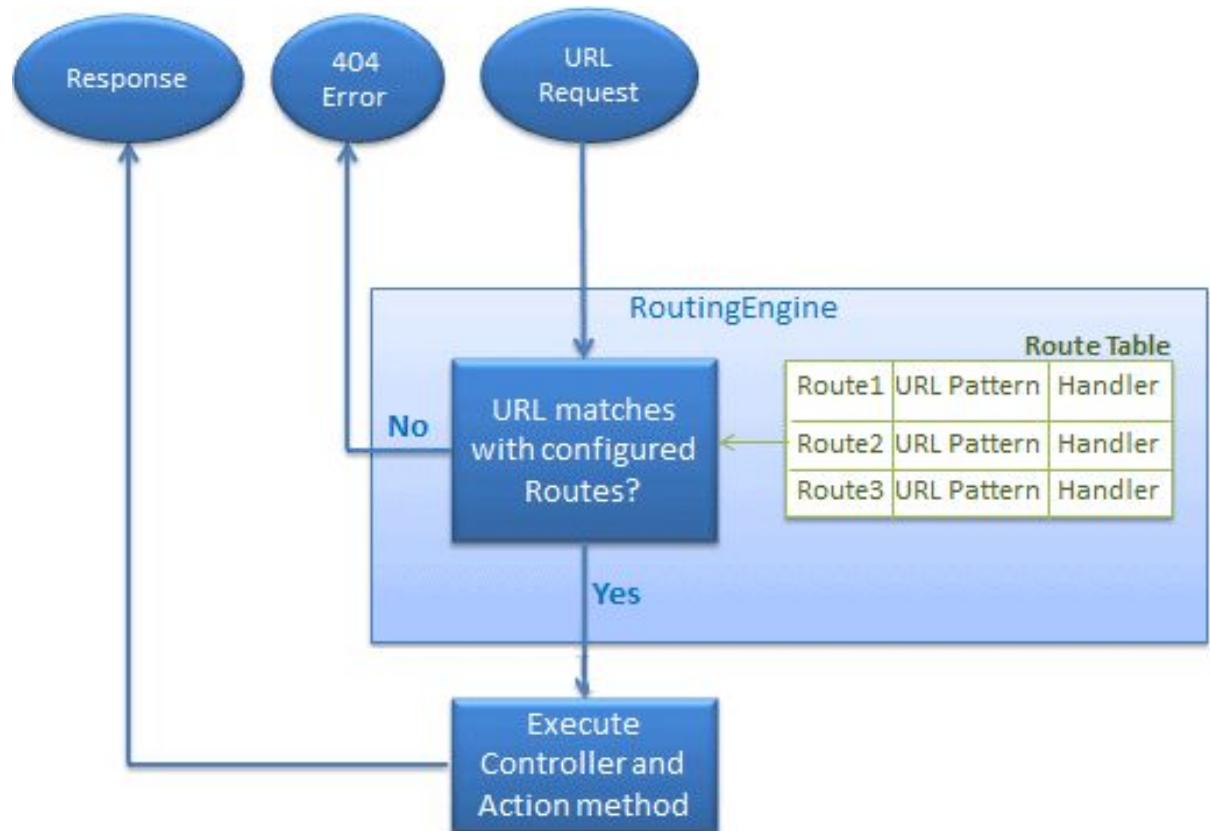
Namespace	Description
System.Data	Contains the key data container classes that model columns, relations, tables, datasets, rows, views, and constraints. In addition, contains the key interfaces that are implemented by the connection-based data objects.
System.Data.Common	Contains base, mostly abstract classes that implement some of the interfaces from System.Data and define the core ADO.NET functionality. Data providers inherit from these classes to create their own specialized versions.
System.Data.OleDb	Contains the classes used to connect to an OLE DB provider, including OleDbCommand, OleDbConnection, and OleDbDataAdapter. These classes support most OLE DB providers but not those that require OLE DB version 2.5 interfaces.
System.Data.SqlClient	Contains the classes you use to connect to a Microsoft SQL Server database, including SqlCommand, SqlConnection, and SqlDataAdapter. These classes are optimized to use the TDS interface to SQL Server.
System.Data.OracleClient	Contains the classes required to connect to an Oracle database (version 8.1.7 or later), including OracleCommand, OracleConnection, and OracleDataAdapter. These classes are using the optimized Oracle Call Interface (OCI).
System.Data.Odbc	Contains the classes required to connect to most ODBC drivers. These classes include OdbcCommand, OdbcConnection, and OdbcDataAdapter. ODBC drivers are included for all kinds of data sources and are configured through the Data Sources icon in the Control Panel.
System.Data.SqlTypes	Contains structures that match the native data types in SQL Server. These classes aren't required but provide an alternative to using standard .NET data types, which require automatic conversion.

# ADO.NET Objects

- **Connection:** You use this object to establish a connection to a data source.
- **Command:** You use this object to execute SQL commands and stored procedures.
- **DataReader:** This object provides fast read-only, forward-only access to the data retrieved from a query.
- **DataAdapter:** This object performs two tasks. First, you can use it to fill a **DataSet** (a disconnected collection of tables and relationships) with information extracted from a data source. Second, you can use it to apply **changes** to a data source, according to the modifications you've made in a **DataSet**.

# Routing

- ASP.NET Routing module is responsible for **mapping incoming browser requests** to particular MVC controller actions
- When you create a new ASP.NET MVC application, the **application is already configured** to use ASP.NET Routing
- All the configured routes of an application stored in **RouteTable** and will be used by the Routing engine to determine appropriate **handler class** or file for an incoming request.



# Configure Routes in MVC

- Every MVC application must configure (register) at least one route configured by the MVC framework by default
- the route is configured using the **MapRoute()**

The screenshot shows the Visual Studio IDE. On the left is the code editor with the file `RouteConfig.cs` open. The code defines a `RouteConfig` class with a static method `RegisterRoutes` that maps routes. Annotations with arrows point to specific parts of the code:

- An arrow points to the line `routes.IgnoreRoute("{resource}.axd/{*pathInfo}");` with the label "Route to ignore".
- An arrow points to the `name: "Default",` part of the `MapRoute` call with the label "Route name".
- An arrow points to the `url: "{controller}/{action}/{id}",` part of the `MapRoute` call with the label "URL Pattern".
- An arrow points to the `defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }` part of the `MapRoute` call with the label "Defaults for Route".

On the right, the Solution Explorer shows the project structure for 'MVC-BasicTutorials'. The `RouteConfig.cs` file is highlighted in blue, indicating it is selected. Other files like `BundleConfig.cs`, `FilterConfig.cs`, and `HomeController.cs` are also visible.

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}"); // Route to ignore

        routes.MapRoute( // Route name
            name: "Default", // URL Pattern
            url: "{controller}/{action}/{id}", // Defaults for Route
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

RouteConfig.cs under App\_Start folder

# URL Pattern

- The URL pattern is considered only after the domain name part in the URL.
- For example, the URL pattern "`{controller}/{action}/{id}`" would look like : **localhost:1234/{controller}/{action}/{id}**.



# Multiple Routes

```
namespace MVCProject
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

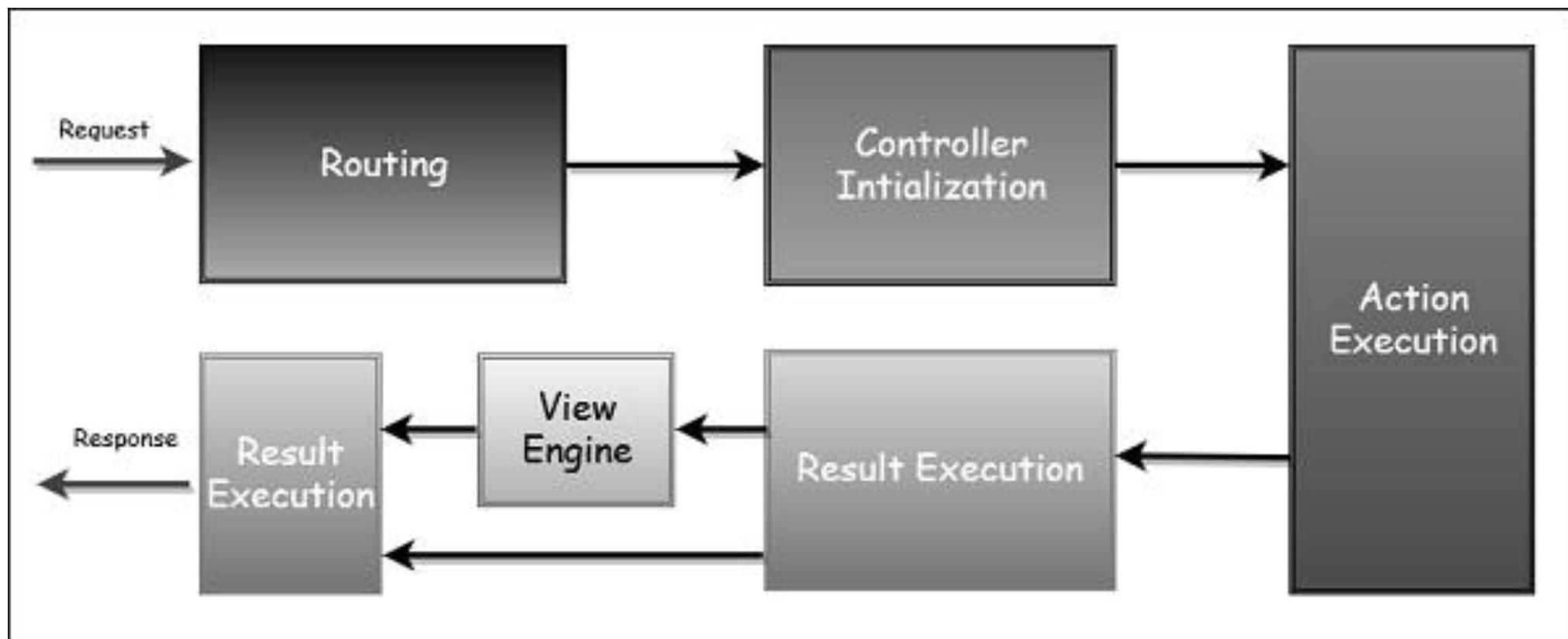
            routes.MapRoute(
                name: "Student",
                url: "students/{id}",
                defaults: new { controller = "Student", action = "Index" }
            );

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}
```

the URL pattern for the Student route is `students/{id}`, which specifies that any URL that starts with `domainName/students`, must be handled by the `StudentController`.

Notice that we haven't specified `{action}` in the URL pattern because we want every URL that starts with `students` should always use the `Index()` action of the `StudentController` class.

# Request Life Cycle



# Register Routes

All the routes are registered in Application\_Start() in **Global.asax**.

```
namespace MVCProject
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

# References

1. C .Net Web Developers Guide by Syngress
2. ASP.NET 4.5, Covers C# and VB Codes, Black Book , Kogent Learning Solutions Inc.

# **Session-17**

**Layout, Bundle, Minification and  
MVC Security**

# Contents

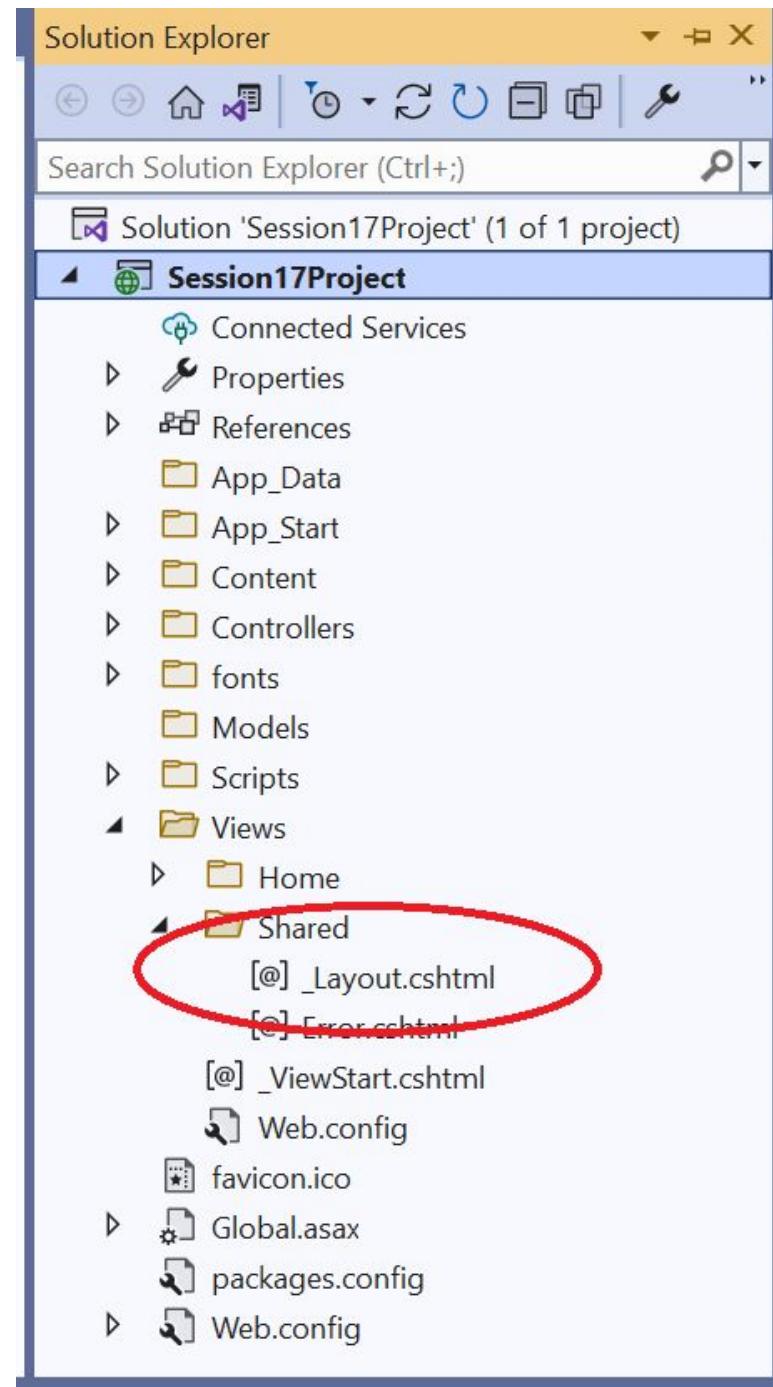
- Layout View
- Understanding Bundles and Bundle Types
- Minification
- Error handling in MVC with log entry
- MVC Security
- Cross Site Request Forgery Attacks
- Anti-forgery token
- Introduction to OAuth

# Layout View

- An application may contain a specific UI portion that **remains the same throughout the application**, such as header, left navigation bar, right bar, or footer section.
- ASP.NET MVC introduced a Layout view which contains these common UI portions so that we don't have to write the same code in every page
- The layout view **eliminates duplicate coding** and **enhances development speed** and **easy maintenance**

# Layout View

- The layout view has the **same extension as other views**, .cshtml or .vbhtml.
- Layout views are shared with multiple views, so it must be stored in the **Shared folder**



# Layout View

Method	Description
RenderBody()	Renders the portion of the child view that is not within a named section. Layout view must include the RenderBody() method.
RenderSection(string name)	Renders a content of named section and specifies whether the section is required.

[Home](#)[About](#)[Contact](#)

# ASP.NET

## @RenderBody

ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.

[Learn more »](#)

## Getting started

ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and gives you full control over markup for enjoyable, agile development.

[Learn more »](#)

## Get more libraries

NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visual Studio projects.

[Learn more »](#)

## Web Hosting

You can easily find a web hosting company that offers the right mix of features and price for your applications.

[Learn more »](#)

## @RenderSection

Footer Section

# Layout View

## @RenderSection :

- **false** parameter denotes that particular section is optional and It is your choice whether to use it or not.
- If you use **true** parameter then it is compulsory to use this section in child page otherwise you will get error.

```
@RenderSection("subheader", required:false)
```

# Layout View

## Layout View (\_myLayout.cshtml)

```
<div class="container-fluid">
    <p>
        @RenderBody()
    </p>
    <div class="border-end bg-primary" style="color: white">
        @RenderSection("subheader", required:false)
    </div>
</div>
```

## Child View (About.cshtml)

```
@{
    ViewBag.Title = "About";
}
<h2>@ViewBag.Title.</h2>
<h3>@ViewBag.Message</h3>

<p>Use this area to provide additional information.</p>

@section subheader {
    About Us Footer
}
```

# Bundling

- Bundling and minification techniques were introduced in MVC 4 to **improve request load time**.
- Bundling **allows us to load the bunch of static files from the server in a single HTTP request**.

Loading Scripts in Separate Request



Loading Bundle



# Bundle Types

- **ScriptBundle:** ScriptBundle is responsible for JavaScript minification of single or multiple script files.

```
bundles.Add(new ScriptBundle("~/bundles/myjsfiles").Include(  
    "~/Scripts/scripts.js")); //In BundleConfig.cs  
  
@Scripts.Render("~/bundles/myjsfiles") //In .cshtml file
```

- **StyleBundle:** StyleBundle is responsible for CSS minification of single or multiple style sheet files.

```
bundles.Add(new StyleBundle("~/Content/mycss").Include(  
    "~/Content/styles.css")); //In BundleConfig.cs  
  
@Styles.Render("~/Content/mycss") //In .cshtml file
```

# Minification

Minification technique optimizes script or CSS file size by removing unnecessary white space and comments and shortening variable names to one character.

Example: JavaScript

```
details = function(name)
{
  //this is comment
  var msg= "Hello"+ name;
  alert(msg);
}
```

Minification

```
sayHello=function(n){var t="Hello"+n;alert(t)}
```

# Error handling in MVC with log entry

- HandleError is the default built-in exception filter in ASP.NET MVC
- HandleError Attribute filter works only when custom errors are enabled in the **Web.config** file of your application.

```
<system.web>
    <customErrors mode="On">
        <error statusCode="404" redirect="~/Error/NotFound"/>
    </customErrors>
</system.web>
```

Controller

Action

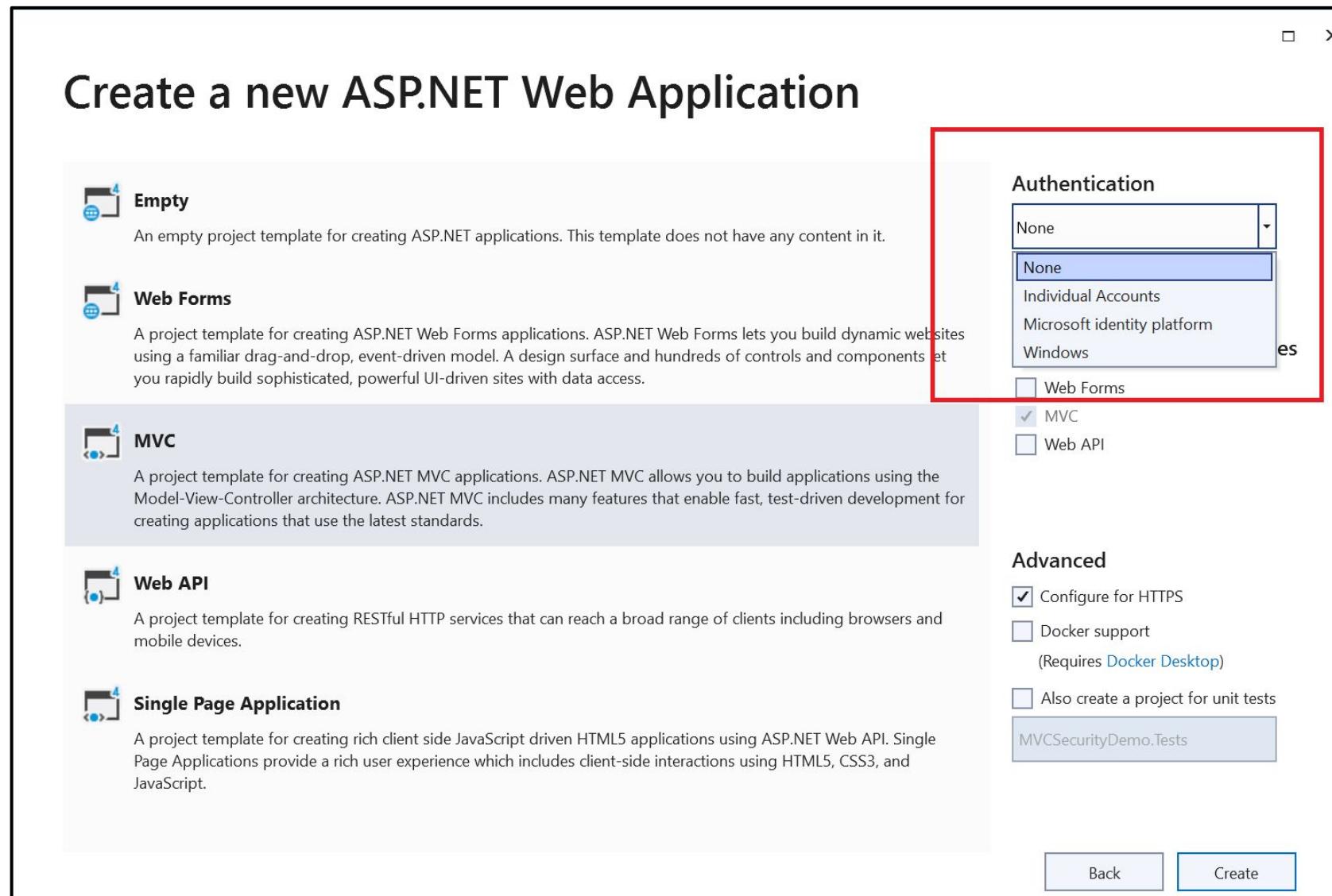
# MVC Security

- MVC Security **implements security features** in the application. It also supports new membership features included with ASP.NET and available for use from ASP.NET MVC.
- Also it has the **new identity components** that is a part of ASP.NET and see how to **customize membership** for our **users** and **roles**.

# MVC Security

**Authentication** of user means **verifying the identity of the user**. This is really important.

You might need to present your application only to the authenticated users for obvious reasons.



# MVC Security

## Authentication options

1. **None** : The first option is No Authentication/None and this option is used when you want to build a website that doesn't care who the visitors are.
2. **Individual User Accounts** : This is the traditional forms-based authentication where users can visit a website. They can register, create a login, and by default their username is stored in a SQL Server database using some new ASP.NET identity features
3. **Microsoft Identity platform**: It's an open-source libraries, and application management tools
4. **Windows** :which works well for intranet applications. A user logs into Windows desktop and can launch a browser to the application that sits inside the same firewall

# Cross Site Request Forgery Attacks

- CSRF attacks are possible against web sites that use cookies for authentication, because browsers send all relevant cookies to the destination web site.
- Example of a CSRF attack:
  1. A user logs into [www.somewebsite.com](http://www.somewebsite.com) using form authentication
  2. The server authenticates the user. The response from the server includes an authentication cookie.
  3. Without logging out, the user visits a malicious web site. This malicious site contains the given HTML form.
  4. The user clicks the submit button. The browser includes the authentication cookie with the request.
  5. The request runs on the server with the user's authentication context, and can do anything that an authenticated user is allowed to do.

Malicious websites HTML form

```
<h1>Congratulations!! </h1>
<form action="http://somewebsite.com/api/account"
method="post">

<input type="hidden" name="Transaction"
value="withdraw" />

<input type="hidden" name="Amount" value="2000000" />

<input type="submit" value="Click Me"/>

</form>
```

Notice that the form action posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.

# Cross Site Request Forgery Attacks

- As soon as user clicks the form button, the malicious page could just as easily run a script that submits the form automatically.
- Even SSL does not prevent a CSRF attack, because the malicious site can send an "https://" request.

# Anti-forgery token

To help prevent CSRF attacks, ASP.NET MVC uses anti-forgery tokens, also called ***request verification tokens***.

1. The client requests an HTML page that contains a form.
2. The server includes two tokens in the response. One token is sent as a cookie. The other is placed in a hidden form field. The tokens are generated randomly so that an adversary cannot guess the values.
3. When the client submits the form, it must send both tokens back to the server. The client sends the cookie token as a cookie, and it sends the form token inside the form data.
4. If a request does not include both tokens, the server disallows the request.

# Anti-forgery token

Example of an HTML form with hidden form token:

```
<form action="/Home/Test" method="post">

<input name="__RequestVerificationToken"
type="hidden"
value="6fGBtLZmVBZ59oUad1Fr33BuPxANKY9
q3Srr5y[...]" />

<input type="submit" value="Submit" />

</form>
```

To add the anti-forgery tokens to a Razor page, use the **HtmlHelper.AntiForgeryToken** helper method:

```
@using (Html.BeginForm("Form2", "Employee"))
{
    @Html.AntiForgeryToken()
}
```

# OAuth

- OAuth stands for **Open Authorization**
- OAuth is an open standard for authorization.
- OAuth provides client applications a "secure delegated access" to server resources on behalf of a resource owner.
- It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials.
- It is an open standard for token-based authentication and authorization.

# References

1. C .Net Web Developers Guide by Syngress
2. ASP.NET 4.5, Covers C# and VB Codes, Black Book , Kogent Learning Solutions Inc.
3. <https://docs.microsoft.com/>

# **Session-18**

**Entity Framework and ASP.NET  
MVC Core**

# Contents

- Introduction to Entity Framework
- Different Approaches
  - Code First
  - Database First
  - Model First
- Data Annotations Attributes in EF 6
- Introduction to ASP.NET Core
- Difference ASP.NET MVC and MVC Core

# Introduction to Entity Framework

- Entity Framework is an Object Relational Mapper (ORM) which is a type of tool that simplifies **mapping between objects** in your software to the **tables** and **columns** of a relational database.
- It is an **open source ORM framework** for ADO.NET which is a part of .NET Framework.
- An **ORM takes care of creating database connections** and **executing commands**, as well as **taking query results** and automatically materializing those results as your application objects.

# Different Approaches

The Entity Framework provides three approaches to create an entity model and each one has their own pros and cons.

- Code First
- Database First
- Model First

# Different Approaches

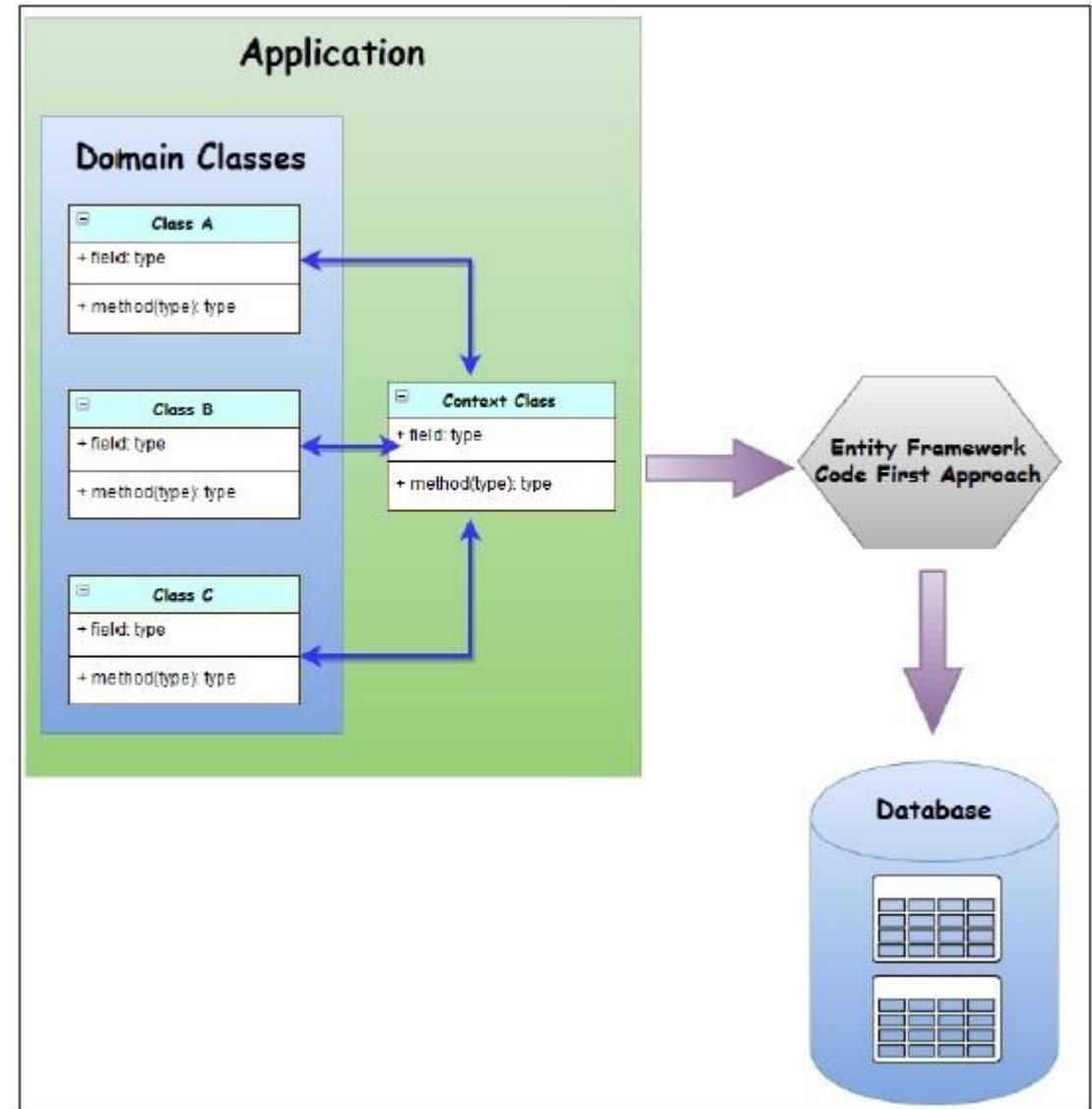
## Code First:

- Code First modeling workflow **targets a database that doesn't exist** and Code First will create it.
- It can also be used if you **have an empty database** and then **Code First will add new tables** to it.
- Code First allows you to **define your model** using C# or VB.Net classes.

# Different Approaches

## Code First:

- The domain classes have nothing to do with Entity Framework. They're just the items of your business domain.
- Entity Framework, then, has a context that manages the interaction between those classes and your database.



# Different Approaches

## Database First:

- It creates model codes (**classes, properties, DbContext etc.**) from the database in the project and **those classes** become the **link between the database and controller**.
- creates the entity framework **from an existing database**.

# Different Approaches

## Model First:

- Model First is good when you are starting a new project where the database doesn't even exist yet.
- The model is stored in an EDMX file and can be viewed and edited in the Entity Framework Designer.
- In Model First, you define your model in an Entity Framework designer then generate SQL, which will create database schema to match your model and then you execute the SQL to create the schema in your database.
- The classes that you interact with in your application are automatically generated from the EDMX file.
-

# Data Annotations Attributes in EF 6

Namespace to include : **System.ComponentModel.DataAnnotations**

Attribute	Description
<a href="#"><u>Key</u></a>	Can be applied to a property to specify a key property in an entity and make the corresponding column a PrimaryKey column in the database.
<a href="#"><u>Required</u></a>	Can be applied to a property to specify that the corresponding column is a NotNull column in the database.
<a href="#"><u>MinLength</u></a>	Can be applied to a property to specify the minimum string length allowed in the corresponding column in the database.
<a href="#"><u>MaxLength</u></a>	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database.
<a href="#"><u>StringLength</u></a>	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database.

# Data Annotations Attributes in EF 6

Namespace to include :

**System.ComponentModel.DataAnnotations.Schema Attributes**

Attribute	Description
<a href="#"><u>Table</u></a>	Can be applied to an entity class to configure the corresponding table name and schema in the database.
<a href="#"><u>Column</u></a>	Can be applied to a property to configure the corresponding column name, order and data type in the database.
<a href="#"><u>Index</u></a>	Can be applied to a property to configure that the corresponding column should have an Index in the database. (EF 6.1 onwards only)
<a href="#"><u>ForeignKey</u></a>	Can be applied to a property to mark it as a foreign key property.
<a href="#"><u>NotMapped</u></a>	Can be applied to a property or entity class which should be excluded from the model and should not generate a corresponding column or table in the database.

# Some Examples of Data Annotations in EF

## 1. Table

```
using System.ComponentModel.DataAnnotations.Schema;

[Table("StudentMaster")]
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}
```

## 2. Key

```
using System.ComponentModel.DataAnnotations;

public class Student {

    [Key]
    public int StudentKey { get; set; }
    public string StudentName { get; set; }
}
```

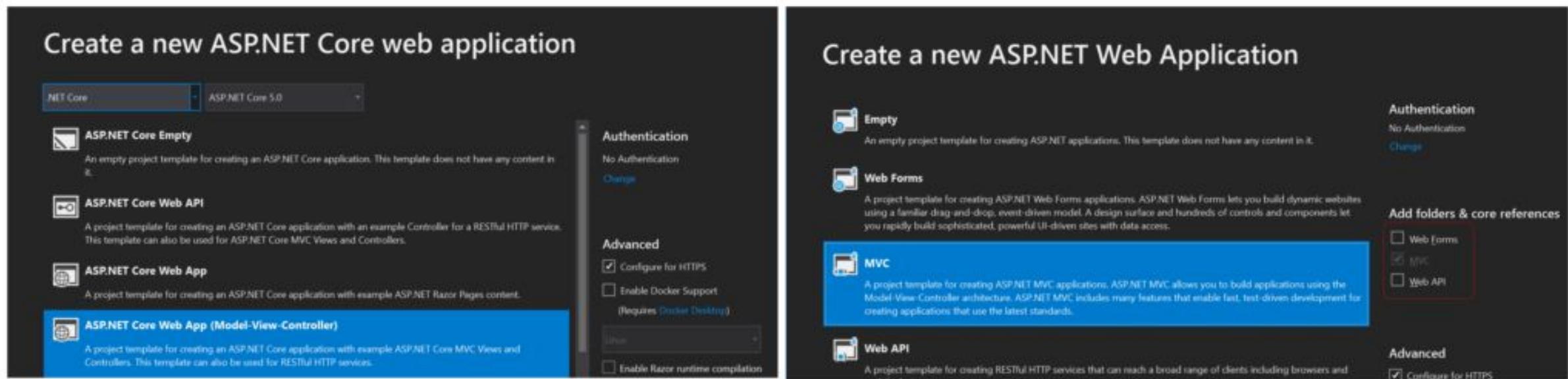
# Introduction to ASP.NET Core

ASP.NET Core is a **cross-platform, high-performance, open-source framework** for building modern, cloud-enabled, Internet-connected apps. With ASP.NET Core, you can:

- Build web apps and services, **Internet of Things (IoT)** apps, and mobile backends.
- Use your favorite development tools on **Windows, macOS, and Linux**.
- Deploy to the cloud or on-premises.
- Run on **.NET Core**.

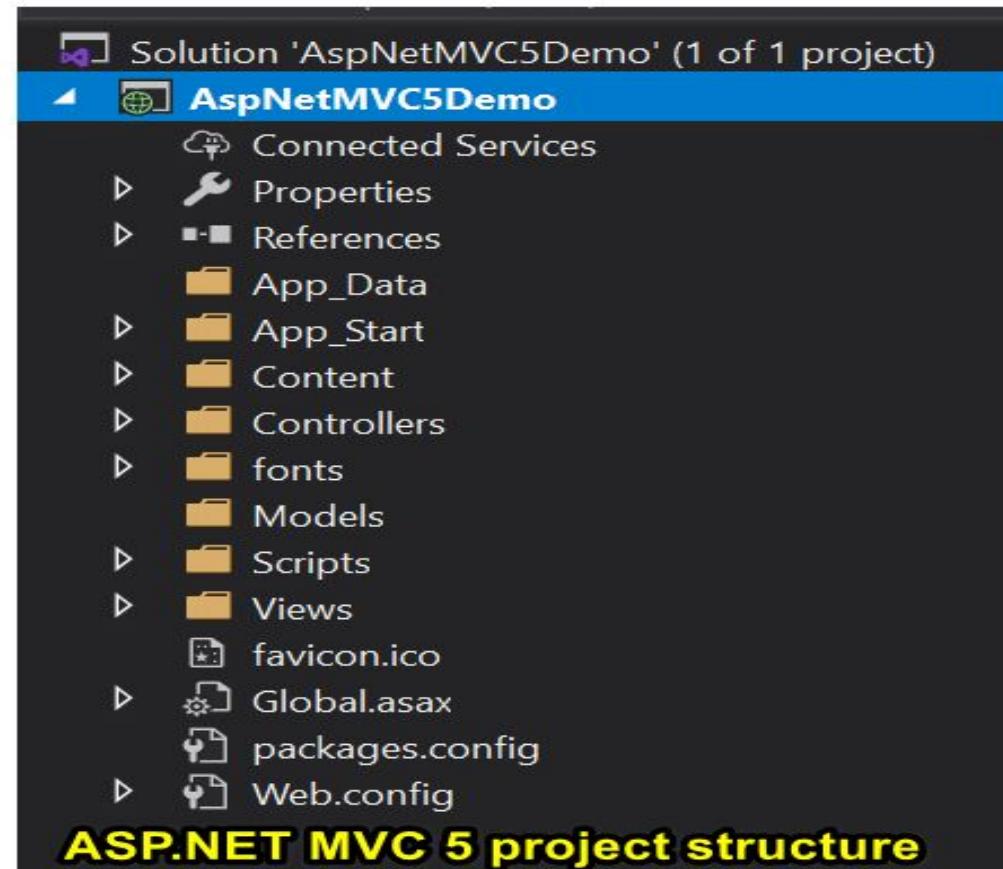
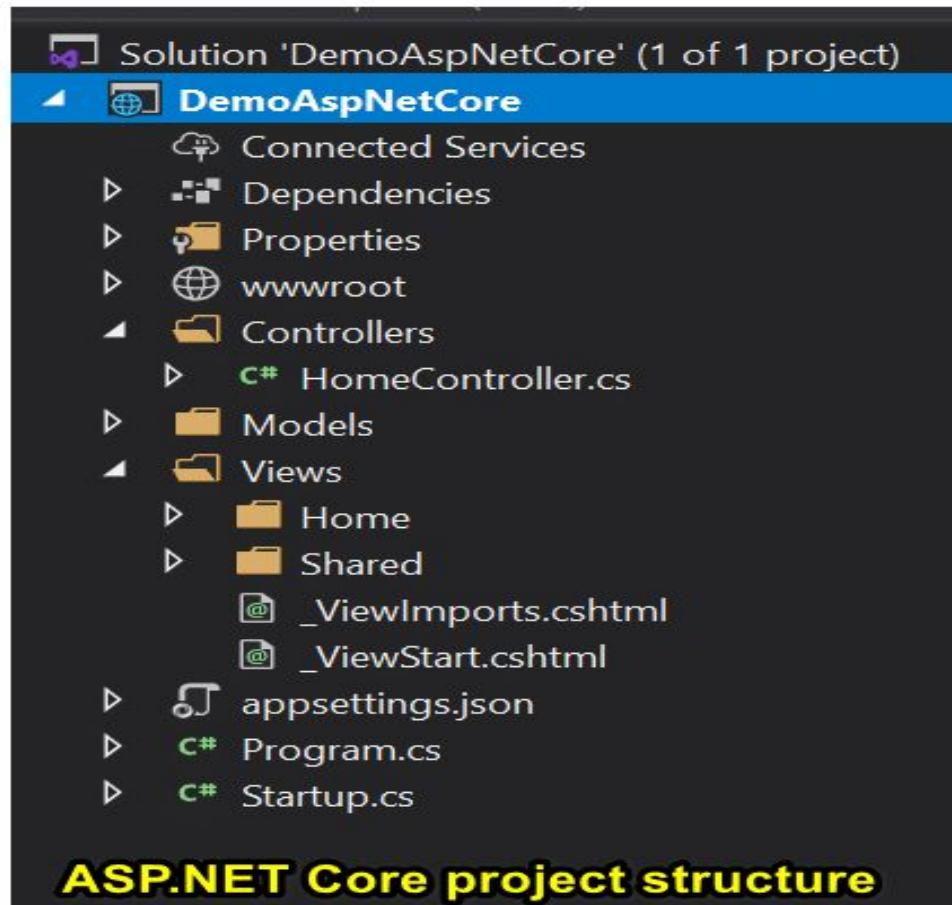
# Difference ASP.NET MVC and MVC Core

Difference 1 – Single aligned web stack for ASP.NET Core MVC and Web APIs



# Difference ASP.NET MVC and MVC Core

Difference 2 – Project (Solution) Structure Changes : ASP.NET Core 5 MVC solution explorer on the right-hand side, there is no Web.config, Global.asax

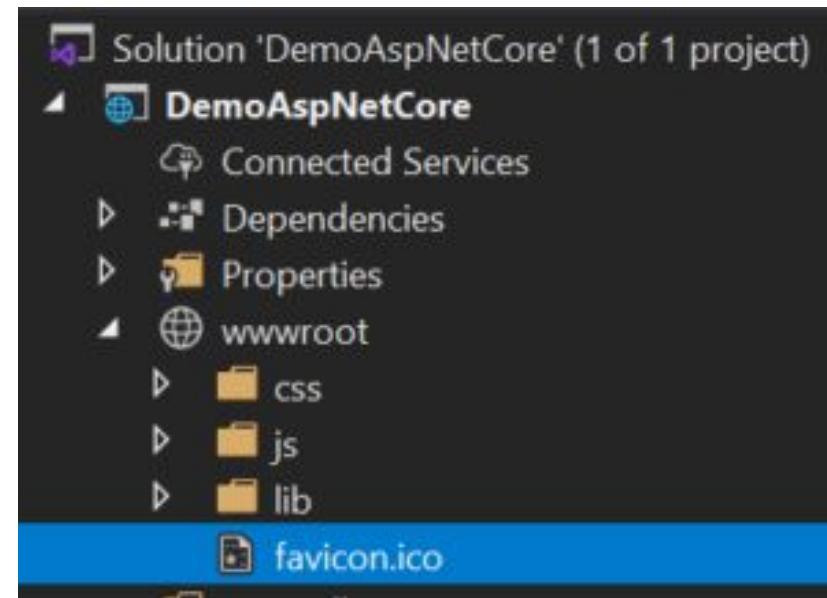


# Difference ASP.NET MVC and MVC Core

## Difference 3 – ASP.NET Core apps don't need IIS for hosting

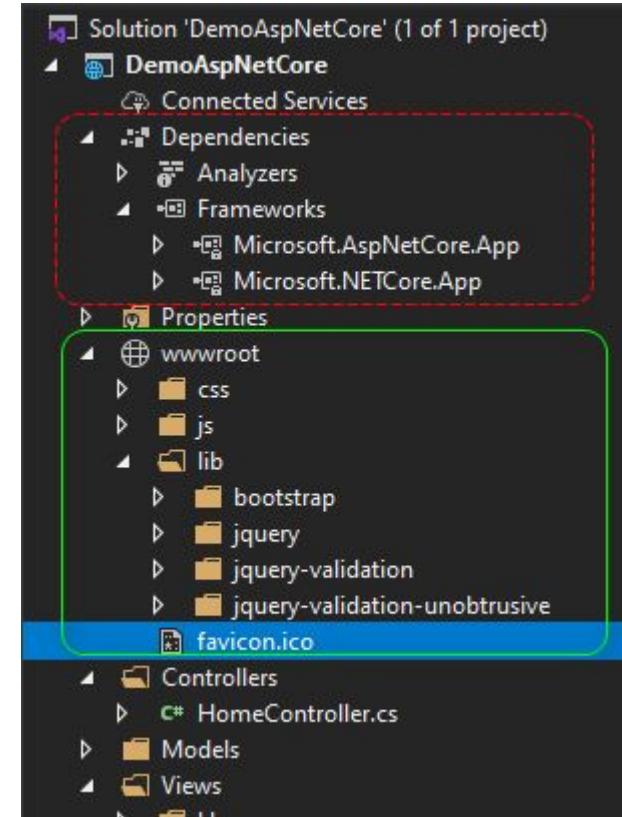
The goal of ASP.NET Core is to be cross-platform using .NET Core (.NET 5). With this in mind, Microsoft decided to host ASP.NET Core applications on IIS and be self-hosted or use the Nginx web server on Linux.

## Difference 4 – wwwroot folder for static files



# Difference ASP.NET MVC and MVC Core

**Difference 5 – New approach to Server-side and client-side dependency management of packages.** Leverage the experience of working in Visual Studio IDE and deploy ASP.NET Core applications either on Windows, Linux, or Mac using .NET Core, its Server side management of dependencies. Client-side dependency management is more important because the client-side has more different packages from the server-side. Client-side will indeed have jQuery, Bootstrap, grunt, and any Javascript frameworks like AngularJS, Backbone, images, and style files.



Server Side (red) and Client-Side (green) Dependency Management

# Difference ASP.NET MVC and MVC Core

## Difference 6 – Inbuilt Dependency Injection (DI) support for ASP.NET Core 5

**Dependency Injection (DI)** achieves loosely coupled, more testable code; it's vital because it helps write unit tests.

**Note :** The Dependency Injection Design Pattern in C# is a process in which we are **injecting the object of a class into a class that depends on that object**. Dependency Injection pattern involves 3 types of classes:

- **Client Class:** The Client class (dependent class) is a class that depends on the service class.
- **Service Class:** The Service class (dependency) is a class that provides service to the client class.
- **Injector Class:** The Injector class injects the service class object into the client class.

# References

1. C .Net Web Developers Guide by Syngress
2. ASP.NET 4.5, Covers C# and VB Codes, Black Book , Kogent Learning Solutions Inc.
3. <https://docs.microsoft.com/>

# **Session-19**

**Localization in MVC  
and  
Deploying MVC Application**

# Contents

- Localization
- ISO Codes for Cultures
- CultureInfo class
- Server Side Culture Declaration
- Setting the culture to the client's culture
- Deploying MVC Application on IIS

# Localization

- **Internationalization** is the process of enabling your code to run correctly all over the world. It has two parts: **globalization** and **localization**.
- **Globalization** is about **writing your code to accommodate multiple languages** and **region** combinations. The combination of a language and a region is known as a **culture**.
- **Localization** the **process of customization** to make **our application** behave depending on the **current culture** and **locale**
- The namespace required : **System.Globalization**

# ISO Codes for Culture Combinations

- There are **International Organization for Standardization (ISO)** codes for all culture combinations.
- For example,
  1. in the code **da-DK**, **da** indicates the Danish language and **DK** indicates the Denmark region,
  2. in the code **fr-CA**, **fr** indicates the French language and **CA** indicates the Canadian region.

# CultureInfo Class

- In the .NET Framework , the CultureInfo class from System.Globalization namespace **provides culture-specific information** such as the **associated language, country or region, calendar, and cultural conventions.**
- The **CurrentCulture** property represents the culture that the current thread uses.
- The **CurrentUICulture** property represents the current culture that Resource Manager uses to look up culture-specific resources at run time.

# Server Side Culture Declaration

- ASP.NET enables you to define the culture used by your entire ASP.NET application or by a specific page within your application. You can specify the culture for any of your ASP.NET applications by means of the appropriate configuration files.
- Defining the `<globalization>` section in the **web.config** file

```
<configuration>
  <system.web>
    <b><globalization culture="ru-RU" uiCulture="ru-RU" /></b>
  </system.web>
</configuration>
```

Note : Setting the culture at either the **server-wide** or the **application-wide level**

# Server Side Culture Declaration

- Defining the culture at the page level using the @Page directive

```
<%@ Page Language="C#" UICulture="ru-RU" Culture="ru-RU" %>
```

This example determines that the Russian language and culture settings are used for everything **on the page**.

# Setting the culture to the client's culture

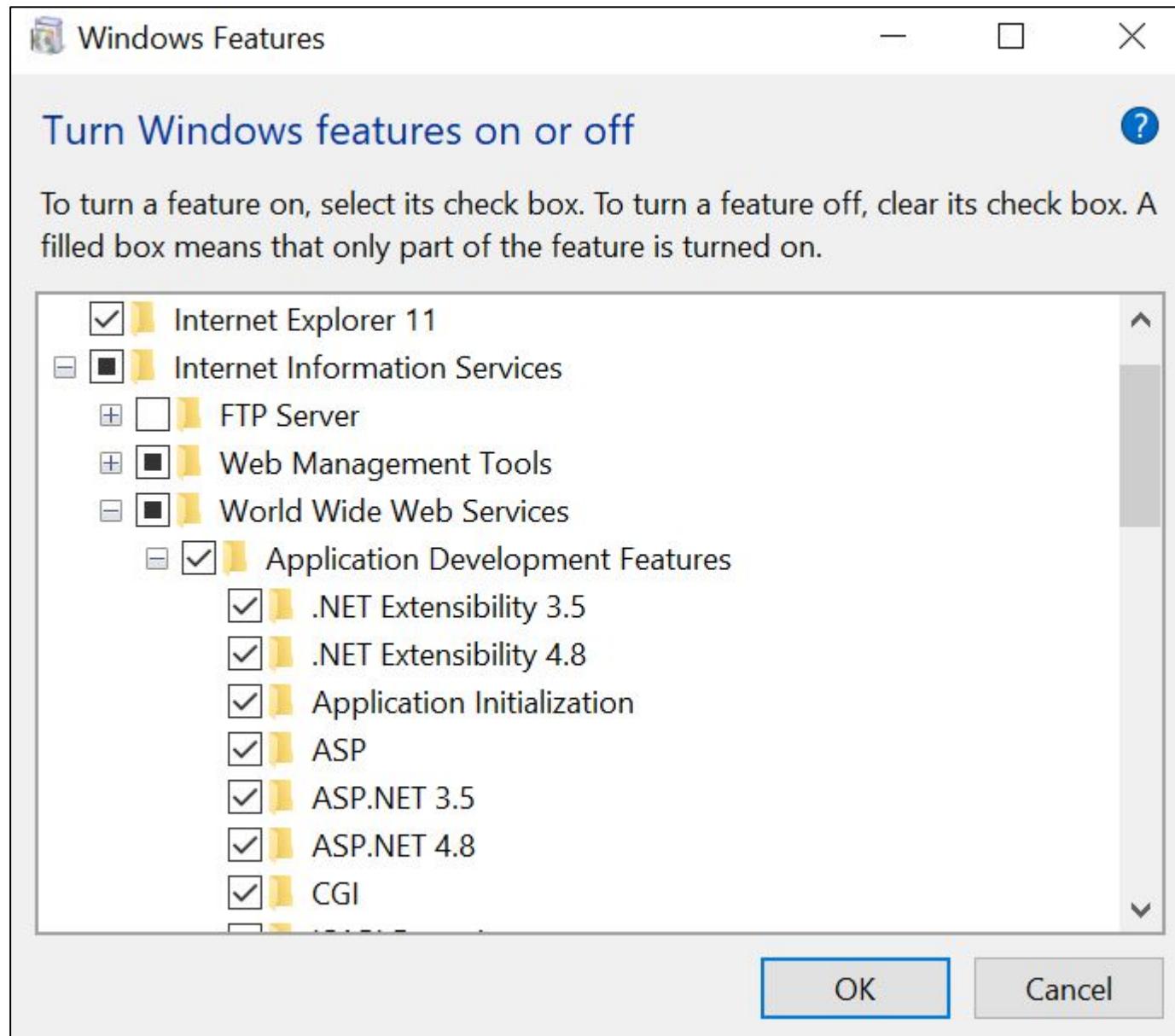
When working with culture on the client side, the culture that is used is the culture that is defined on the server. Therefore, if you have the culture set to **en-US** on your server, this is the default that is utilized on the client-side operations as well.

```
int value=3000;  
//Spanish(Mexico)  
Thread.CurrentThread.CurrentCulture = new System.Globalization.CultureInfo("es-MX");  
Console.WriteLine(value.ToString("c"));  
  
//English(UK)  
Thread.CurrentThread.CurrentCulture = new System.Globalization.CultureInfo("en-GB");  
Console.WriteLine(value.ToString("c"));
```

# Deploying MVC Application

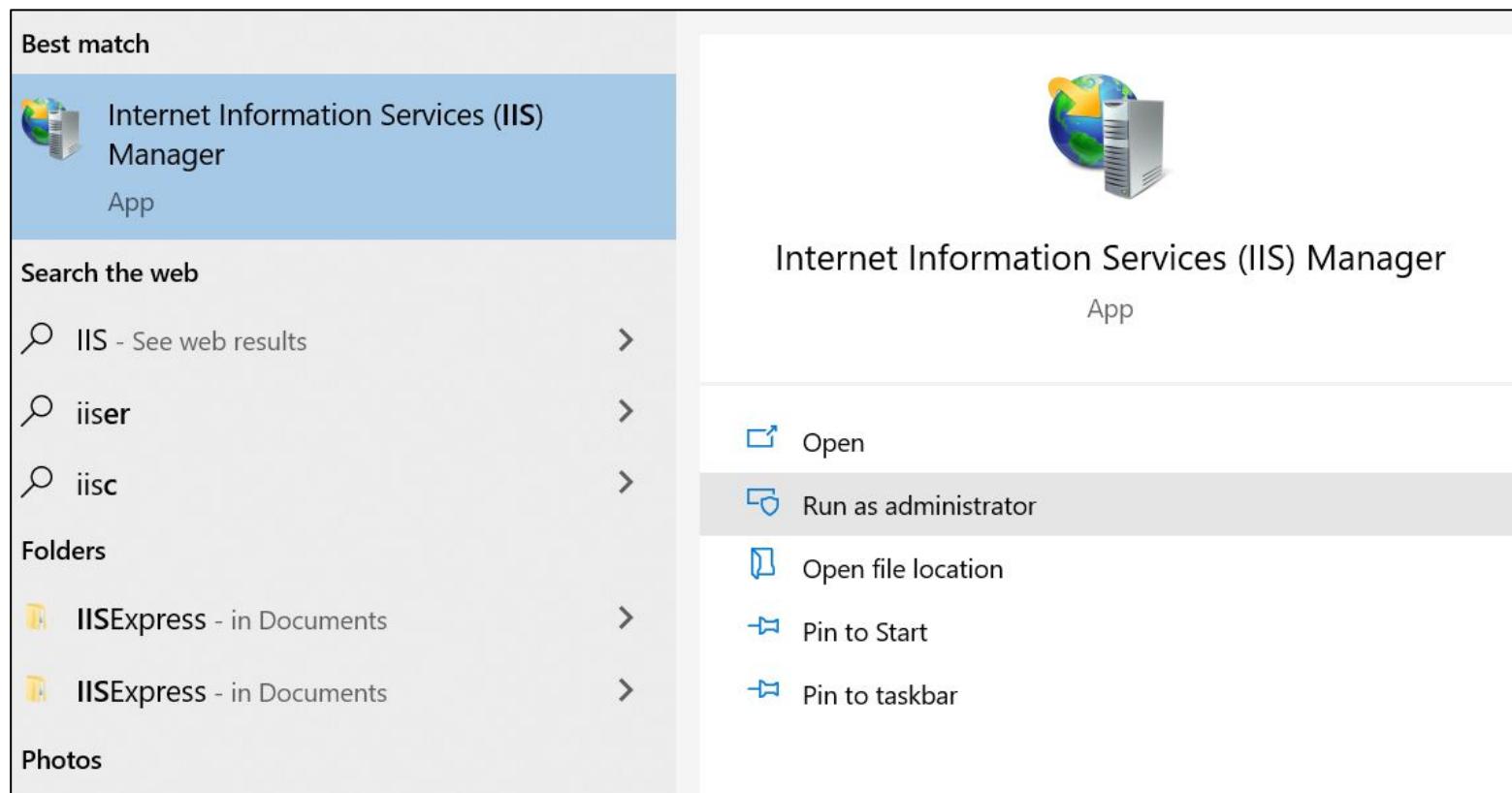
## Enabling IIS and required IIS components on Windows 10

1. Open Control Panel and click Programs and Features > Turn Windows features on or off.
2. Enable Internet Information Services.
3. Expand the Internet Information Services feature and verify that the web server components listed in the next section are enabled.
4. Click OK.
5. Check from start menu for “IIS Manager” in Windows



# Deploying MVC Application

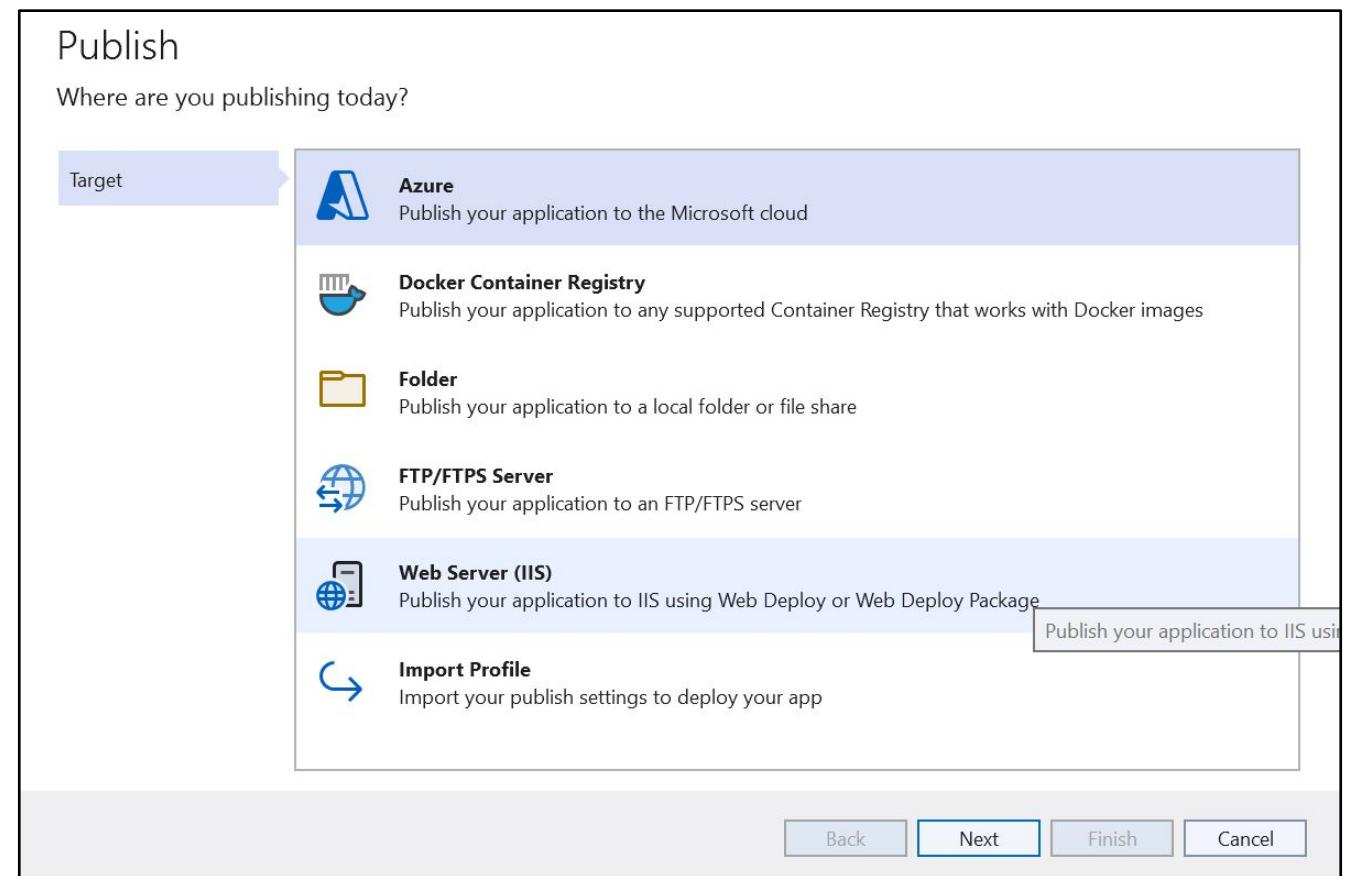
## 6. Start IIS Manager with Administrator



# Deploying MVC Application

## Publish the MVC application

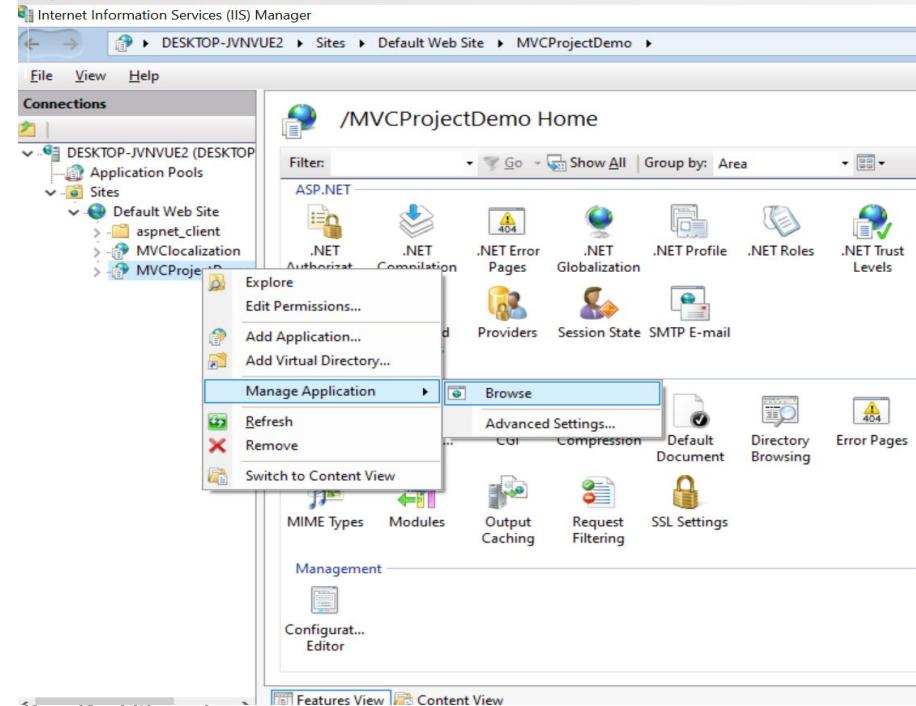
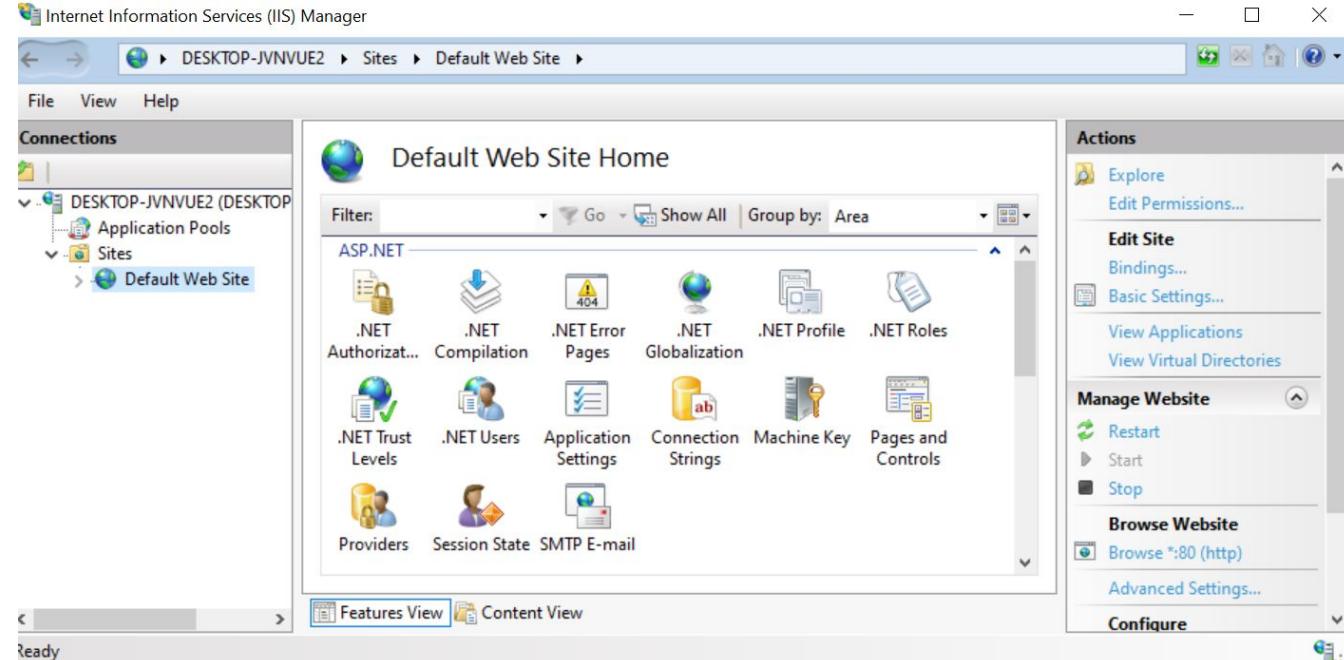
1. Open the Visual Studio with Administrator
2. In the Solution explorer, Right click the Project and Select “Publish”
3. Select Webserver (IIS) and Click next
4. Then Select “Web Deploy Package” and click Next
5. Then provide the location of your Project and Mention site name.
6. Then Publish



# Deploying MVC Application

## Creating Virtual Directory

1. Open the IIS Manager with Administrator
2. Right Click on the Default Web Site and Create Virtual Directory
3. Give the Alias and path of the Application folder
4. Click OK
5. Test the website execution by right click on virtual directory > Manage Application and Browse



# References

1. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development
2. <https://vmsdurano.com/deploying-your-asp-net-mvc-5-app-to-iis8/>

# **Session-20**

**Windows Communication  
Foundation(WCF)**

# Contents

- Services
- SOA
- WCF overview
- Contracts
  - Service Contracts
  - Data Contracts
  - Message Contracts
- Programming Model of WCF
- Binding Types
- Adding WCF in VS2022
- WCF Service Creation

# Services

- **Services** are the **logical encapsulation** of self-contained **business functionalities**
- Services are **not classes** or **namespaces** or **objects**. It's just self-contained business functionality.

Example : The petrol filling station is providing three *services*( ***fuel filling up***, ***air pressure checking***, ***checking puncture***) that are logically encapsulated within one filling station and they are self contained

# Service Oriented Architecture (SOA)

**SOA** is nothing but an architectural style where two **non-compatible applications** can **communicate** using a **common language**.

Eg. Windows Communication Foundation (WCF)



The message format can be XML, JSON or even a plain CSV file

# WCF Overview

- WCF stands for Windows Communication Foundation.
- It is a framework for building, configuring, and deploying network-distributed services by which we can send asynchronous message/data from one service endpoint to another service endpoint.
- The basic feature of WCF is interoperability
- WCF is used to build service-oriented applications.
- A WCF application consists of three components –
  - WCF service,
  - WCF service host, and
  - WCF service client.

# Fundamental Concepts of WCF

- **Message** : Message instances are sent as well as received for all types of communication between the client and the service.
- **Endpoint** : It defines the address where a message is to be sent or received.
- **Binding**: It defines the way an endpoint communicates. It comprises of some binding elements that make the infrastructure for communication. For example, a binding states the protocols used for transport like TCP, HTTP, etc., the format of message encoding, and the protocols related to security as well as reliability.
- **Contracts**: It is a collection of operations that specifies what functionality the endpoint exposes to the client. It generally consists of an interface name.

# Fundamental Concepts of WCF

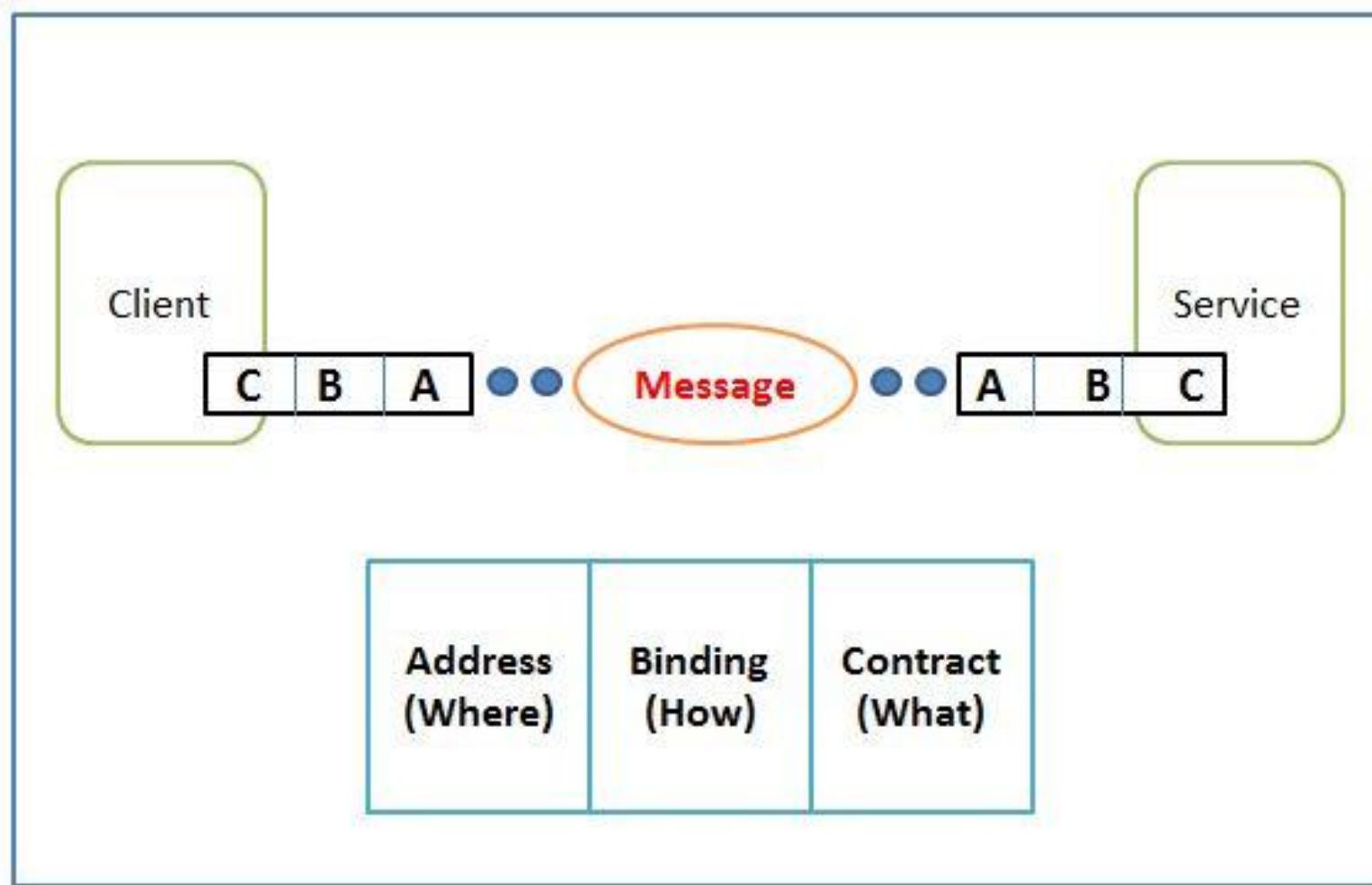
- **Hosting:** refers to the **WCF service hosting** which can be done through many available options like self-hosting and IIS hosting
- **WCF Client:** A client application that **gets created for exposing the service operations** in the form of **methods** is known as a WCF client.
- **Channel:** Channel is a **medium** through which a **client communicates with a service**.
- **SOAP:** Although termed as '**Simple Object Access Protocol**'. It **is an XML document** comprising of a **header** and **body** section.

# Contracts

The contracts layer is just **next to the application layer** and contains information similar to that of a real-world contract that specifies the operation of a service and the kind of accessible information it will make. Types discussed below:

- **Service contract** – This contract **provides information** to the **client** as well as to the **outer world** about the **offerings of the endpoint**, and the **protocols** to be used in the communication process.
- **Data contract** – The **data exchanged** by a **service** is defined by a **data contract**. Both the **client** and the **service** has **to be in agreement** with the **data contract**.
- **Message contract** – A **data contract is controlled** by a **message contract**. It primarily does the **customization** of the **type formatting** of the **SOAP message parameters**. SOAP stands for Simple Object Access Protocol.

# Programming Model of WCF



The ABCs of  
Endpoints

# Programming Model of WCF

- **Address:** "A" stands for Address that specifies where the service is. Every service has a unique address whose format is as follows:

*[Transport] ://[Domain Name]:[Port]//[Service Name]*

- **Binding:** "B" stands for Binding that specifies how to talk with the service. All built-in bindings are defined in the **System.ServiceModel** namespace.
- **Contract:** "C" stands for Contract that tells us about what can service do for me.

# Types of Bindings

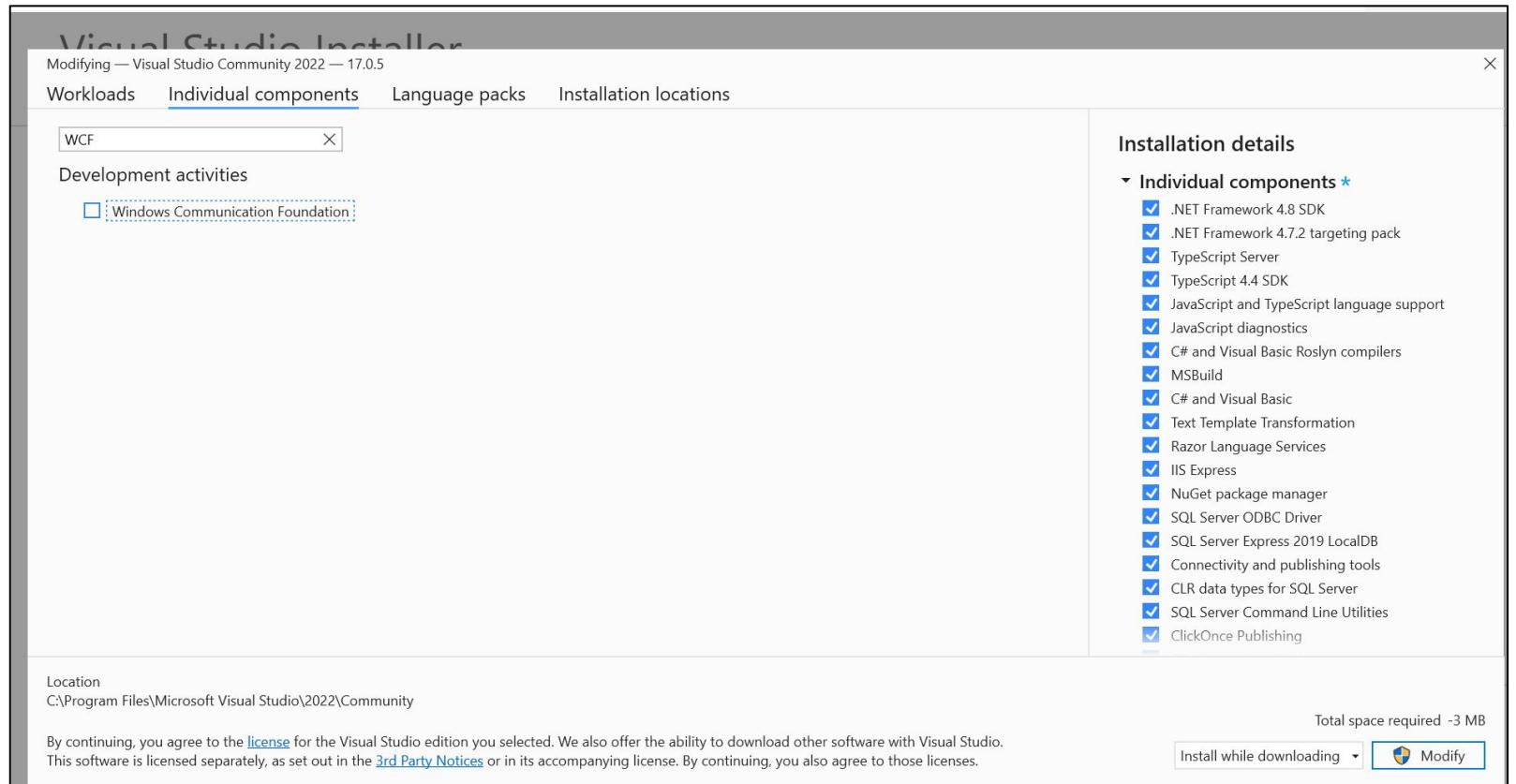
- **BasicHttpBinding:** This is the **basic web service** for the **communicate** with a **web server** or **a client application**. It is designed to expose a WCF service as an **asmx** web service. This binding uses the **HTTP protocol** for communication.
- **NetMsmqBinding:** It is used to **implement message queuing** in a WCF service. It **enables a client application** to **send a message** to a **WCF service** even if the **service is unavailable**.
- **NetTcpBinding:** It allows the communication **between** a WCF service and a **.NET client application** over a **network**. It uses the **TCP protocol** and is a much faster and more reliable binding compared to HTTP protocol binding.

# Types of Bindings

- **NetPeerTcpBinding:** For peer-to-peer communication where messages are sent and received using the TCP protocol.
- **WSHttpBinding:** It is similar to BasicHttpBinding and uses the **HTTP** or **HTTPS** protocol. It also supports WS-Transactions and WS-Security that are not supported by the BasicHttpBinding.
- **WSDualHttpBinding:** It is similar to WSHttpBinding, except it **supports bi-directional communication** which means both clients and services can communicate with each other by sending and receiving messages.
- **NetNamedPipeBinding:** This binding uses named pipes for communication between two services on the **same machine** that is most secure and the fastest binding among all the bindings.

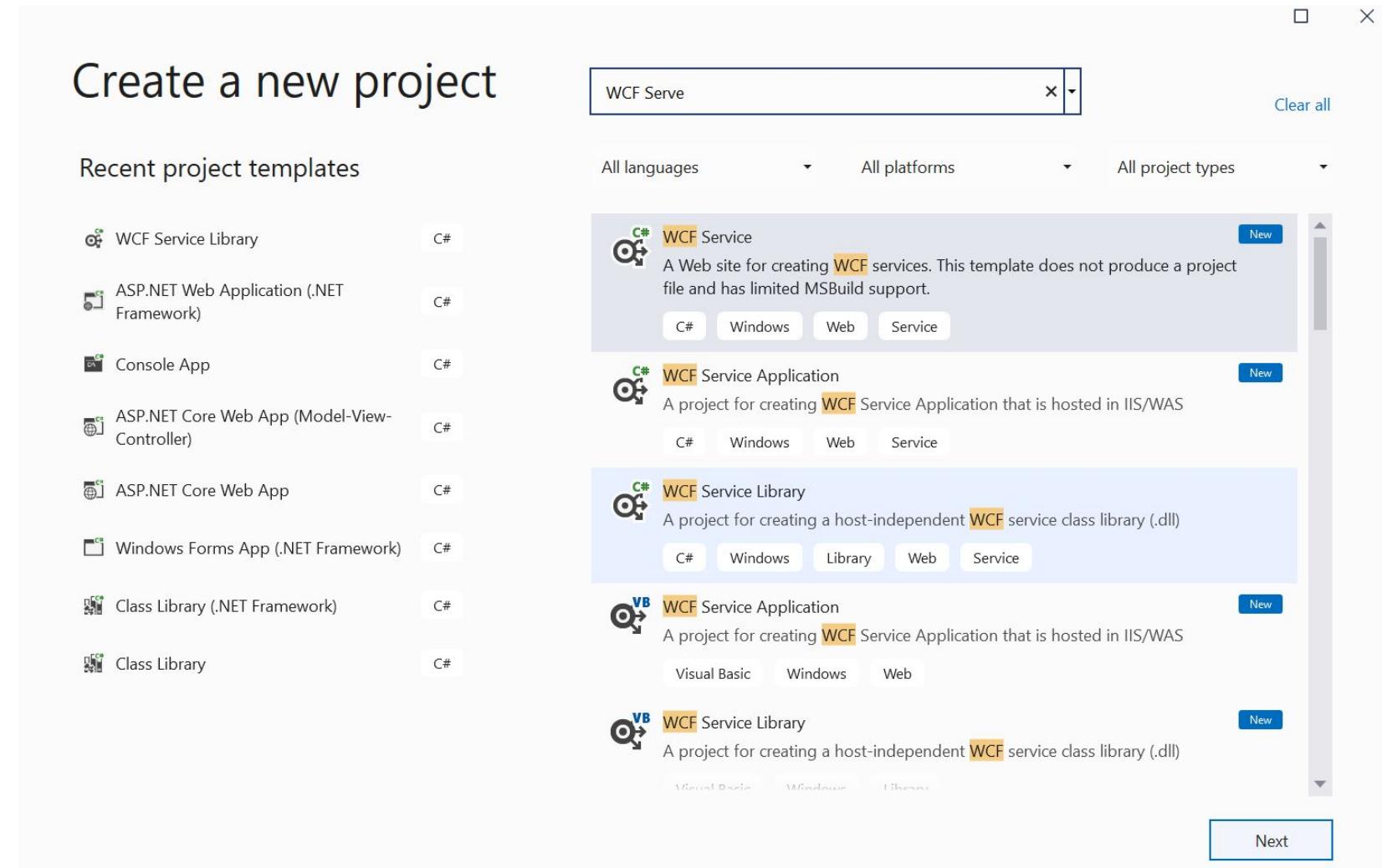
# Adding WCF in Visual Studio

Search ‘WCF’ in the individual components and add **Windows communication foundation** in Visual Studio just like we add previous components



# WCF Service Creation

1. Once WCF installed completely, **Create the New Project and Select WCF Service Library.**
2. Click Next.
3. Then, Give the name for your Project and Click Create



# WCF Service Creation

4. One interface and one class gets added in the project.
5. Check if the bindings and contracts mentioned properly in App.Config file
6. Test and execute

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Runtime.Serialization;
5 using System.ServiceModel;
6 using System.Text;
7
8 namespace DemoServiceLibrary
9 {
10     // NOTE: You can use the "Rename" command on the "Refactor" menu to change this
11     [ServiceContract]
12     public interface IService1
13     {
14         [OperationContract]
15         string GetData(int value);
16
17         [OperationContract]
18         CompositeType GetDataUsingDataContract(CompositeType composite);
19
20         // TODO: Add your service operations here
21     }
22
23     // Use a data contract as illustrated in the sample below to add composite types.
24     // You can add XSD files into the project. After building the project, you can
25     [DataContract]

```

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <appSettings>
        <add key="aspnet:UseTaskFriendlySynchronizationContext" value="true" />
    </appSettings>
    <system.web>
        <compilation debug="true" />
    </system.web>
    <!-- When deploying the service library project, the content of the config file must be added to the host's
        app.config file. System.Configuration does not support config files for libraries. -->
    <system.serviceModel>
        <services>
            <service name="DemoServiceLibrary.Service1">
                <host>
                    <baseAddresses>
                        <add baseAddress = "http://localhost:8733/Design_Time_Addresses/DemoServiceLibrary/Service1/" />
                    </baseAddresses>
                </host>
                <!-- Service Endpoints -->
                <!-- Unless fully qualified, address is relative to base address supplied above -->
                <endpoint address="" binding="basicHttpBinding" contract="DemoServiceLibrary.IService1">
                    <!--
                        Upon deployment, the following identity element should be removed or replaced to reflect the
                        identity under which the deployed service runs. If removed, WCF will infer an appropriate identity
                        automatically.
                    -->
                    <identity>
```

# References

1. <https://docs.microsoft.com>

# **Session-21**

## **Web APIs**

# Contents

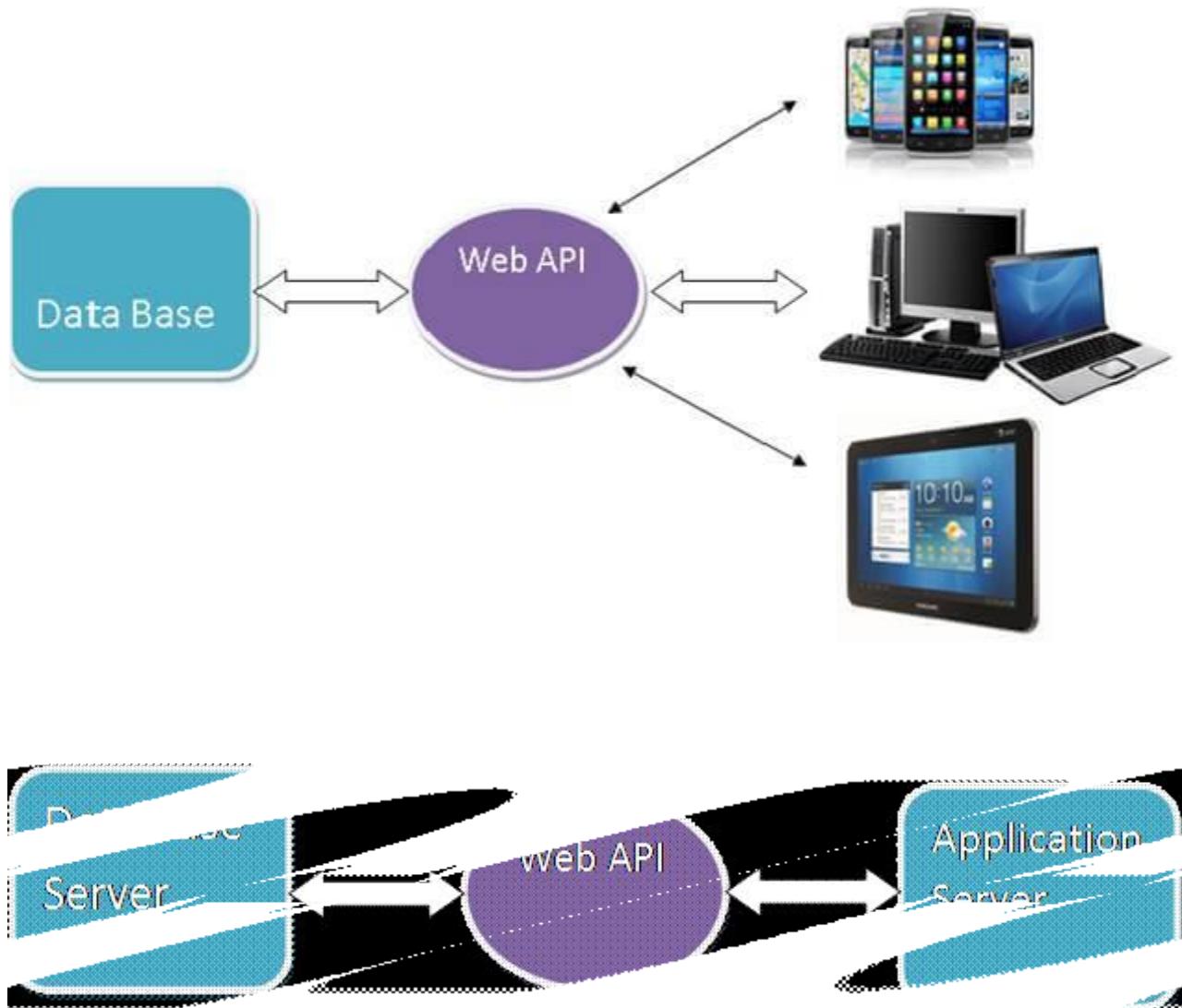
- API
- Web API
- Uses of API
- Creating WebAPI project

# API

An API, or *application programming interface*, is a **set of rules** that define how applications or devices can connect to and communicate with each other.

# Web API

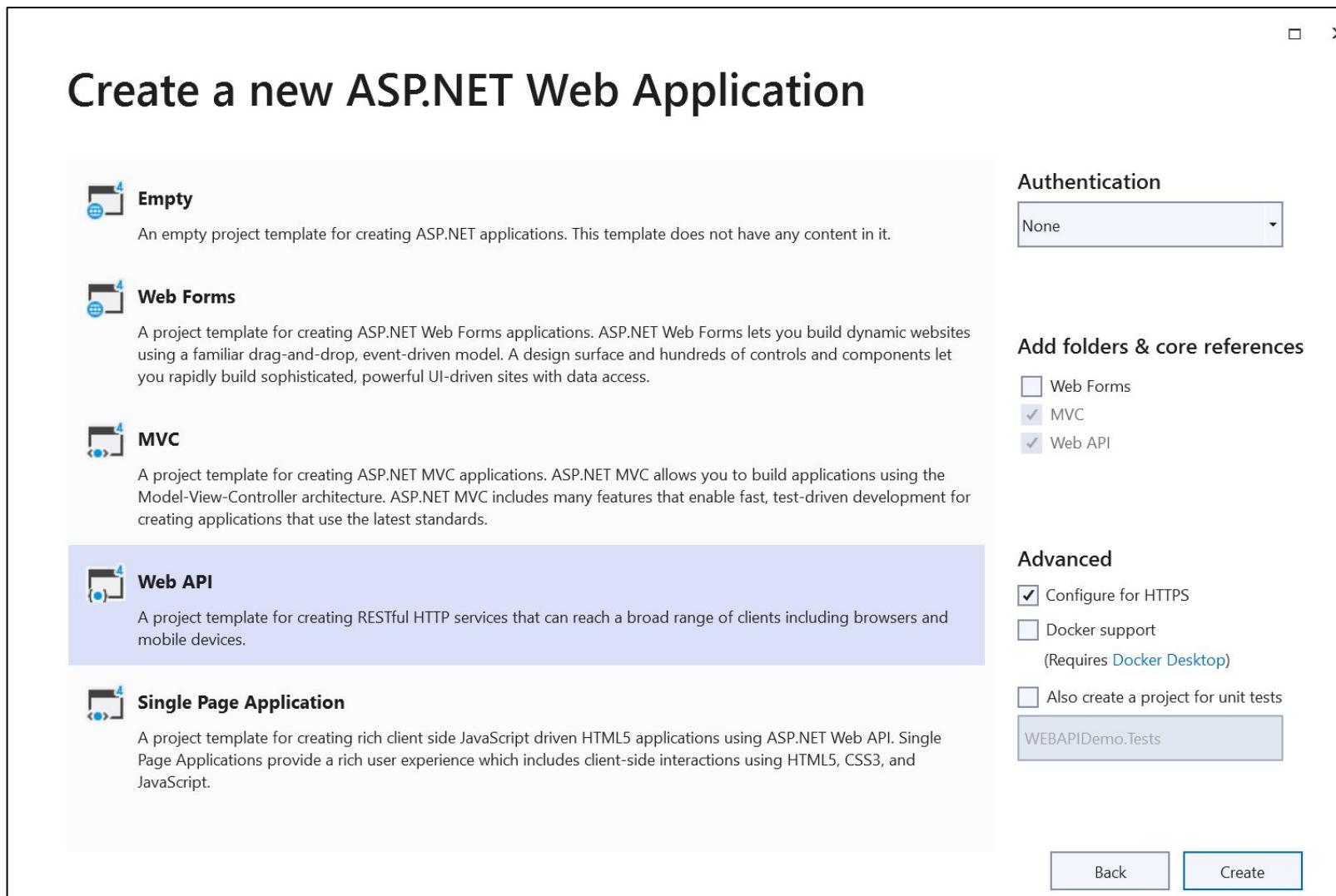
- Web API is an application programming interface (API) that is used **to enable communication or interaction with software components** with each other.
- ASP.NET Web API is a framework that makes it **easy to build HTTP Service** that reaches a **broad range of clients**, including browsers and mobile devices.
- Using ASP.NET, web API can **enable communicating** by different **devices** from the **same database**.



# Uses of Web API

- It is used to **access service data** in **web applications** as well as many **mobile apps** and other **external devices**.
- It is used to **create RESTful web services**. REST stands for **Representational State Transfer**, which is an architectural style for networked hypermedia applications.
- It is primarily used **to build Web Services** that are lightweight, maintainable, and scalable, and support limited bandwidth.
- It is used to **create** a simple **HTTP Web Service**. It supports XML, JSON, and other data formats.

# Creating Web API project



# Creating Web API project

The screenshot shows the Visual Studio interface with the ASP.NET Overview page open on the left and the Solution Explorer on the right.

**ASP.NET Overview Page:**

- Left Sidebar:** Contains links for "Overview", "Connected Services", and "Publish".
- Header:** "ASP.NET" and "Learn about the .NET platform, create your first application and extend it to the cloud."
- Build Your App:** Features a large brace icon ({}), a "Build Your App" button, and links to "Get started with ASP.NET" and ".NET application architecture".
- Connect To The Cloud:** Features a cloud icon with an upward arrow, a "Connect To The Cloud" button, and links to "Publish your app to Azure" and "Get started with ASP.NET on Azure".

**Solution Explorer:**

- Shows the solution "WEBAPIDemo" with one project "WEBAPIDemo".
- Lists the project files: Properties, References, App\_Data, App\_Start, Areas, Content, Controllers, fonts, Models, Scripts, Views, favicon.ico, Global.asax, packages.config, and Web.config.

# Creating Web API project

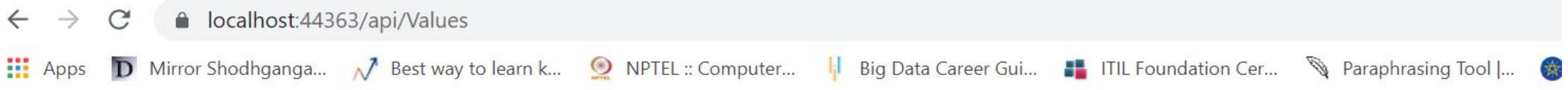
The screenshot shows the Visual Studio interface with the 'WEBAPIDemo' solution open. The Solution Explorer on the right lists the project structure, including files like RouteConfig.cs, WebApiConfig.cs, HomeController.cs, and ValuesController.cs. Two specific files are highlighted with red boxes: RouteConfig.cs and WebApiConfig.cs under the App\_Start folder, and ValuesController.cs under the Controllers folder.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Web.Http;
5
6 namespace WEBAPIDemo
7 {
8     public static class WebApiConfig
9     {
10         public static void Register(HttpConfiguration config)
11         {
12             // Web API configuration and services
13
14             // Web API routes
15             config.MapHttpAttributeRoutes();
16
17             config.Routes.MapHttpRoute(
18                 name: "DefaultApi",
19                 routeTemplate: "api/{controller}/{id}",
20                 defaults: new { id = RouteParameter.Optional }
21             );
22         }
23     }
24 }
25
```

This project is same as default MVC project with two specific files for Web API, WebApiConfig.cs (to configure routes and other things for Web API) in **App\_Start** folder and ValuesController.cs in **Controllers** folder as shown below.

# Creating Web API project

Web API has returned a result in XML or JSON format. Our output looks like below.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
  <string>value1</string>
  <string>value2</string>
</ArrayOfstring>
```



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">value</string>
```

# References

1. <https://docs.microsoft.com>